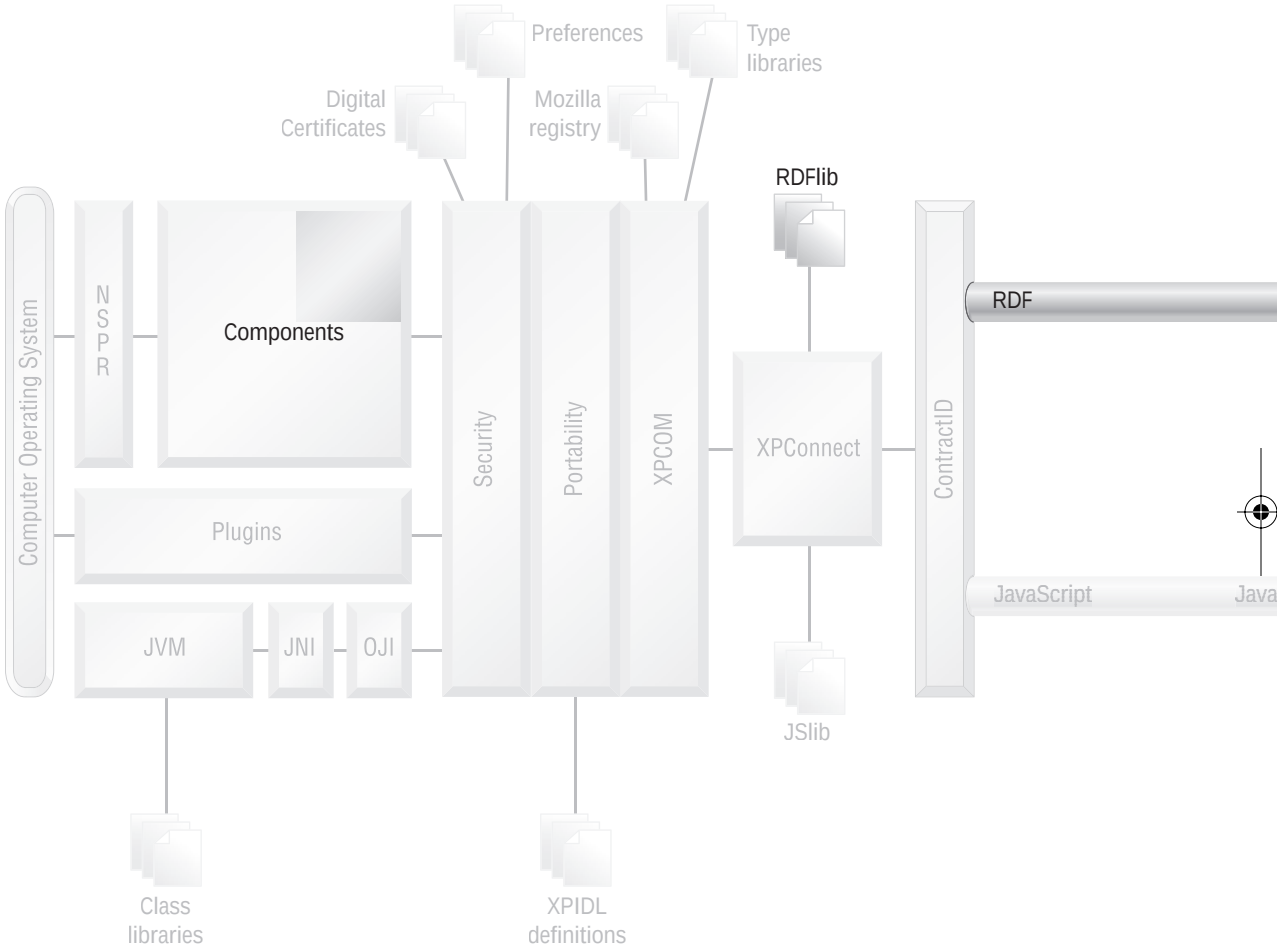
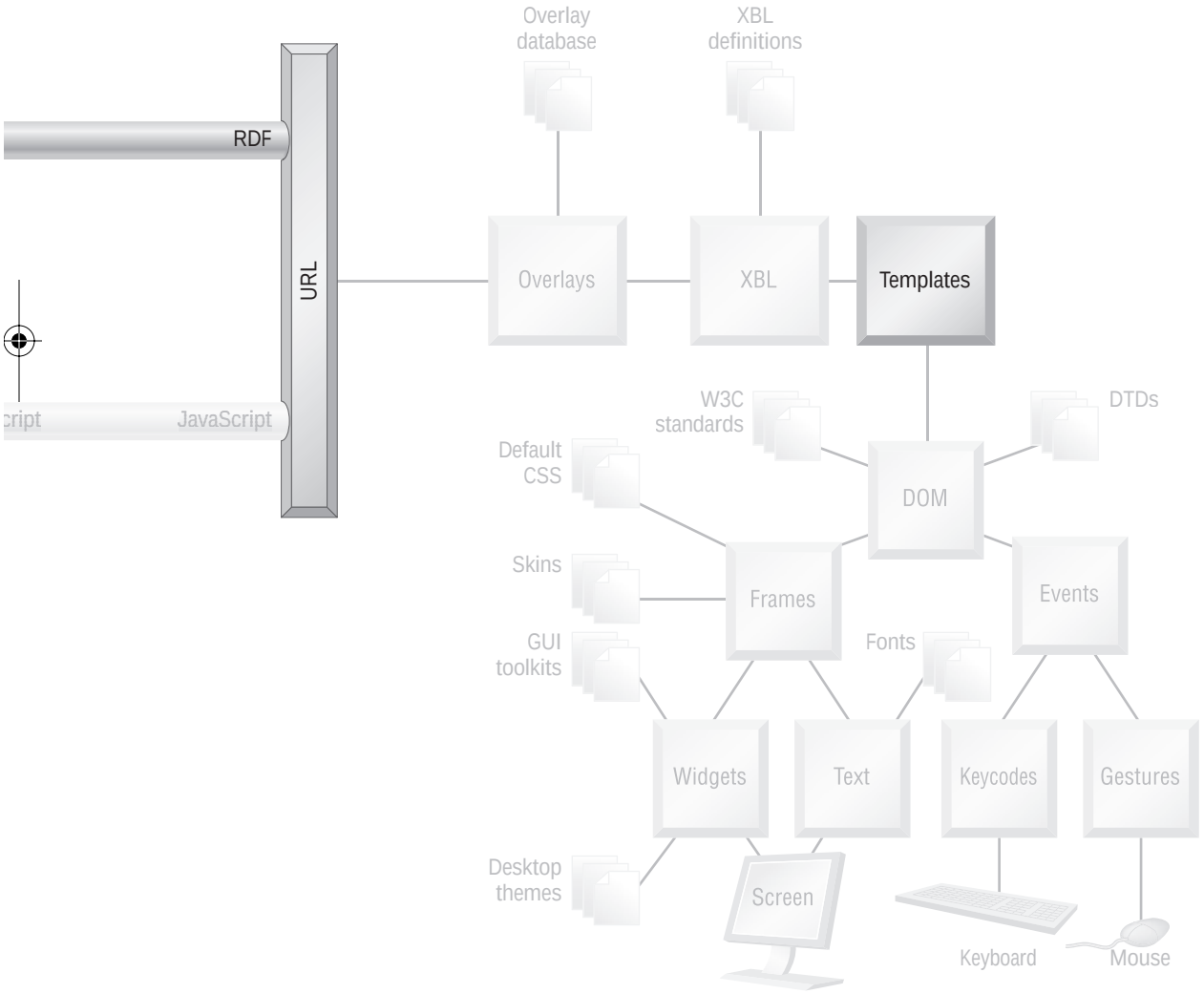

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 14

Templates



This chapter describes how to specify content for a XUL document using a stream of RDF data. This is done with a combination of ordinary XUL tags, XUL template tags, and RDF tags.

Mozilla's template system is a subset of XUL's tags. These tags are used to create a document whose content is not fixed. Such a document is the basis for display of data that varies over time, either because of user interaction, or because of its origin. It is also the basis for applications whose UI depends on external information. That external information might be as simple as a file, as complex as a database, or as remote as a network device. In all cases, such a document has an appearance that varies according to the viewing occasion.

The template system enables many classes of application. When the template-based information is updated by equipment, the user interface acts like a telemetry application, such as a network manager or environmental control system. When the template-based information is updated by the end user, the user interface acts like a data management application. The template system is particularly good at supporting drill-down data management activities, like category analysis, work breakdowns, content and document management systems, and network visualizations. It can also be used for the more traditional data entry style of application.

In the traditional Web application environment, an HTML document with dynamic appearance can be achieved in two ways. HTML can be generated by a program installed behind a Web server (like a CGI program), or existing HTML can be heavily scripted (Dynamic HTML). In either case, 3GL code has to be written to produce the desired results.

Mozilla's template system requires no 3GL code and no Web server. It is, of course, specific to Mozilla. All that is required is an RDF document and a set of rules that state what to do with that RDF. These rules are expressed as XUL tags. The Mozilla Platform automatically pumps the RDF facts into the XUL template document when it is being loaded. The set of rules is used to modify the final content displayed, as directed by the pumped-in facts. The XUL template system is therefore a data-driven system. Some templates require full security access to the platform, such as is provided by the chrome area.

The RDF content consumed by templates has two possible origins. Content might come from an ordinary RDF document stored as a file on some file system. In this case, the content can be RDF facts on any topic. The NoteTaker running example in this book ultimately does that. Alternately, that content might be produced "live" by the Mozilla Platform. In that case, the content consists of RDF facts on a platform-specific topic. An example is window management within the Mozilla Platform. The DOM Inspector consults that internal RDF information in order to build the File | Inspect a Window... menu. This menu consists of currently open windows.

Understanding templates means understanding the template rules system. Sets of rules can be trivial or complex. In the most trivial case, the rules are implicit and not stated directly. In the complex case, rules are a bit like a

database query and a bit like JavaScript `switch` statements. Both cases have the use of special template variables.

Like many aspects of XUL, the template system starts with direct and obvious syntax:

```
<template>
  <rule> ... </rule>
  <rule> ... </rule>
</template>
```

Templates are as complex as trees, and this basic syntax doesn't last long—it has a number of subtle points.

Templates do not carry any content of their own: None of the template tags are boxlike tags. Template tags are more like macro processing instructions and the `#ifdef` features of C's preprocessor. These tags are always used inside some other XUL tag; they are not top-level tags like `<window>`.

The NPA diagram at the start of this chapter shows the extent of the template system. From the diagram, templates are a small system of their own, somewhat separate from the rest of Mozilla's processing. They are the final step in the content assembly process when a XUL document is loaded. Templates have nothing to do with presentation of XUL content. Because templates work intimately with RDF, both RDF files and URL/URI names are heavily used by templates. As for most features of the platform, a few XPCOM objects are responsible for much of the template functionality.

14.1 An Example Template: Hello, World

Listing 14.1 is a XUL document containing a trivial template that implements "hello, world" one more time.

Listing 14.1 XUL application showing "hello, world" use of template technology.

```
<?xml version="1.0"?>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">

  <vbox datasources="test.rdf" ref="urn:test:seqroot">
    <template>
      <label
         uri="rdf:*"
         value="Content: rdf:http://www.example.org/Test#Data"/>
    </template>
  </vbox>

</window>
```

From the listing, the template system consists of its own tags, like `<template>`, and special-purpose attributes like `ref` that are added to other tags.

Mozilla provides several syntax options for rules that make up the template query system. In this example, the most trivial syntax of all is used. Only one template rule exists, and it is implied. That rule says: Process all facts in the nominated RDF container, and generate content to represent those facts. The nominated container has URI `urn:test:seqroot`, and the content to represent the facts is a `<label>` tag. Note that there are nontemplate XUL tags both outside and inside the `<template>` tag.

Listing 14.2 is an RDF file that matches the RDF graph structure that this template expects:

Listing 14.2 Trivial example of an RDF file used for templates.

```
<?xml version="1.0"?>
<RDF
  xmlns:Test="http://www.example.org/Test#"
  xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <Description about="http://www.example.org/">
    <Test:Container>
      <Seq about="urn:test:seqroot">
        <li resource="urn:test:welcome"/>
        <li resource="urn:test:message"/>
      </Seq>
    </Test:Container>
  <Description about="urn:test:welcome" Test:Data="hello, world"/>
  <Description about="urn:test:message" Test:Data="This is a test"/>
</RDF>
```

This RDF file has no special features other than a `<Seq>` tag whose resource name is used in the XUL template code in Listing 14.1. This URN is used as the starting point for the template. `Test` is an `xmlns` namespace. `Data` and `Container` are made-up property (predicate) names for that namespace. The URNs and the `"http://www.example.org/Test"` URL are all equally made up. No formal process is required to allocate these names; just make good design choices. The `Data` property also appears in the `<label>` tag in Listing 14.1, where it is quoted with its full URL.

If these two listings are saved to files, and Listing 14.1 is loaded into the platform, then Figure 14.1 is the result. Diagnostic styles have been turned on to show the structure of the resulting window.

This screenshot shows that two `<label>` tags have been generated in the final XUL. The value of each tag consists of both fixed content (“`Content: `”) and a string that matches one of the two `<Description>` facts in the RDF doc-

Fig. 14.1 XUL document created by a template and two facts.

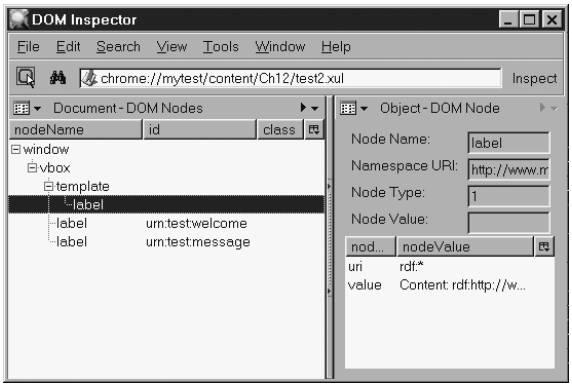


Fig. 14.2 DOM Inspector view of a template-generated document.

ument. By comparison, the <vbox> tag appears only once. The structure of this final document can also be examined using the DOM Inspector. Figure 14.2 shows a full DOM Inspector breakdown of Figure 14.1.

This screenshot shows that the template system has indeed added two tags to the document, one for each RDF fact in the <Seq> tag. The highlighted <label> tag is the original template label; the other two <label> tags are the template-generated content. So the final document contains two subtrees—one for the specification of the template, and one for the generated content. The subtree starting the <template> tag contributes no visual content to the document. When the template system generated the tags in the other subtree, it gave them ids equal to the URN of the resource in the matched RDF fact.

The XUL for this template can be made far more complex by using the extended features of the template rule system.

Before diving into the XUL syntax of the template system, we first take a big step back and consider what it all means from the point of view of facts. After all, templates require RDF content to work on, and that means fact processing.

14.2 TEMPLATE CONCEPTS

Mozilla's template system extends all the standard features of Mozilla applications: XUL, JavaScript, and RDF. There is also the matter of data sources. These extensions are explained and then tied together with an example.

14.2.1 RDF Queries and Unification

The XUL template system is a query system. It searches through data and returns the items that match the search specification. The data searched

through is a set of RDF facts. A template query is therefore an RDF-specific piece of processing.

The queries that XUL templates do are often described as a pattern-matching system. In the most general computer science sense, this is true, but pattern matching also has a simple, everyday meaning. That everyday meaning is used for file name masks like `*.txt` and for regular expressions like "`^[a-zA-Z]*66$`". Tools like command line shells, Perl, `grep`, `vi`, and even JavaScript use everyday pattern matching, which is very similar to simple filtering. In simple filtering, a large list of items (or characters) is reduced to a smaller list.

XUL template queries are not filters and do not do pattern matching in the everyday sense. If an RDF document contains a certain fact, then a XUL template that queries that document can do much more than just choose or ignore that fact. Template queries are not just simple filters.

Instead of simple filtering, template queries do *unification*. Unification is where a set of data items is combined into a final result. Solving a jigsaw puzzle is a real-world example. When all the jigsaw puzzle's pieces are fitted together, the result (a finished picture) is achieved. If there are more puzzle pieces than are needed (perhaps several puzzle sets have been mixed together), then some pieces will be discarded as irrelevant. Unification can have more than one outcome. If there are enough jigsaw pieces, then several pictures might be put together at once, not just one. If the jigsaw pieces are similar shapes, then even more pictures might be possible because of the many possible arrangements of pieces.

In Mozilla, the jigsaw puzzle pieces are the subjects, predicates, and objects of a set of RDF facts. The desired result is stated by a XUL template's query. Any combination of pieces that fits the query specification (the rules in the template) is returned as a number of pieces of information—a new tuple. If more than one combination of pieces fits, more than one tuple is returned.

Such a system might sound familiar. The SELECT statement in SQL acts exactly like this when it contains a join—that is, a FROM clause with two or more tables. The column names in the returned rows are drawn from all the joined tables. The set of returned columns does not exactly match one table row; instead, it matches pieces of rows from several tables. More than one row can be returned in a SELECT query's result set.

In fact, XUL template queries are examples of relational calculus, just as SQL queries are examples of relational algebra. University researchers have shown that these two relational approaches are fundamentally the same, even though they are programmed differently and have little syntax in common.

The XUL syntax for templates is unusual and best avoided to start with. To learn about template queries, let's return instead to the boy, dog, and ball example of Chapter 11, RDF.

Recall that example was used to describe a "pure" fact system, free of any RDF or XML syntax. It is used again here to show "pure" examples of fact queries and fact unification. The example facts from Chapter 11, RDF, are repeated in Listing 14.3.

Listing 14.3 Predicate triples for boy and dog example, from Chapter 11.

```
<- 1, is-named, Tom ->
<- 1, owner, 2 ->
<- 1, plays-with, 5 ->
<- 2, is-named, Spot ->
<- 2, owned-by, 1 ->
<- 2, plays-with, 5 ->
<- 5, type-of, tennis ->
<- 5, color-of, green ->
```

The information modeled is "Tom and his dog Spot play with a green tennis ball," and each thing in the model has an identifying number. We can query this set of facts using a single- or multi-fact query.

14.2.1.1 Single-Fact Queries In Chapter 11, RDF, we noted that facts (and RDF documents) are often ground. Ground facts are a good thing because every piece of a ground fact is just a literal piece of data, with no unknowns. Literals are easy to work with. A fact that isn't ground was also shown in Chapter 11, RDF. That fact, slightly changed, is repeated here:

```
<- 1, owner, ??? ->
```

Because the object part of this triple isn't known, this fact is not ground. It is useless as a piece of data, but it is useful as a starting point for a fact query. Let's use a placeholder variable for the unknown part. Such variables start with a question mark (`?`) just as DOS variables start and end with `%`, or just as UNIX shell variables start with `$`. A variable cannot be in the ground state, or else it wouldn't be a variable—it would be a literal again. We say that the process of turning a variable with unknown value into a variable with a known value is "grounding the variable" or that it is done "to ground the variable":

```
<- 1, owner, ?dogId ->
```

When a query is run, unification causes all variables to be turned into literals from the set of available facts. That is a very fancy way of saying, "Ground All Variables, Please." For this simple example, the second fact stated in Listing 14.3 matches this variable-laden query:

```
<- 1, owner, 2 ->
```

`?dogId` has no value, and if the value `2` were to replace it, a fact matching an existing fact would be constructed. The variable `?dogId` can therefore be ground to the value `2`. This is a trivial example of a query that returns one resulting fact.

Suppose that Tom has a second dog, called Fido. These additional facts would then exist:

```
<- 1, owner, 3 ->
<- 3, is-named, Fido ->
```

If the fact containing ?dogId is again treated as a query, then there are two facts that can fit the query:

```
<- 1, owner, 2 ->
<- 1, owner, 3 ->
```

?dogId can be ground to either 2 or 3, so there are two solutions. We can say the result set contains two facts, two rows, or two items.

The preceding fact that acts like a query can also be expanded. It might read:

```
<- ?personId, owner, ?dogId ->
```

In this case, a match requires a combination of values that satisfies both variables at once (?personId is ground *and* ?dogId is ground). If we use the existing facts, the result set is still two rows: either (?personId is ground to 1 *and* ?dogId is ground to 2), yielding one fact, or (?personId is ground to 1 *and* ?dogId is ground to 3), yielding the other fact. Adding extra variables does not always mean more facts will appear in the results. It just means that more things need to match correctly.

Suppose that Jane (with person id = 4) also owns Fido, but not Spot. Fido is shared, but Spot is exclusively owned by Tom. There are then two more facts in the fact store:

```
<- 4, is-named, Jane ->
<- 4, owner, 3 ->
```

If the last query is posed again, then three solutions result:

- ?personId ground to 1 (Tom) *and* ?dogId ground to 2 (Spot)
- ?personId ground to 1 (Tom) *and* ?dogId ground to 3 (Fido)
- ?personId ground to 4 (Jane) *and* ?dogId ground to 3 (Fido)

Even though ?personId can be set to 4 (Jane) and ?dogId can be set to 2 (Spot), no fact results from this combination because no such fact (Jane owning Spot) appears in the fact store. When a fact is matched against a query fact, there must be a complete fit, not a partial fit.

Finally, note that the information to be ground in this last single-fact query has an alternate notation. It can be written as a collection of unknowns that need to be solved. That collection can be written as a tuple. In this example, such a tuple is a double, not a triple. That tuple-to-be-ground can be stated like this:

```
<- ?personId, ?dogId ->
```

This tuple doesn't describe the query at work. It merely explains what unknowns are involved and therefore what variables will be set when the query returns solutions. That is useful information for programmers, and somewhat similar to the INTO clause of Embedded SQL's SELECT. If someone else is responsible for creating the query, then such a tuple is all you really need to know to grab the results returned.

In the last example, the three solutions found fill this tuple like so:

```
<- 1, 2 ->
<- 1, 3 ->
<- 4, 3 ->
```

The tuple of two variables neatly collects the unknown information together. The query processor still needs information that matches the structure of the facts in the fact store. That is why queries are always posed using the longer syntax shown earlier.

If the query processor isn't given enough information, it can't guess what to do. It is not enough to say, "Smart computer, please fill these variables for me somehow." The query must tell the query processor what to look at. In a single-fact query, the prescription given is very simple: "Dumb computer, examine all triples for a match against this query triple."

Mozilla's template system supports single-fact queries. Single-fact queries can be specified using the template extended query syntax. Single-fact queries cannot be specified using the template simple query syntax.

Looking ahead slightly, to query a single fact, the `<conditions>` tag of the query must be arranged one of two ways. If the facts sought are RDF container containment facts, with predicates `rdf:_1`, `rdf:_2`, and so on, then `<conditions>` should hold a `<content>` and a `<member>` tag. If the facts sought have well-known predicates, then the `<conditions>` tag should hold a `<content>` and a `<triple>` tag. Template queries that are single-fact queries are discussed further under "Common Query Patterns."

In summary, a single-fact query tries to ground a query fact against a real fact, and when it succeeds, the ground results are called solutions.

14.2.1.2 Multifact Queries

The XUL template system supports multifact queries. A multifact query is a bit like an SQL join, and a bit like navigating a tree data structure.

A multifact query poses a question that requires two or more whole facts to be combined. In other words, it requires some deduction. Mozilla's support for deduction is like simplistic Prolog predicate calculus, except the similarity is heavily disguised by syntax. We consider "pure" multifact queries first, before examining the XUL syntax.

Using the boy and dog example, suppose that a needed query is: "What is the name of the dog that Tom owns?" Using variables, that query could be stated as three facts to be unified:

```
<- ?personId, is-named, Tom ->
<- ?personId, owner, ?dogId ->
<- ?dogId, is-named, ?dogName ->
```

Forming a multifact query such as this takes a little practice. It is the same kind of practice required to learn how to do multitable SQL joins or multivariable regular expressions. The core problem is that you are required to write the query straight down direct from your brain.

This particular query was constructed as follows. First, the things we already knew were identified ("Tom"). Next, the things we didn't know, but wanted to know, were noted: `dogName`. We looked at the available tuples to see where these known and unknown things might fit in. That gave us two tuples, the first and third in the final query. Looking at those tuples, we saw what some of the predicates must be: `is-named` is the useful predicate for both tuples. To ground those tuples fully, some other unknowns (the tuple subject terms) would have to be found: `personId` and `dogId`. So we added those variables to the list of unknowns. We noticed that these two tuples don't share any unknowns, so they weren't "connected." We then looked again for more tuples that might connect the ones we really need together. We discovered the second tuple, which links `personId` and `dogId`. With that addition, we were satisfied that all required unknowns were present, and that the set of tuples was connected so that the tuples formed a single query.

If these three facts are submitted as a single query to the query processor, then all variables must be ground at once if a solution is to be found. Because `?personId` and `?dogId` appear in two triples each, whatever value they take on must simultaneously match both triples. In this way, triples with data items in common can be linked (or joined or tied) together. The only possible solution in the example fact store (without Fido or Jane) is:

```
<- 1, is-named, Tom ->
<- 1, owner, 2 ->
<- 2, is-named, Spot ->
```

Compare these three facts with the preceding query. This solution unifies the unknown variables

```
<- ?personId, ?dogId, ?dogName ->
```

to a single set of possibilities

```
<- 1, 2, Spot ->
```

The value `Spot` is the value sought; the other variables are just used to tie the facts together. So Spot is the name of the dog that Tom owns, and the query has been answered. The other variables can be examined or ignored according to need. This multifact query shows why variables undergo "unification" rather than simply being "set": They must all be matched at once before a solution is found.

How does the Mozilla query processor find this solution? There are many possible techniques. The simplest technique is to go though every combination of three facts in the fact store and compare each combination against the query. That is a "brute force" solution and very inefficient. It is not done in Mozilla. Mozilla uses a narrower approach, which involves exploring part of the fact graph structure only. We'll see more on that shortly.

If Fido and Jane are put back into the fact store, then the same query would report two solutions:

```
<- 1, is-named, Tom ->
<- 1, owner, 2 ->
<- 2, is-named, Spot ->

<- 1, is-named, Tom ->
<- 1, owner, 3 ->
<- 3, is-named, Fido ->
```

The values that the unknown variables are ground to are thus

```
<- 1, 2, Spot ->
<- 1, 3, Fido ->
```

Note how the query is designed to match the particular facts in the fact store. Subject and objects are matched up using variables like `?personId`. The results, on the other hand, are just a set of ground variables that make up one tuple per solution. In this example, there are three variables, so the result tuple is a triple. Any number of variables might be used, though.

This example is equivalent to a three-table join in SQL. Listing 14.4 shows an imaginary `SELECT` query that does the same job as the last fact query. Each of the three imaginary tables matches one of the facts in the preceding three-fact query.

Listing 14.4 SQL `SELECT` analogous to a three-fact query.

```
SELECT p.personId, d.dogId, d.dogName
FROM   persons p, owners o, dogs d
WHERE  p.personName = "Tom"
AND    p.personId   = o.personId
AND    o.dogId      = d.dogId
```

Just as the join in the SQL query links tables (relations) together, the variables in a factual query link facts together. Compare the use of `personId` in the two types of query, and you should be able to see a little similarity, even though the syntax is vastly different.

In the general case, this example shows how a big bucket of facts can be examined using a carefully constructed query that contains variables. If the bucket of facts is an RDF document, then clearly Mozilla's templates can do simple but general-purpose queries on that document.

The template system supports multifact queries in two ways. The simple template syntax automatically performs a two-fact query, provided that the RDF data is organized correctly. The extended template syntax allows multifact queries of any length to be created by stringing together one `<content>` tag, any number of `<member>` and `<triple>` tags, and any number of `<binding>` tags. Each such tag (except for `<content>`) represents one fact in the query.

14.2.1.3 Query Solution Strategy

Chapter 11, RDF, explains how a collection of facts can be viewed in one of three ways: as a simple unstructured set of items, as a complex graph-like data structure, or as a simple set that has use-

ful subsets of items identified with RDF containers like `<Seq>`. Mozilla uses a combination of the RDF graph (the second view) and containers (the third view) to get its XUL template queries done.

To the application programmer, it appears that Mozilla uses a drill-down algorithm to solve queries. This drill-down process is equivalent to a depth-first tree traversal. This algorithm requires that the query have a starting point that is known—the *root* of the query. In the preceding example, `Tom` is a known starting point. In practice, the starting point in Mozilla should be a fact subject, as well as a fact object. Such a fact subject is always a URI (a URL or URN).

After this starting point is known, the Mozilla query processor navigates through the graph of the RDF facts from that starting point. Each fact in the query equals one arc (one predicate or property) traversed. So for a three-fact query, the processor will penetrate only three hops into the graph from the starting point. It will only look that far for solutions to the query.

This query system can be visualized using actual RDF graphs. We return to a variation of the boy and dog example. In this variation, the dog Spot has discovered that he can play by himself. He has also managed to get ownership of a softball, which is entirely his. Figure 14.3 shows the graph that results from these additional facts, with a particular query highlighted.

In Figure 14.3, all facts are part of the fact store. The query illustrated is a two-fact query that starts at Tom's identifier, which is one (1). Perhaps the query is something like "What things do Tom's pets play with?" The dark lines on the graph show fact terms that are reached by the query—a two-fact query must traverse exactly two arrows. The pale lines are not reached. There are three initial solutions to this query. Each one matches a unique path that starts from the (1) identifier: path 1-2-Spot, path 1-2-7, and path 1-2-5. To be more specific, we might require that the first fact have an owner predicate (a

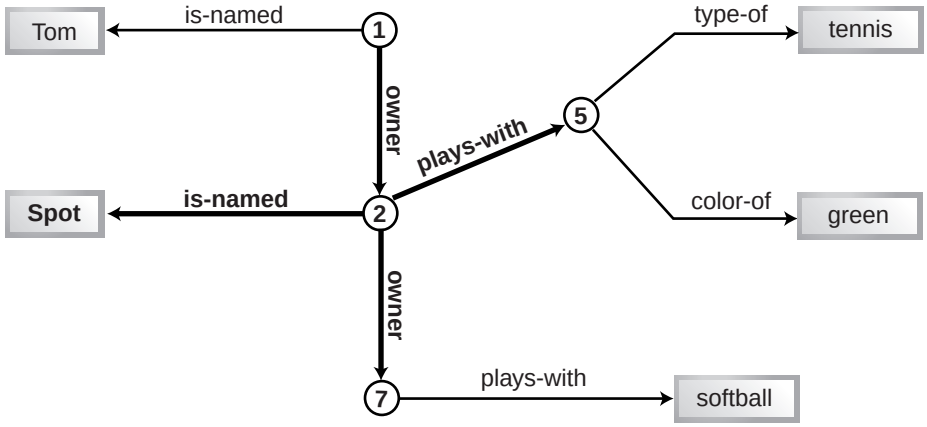


Fig. 14.3 Enhanced boy and dog graph showing a query path.

pet of Tom's) and the second fact have a plays-with predicate. In that case, the 1-2-Spot combination would no longer be a solution, and just two solutions would be found: "Tom's dog Spot plays with a softball" and "Tom's dog Spot plays with a tennis ball."

We can experiment a little further with this example. Figure 14.4 shows a different query on the same graph of facts.

This is again a two-fact query, but this time it starts at Spot's identifier (2). There are again three solutions. This time, note that the path 2-1-Tom is not a solution. This is because the arrows point the wrong way in that path. Rather than 2 being a fact subject and 1 being an object, it is the other way around. The query system can't go in reverse like that. Even for the solutions possible, this second query probably doesn't make much sense. The predicates in the possible solution paths are all quite different. If this query seemed necessary, it would be fair to guess that either the data model behind the facts was wrong, or the query was poorly thought up.

The disadvantage of this system is that not all the facts in the fact store are looked at. In theory, a query might miss some of the solutions. The advantage of this system is speed. In practice, if the RDF facts are neatly ordered, a quick search from a known starting point is sufficient to provide all answers.

This drill-down approach is part illusion; the template system is actually more complicated than that. It is, however, a good enough approximation and is a recommended way to interpret XUL template code.

The way to organize an RDF document so that this system works is to use an RDF container: a `<Seq>`, `<Bag>`, or `<Alt>`. The known starting point provided to the query system could be the URI of the fact holding the container. The sought-after data should be facts within that container. The query then drills down into the container, retrieving the required facts. This is an indexing strategy.

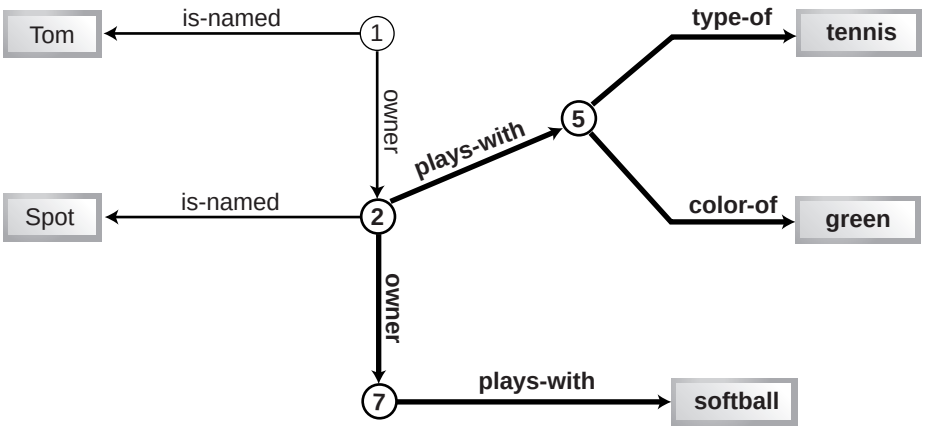


Fig. 14.4 Enhanced boy and dog graph showing second query path.

Listings 14.5 and 14.6 show an RDF fragment and a matching query.

Listing 14.5 Example RDF fragment useful for a multi-fact query.

```
<Description about="http://www.example.org/">
  <NS:Owns>
    <Bag about="urn:test:seq">
      <li resource="urn:test:seq:fido"/>
      <li resource="urn:test:seq:spot"/>
      <li resource="urn:test:seq:cerebus"/>
    </Bag>
  </NS:Owns>
</Description>
<Description about="urn:test:seq:fido" NS:Tails="0"/>
<Description about="urn:test:seq:spot" NS:Heads="1"/>
<Description about="urn:test:seq:cerberus" NS:Heads="3"/>
```

NS in Listing 14.5 stands for some namespace, presumably declared with `xmlns` elsewhere in the code. The namespace would have some URL like *www.test.com/#Test*. Listing 14.6 uses the NS namespace to identify the fact items equivalent to those in Listing14.5. Use of NS in Listing 14.6 has no particular meaning because the facts stated in that listing are neither XML nor code.

Listing 14.6 Example RDF query for drilling into an RDF container.

```
<- http://www.example.org/, NS:Owns, ?bag ->
<- ?bag, ?index, ?item ->
<- ?item, NS:Heads, ?heads ->
```

In this query, the first fact to be ground drills down to the RDF fact with the bag as subject (one fact only). The second fact drills further down to facts with bag *items* as subject (three facts for the sole bag). The third fact drills down to the fact stating how many heads that bag item has (zero or one fact per bag item, depending on whether the Heads property exists). The end result is that two solutions are found. The set of variables to be unified are

```
<- ?bag, ?index, ?item, ?heads ->
```

and the two solutions found are

```
<- urn:test:seq, rdf:_2, urn:test:seq:spot, 1 ->
<- urn:test:seq, rdf:_3, urn:test:seq:cerberus, 3 ->
```

Recall that RDF automatically assigns predicate names starting with `rdf:_1` to each of a container's children. The first container child was not reported by this query because the last of the three query facts couldn't be matched to a fact with `NS:Tails` instead of `NS:Heads`.

Support for this kind of query is the first priority of Mozilla's template query system. This is the most reliable and useful way to work with templates.

In the preceding example, the `?index` variable is used to stand in for the predicate of a fact. Mozilla's query system cannot use a variable for a

predicate term, but it has a <member> tag that can achieve a similar but more limited effect.

In summary, by poking into an RDF graph of facts from a certain point, a set of query facts can be tested for matches against a neatly ordered collection of facts near that point.

14.2.1.4 Stored Queries The queries that XUL templates specify live as long as the document they are contained in. They do not run once and are then thrown away. They are useable until the window that they are displayed in is torn down.

A collection of RDF facts (perhaps loaded from a file) can be modified anytime after it is created because such a collection is stored in memory in a fact store. Facts can be added to the store, removed, or changed. The template query system can be advised of these changes.

When fact changes occur, each template's query can update its understanding of what solutions exist for that query. If new solutions are possible, they are added to the set of results. If some solutions are no longer possible, they are removed from the result set. Updates to existing solutions are also possible.

The net result is that the solution sets can change over time. Template queries are not always passive "read consistent views" (to use RDBMS jargon). They can be live and fully up-to-date actively changing "event lists" (to use telemetry jargon).

This live updating requires a little scripting assistance from the application programmer. In the end, the query must repeatedly poll (check) the fact store if the solutions are to be kept up to date. Scripts can be written so that this is done in an efficient way, and only when required.

In a XUL document, template queries hang around.

14.2.1.5 Recursive Queries The drill-down strategy used by Mozilla template queries can be applied recursively. Each drill-down end point for a query can be reused if it yields a solution. That end point can be used as the new starting point for a repeat of the same query. This allows the query system to drill down further into the RDF graph structure, possibly finding another solution or solutions. Only when no additional facts exist to consider is the end point of the recursion reached.

This recursive use of queries is useful for treelike data structures. Such structures can have any depth, which equals any number of arc hops in an RDF graph. Without recursion, it wouldn't be possible to retrieve the whole of a tree because the depth searched by a query equals the number of facts in the query.

In a XUL document, template queries inside <tree> tags do extra work.

14.2.1.6 Query Lists The Mozilla template system allows a number of queries to be put together into a simple list.

When query processing starts, all queries in such a list are run at the same time. Any solution found that fits more than one query will be assigned to just one of those queries. The query chosen will be the one nearest the head of the list.

The way this is done is simple: For each drill-down hop into the RDF graph, the query processor examines the query list for queries that are part-solved by the facts found so far. Those queries that are part-solved are considered again after the next hop, while the rest are ignored from that point on. When the query processor has finished drilling down, only those queries completely solved by the found facts remain.

This system allows a set of facts to be assessed according to one set of criteria (one query) and, if that fails, to be reconsidered according to another set of criteria (a second query). This is much like the boolean conditions in a series of `if … else if …` statements.

Query lists in Mozilla allow several completely different sets of content to be generated from the one set of facts. Each query can provide fact solutions for one such set.

That concludes template queries. After the query succeeds, something needs to be done with the solution.

14.2.2 XUL Content Generation

In the ordinary case, all a XUL template does with retrieved RDF data is display it.

A template does this by mixing the retrieved data with ordinary XUL content. It acts like a simple pretty-printer or like a report-writing tool. The output content is contributed to the XUL document and appears for the user's consumption, just like any XUL content.

If the XUL tag surrounding the template is a `<menupopup>`, `<listbox>`, `<toolbar>`, `<tree>`, or even `<box>`, then the contents of that XUL tag (the menu items, toolbar buttons, tree or listbox rows) can be generated entirely by the template. This means that interactive interfaces can derive directly from an RDF file, rather than be hard-coded.

A simple example illustrates this generation process. Listing 14.7 is an RDF file containing two facts inside a container. Each fact has a single property/value pair. These pairs are usually the interesting part of an RDF document, and they are usually displayed as XUL content.

Listing 14.7 Simple RDF document used to illustrate displayed content.

```
<?xml version="1.0"?>
<RDF xmlns:Test="http://www.test.com/Test#"
     xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <Description about="urn:test:top">
    <Test:TopSeq>
      <Seq about="urn:test:seqroot">
        <li resource="urn:test:message1"/>
```

```
        <li resource="urn:test:message2"/>
      </Seq>
    </Test:TopSeq>
  </Description>

  <Description about="urn:test:message1" Test:Foo="foo"/>
  <Description about="urn:test:message2" Test:Bar="bar"/>
</RDF>
```

The two properties can be dug out of this file and displayed using a template. Figure 14.5 shows two sets of template-generated content from the same RDF file. This requires two templates in the same XUL document.

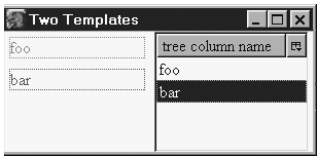


Fig. 14.5 Output from two templates using the same RDF data.

The two templates are side by side in this screenshot. Only the final results of the template content generation are visible. The template on the left generates `<description>` tags with a border style applied. Those tags are contained inside a `<vbox>`. The template on the right generates `<treeitem>` tags, each with `<treerow>` and `<treecell>` content. Those tags are contained inside `<tree>` and `<treechildren>` nested tags. It is easy to see that the content extracted from RDF is the same in both cases, but the structure, appearance, and use of the data are different. For example, the tree items are user-selectable but the plain text on the left isn't.

In summary, templates merge separately stored solution data with other content that determines how the data are to be presented.

14.2.2.1 Template and Content Subtrees The XUL tags that make up a template are not displayed in the final, displayed document. The content tags generated by the XUL system are displayed. Both types of tags exist in the XUL document at the same time, but the template tags are styled to have no display.

The template tags form one DOM subtree of the XUL document. After the template has generated its content, these tags exist only for reference purposes. The template system might reread them at a later point if the application programmer adds scripts that make this happen.

The generated content tags form one DOM subtree for each solution found by the template query. Three solutions mean up to three subtrees. These subtrees occur where any user interaction or scripting support ultimately occurs.

Each of these generated subtrees has a topmost tag. That tag acquires a unique id attribute that matches the URI of a fact subject from the query results. This id acts as a unique key and is used to identify each generated subtree. This id comes from the `uri` attribute, described under content tags.

The "hello, world" example at the start of this chapter shows these different subtrees in a screenshot.

14.2.2.2 Dynamic Content Templates can change over their lifetimes. The content generated by a template can also change over the template's lifetime. Both effects require the use of JavaScript, and both might require a template rebuild. Templates can also delay the generation of content.

Rebuilding a template requires a line of JavaScript code. No document or page needs to be fetched or reloaded. The portion of the XUL document containing the template and its results is changed in-place. Surrounding content is unaffected, except possibly for changes to layout. A typical line of code that does this is

```
document.getElementByTagName('tree').database.rebuild();
```

The template tags themselves can be altered using DOM 1 operations like `removeChild()` and `appendChild()`. If this is done, then the next time the template content is rebuilt the displayed content will be replaced with content generated by the changed template.

If the template is not altered, the content displayed can still change. This occurs when the RDF data that the template uses is changed. If the template content is rebuilt, then content associated with new query solutions will be added, and content associated with old solutions that are no longer relevant will be removed. The template system does these content changes itself using DOM 1 operations.

In a XUL document, the queries not only hang around but also can be reused anytime.

14.2.2.3 Lazy Building Content generation can be delayed until later. This is only possible for recursive queries. When this system is used, only the topmost, most immediate solutions to a template query are sought. These immediate solutions yield content. Later on, if an indication comes from the user or the platform that more content is needed, further solutions are sought. Any extra content generated from those further solutions is added to the displayed results.

Lazy building only works if the template content tag that holds the `uri` attribute is one of these tags:

```
<menu> <menulist> <menubutton> <toolbarbutton> <button> <treeitem>
```

The children of these tags, like the `<menu>` tag in a `<menulist>`, are the pieces built lazily.

Templates that use <tree>s or <menu>s can put off some of the recursive query work until later.

14.2.2.4 Common Source Content Two or more templates can use the same RDF data. If the RDF data changes, then those changes can be reflected in all the templates simultaneously. The templates must be rebuilt for the changes to appear. Each template can be like a big observer of the RDF fact data.

This feature of templates is very powerful and useful. It allows multiple views of a set of data to be displayed at the same time, and all are kept up to date. This is particularly useful for applications that use a desktop metaphor. Examples are design tools and Integrated Development Environments (IDEs). These applications have power users who appreciate being able to visualize the data they work with in several different ways. This is also used in the Classic Mozilla/Netscape browser suites, in the bookmarks system, address book, and elsewhere.

To see this coordination at work, perform the following test:

1. Start the Classic Brower so that a Navigator window appears.
2. Make sure that the Personal Toolbar and Navigation Toolbar both appear (View | Show/Hide).
3. Create a bookmark on the Personal Toolbar by dragging any URL onto that toolbar (drag the bookmark icon to the left of the Location textbox).
4. Display the Bookmark Manager window (Bookmarks | Manage Bookmarks).
5. Make sure that the new bookmark is visible in both the Bookmark Manager and the Personal Toolbar at the same time.
6. In the manager, delete the new bookmark by selecting it and choosing Delete from the right-click context menu.
7. The new bookmark disappears from both the toolbar and manager at the same time.

The Bookmark Manager window contains a template based on a <tree> tag. The Personal Toolbar contains a template based on a <toolbar> tag. The Delete menu item runs a script that deletes the facts that hold information for the new bookmark. That script then causes both templates to rebuild themselves. The piece of content associated with those deleted facts (a <treeitem> in one case, and a <toolbarbutton> in the other) disappears everywhere as a result.

14.2.3 JavaScript Access

The template system adds objects to the AOM of the XUL document. The template system also uses a number of XPCOM components and interfaces, particularly RDF ones. These can be manipulated from JavaScript. JavaScript can perform any of the following tasks:

- Use the `database` AOM object property to manipulate facts and the sources of the RDF data used by the template.
- Use the `builder` AOM object property to control the template build process.
- Create a custom view for a template.
- Fill a template with facts when it has none to start with.
- Add observers to the template system.
- Control sorting of `<tree>`- and `<listbox>`-based templates.
- Use the DOM 1 standards to modify template tags (occasionally done) and template-generated tags (unwise).

These tasks are all described under "Scripting" in this chapter. These tasks often require working with the XPCOM components that support RDF. Those RDF components are discussed in Chapter 16, XPCOM Objects.

14.2.4 The Data Source Issue

A final technical aspect of templates is *data sources*.

The RDF data that templates use can come from an ordinary RDF document or from a preexisting source inside the Mozilla Platform. In both cases, an object called a data source sits between the template processing code and the true origin of the RDF facts.

Mozilla templates can draw facts from more than one data source at a time. This means that facts from several RDF files (for example) can contribute to the content generated by a single template. There is a many-to-many relationship between data sources and templates.

Data sources are discussed extensively in Chapter 16, XPCOM Objects. Here we simply note that the choice of data source for a template is absolutely critical. If the wrong data source is chosen, very little will work. Know your data sources.

Here is a brief overview of the main points. Each template has a composite data source. To work on the template's data source from a script, it is often necessary to find and use one of the data sources that contribute to the composite data source. Such a data source gives the programmer access to the fact store containing the RDF facts of that source. In principle, a full range of database-like operations is possible. In practice, only the data sources associated with RDF files and with the bookmark system are highly useful. The bookmark system has the drawback that it is cryptic. Of the other internal data sources, some are easily read and some are not. The `rdf:null` data source is a convenient choice when the programmer wants to start with an empty set of facts and fill it by hand. It is often used when a custom view is built.

The scripting topic in this chapter describes some of the common template manipulations that require data sources. Chapter 16, XPCOM Objects, is also recommended.

14.3 TEMPLATE CONSTRUCTION

This discussion explains how templates are put together, and describes the individual tags. The overall composition of a template is shown in Listing 14.8, which is pseudo-code, not pure XUL:

Listing 14.8 Basic containment hierarchy for template tags.

```
<top>
  <stuff/>
    <template>
      <rule>
        ... simple or extended rule info goes here ...
      </rule>
      ... zero or more further <rule> tags go here ...
    </template>
  <stuff/>
</top>
```

The tag named <top> can be any ordinary XUL tag—<top> is not a special tag. Although the template proper starts with the <template> tag, some attributes specific to templates must also be added to the <top> tag. Other XUL content can precede, follow, or surround the <top> tag, whatever it is. Typical candidates for <top> are <tree>, <toolbar>, <menulist>, and <listbox>, but <top> could be a <box> or even a <button>.

The tags named <stuff> can also be any ordinary XUL tag—<stuff> is not a special tag. These tags are optional and can be repeated and nested as much as required. The tags between <top> and <template> are copied and generated only once for each template, and they are all displayed before the template content, even if they appear after the <template> tag in the XUL.

The <template> tag is a real XUL tag. Any content within this tag is generated once per query solution found. The <template> tag surrounds all content that might be repeated.

The <rule> tag is a real XUL tag. It is the only content allowed inside <template>. There can be one or more rule tags, and there is a shorthand notation that allows for zero rule tags. Each rule tag is one template query, as described under “Query Lists” earlier. Each rule tag also holds content. This content is duplicated each time a solution for the rule query is found.

The template system has several different syntaxes for the content of the <rule> tag.

The most flexible and powerful syntax is the *extended template syntax*. This syntax requires that template query variables be defined in one spot and then applied later in another spot. This system uses variables called *extended template variables*. The extended syntax requires that each <rule> contain a set of specialist template tags as part of its content.

A convenient and short syntax is the *simple template syntax*. This syntax is designed for the special but common case where the query is looking

through an RDF container for data. The "hello, world" example in Listings 14.6 and 14.7 uses this syntax. This syntax uses variables called *simple template variables*. The simple syntax requires that each <rule> contain only plain XUL content. That content might contain simple template variables. The simple system generates that content when a query solution is found, replacing the variables with fact data in the process.

If a template has only one rule, then the simple syntax has a shorthand version. The <rule> and </rule> tags can be left out. This shorthand version is otherwise the same as the simple syntax.

See the <rule> tag for more detail on rules.

14.3.1 Special XUL Names

The XUL template system uses a number of special literal values. Template variables are used nowhere except inside the <template> tag.

14.3.1.1 Extended Template Variables Mozilla's extended template variables are the variables used to create single fact queries and flexible multifact RDF queries. Such variables always appear inside XML strings.

Extended template variables start with a question mark ("?") and can contain any character. They are case-sensitive. They end with either a space (" ") or a caret or circumflex ("^") or by the termination of the string that they are embedded in. The space or caret is not part of the variable name. If a space is detected, it is left as the first nonvariable name piece of content. If a caret is detected, it is silently consumed, and has no further role.

The following ordinary names are identical. The third example has an XML character entity reference for space.

```
"?name " "?name^" "?name&#x20;"
```

These further names are also valid variable names. Meaningful names are always recommended over unreadable names, though.

```
"?name_two" "?nameThree" "?name-four" "?name66" "?66name" "?$%@$z+"
```

14.3.1.2 Simple Template Variables The simple template notation for rules (see "<rule>") has its own "variables." These variables are really just the URIs of fact predicates. Such variables always appear inside XML strings.

Simple template variables have the format:

```
rdf:URI
```

Such variables end with either a space (" ") or a caret ("^"), or by the termination of the string they are embedded in. The space or caret is not part of the variable. If a space is detected, it is left as the first nonvariable piece of content. If a caret is detected, it is silently consumed, and has no further role.

The URI part of a simple template variable should be a valid URI. If it is to be processed meaningfully, it should be constructed to suit the context in which it is used.

Examples of meaningful URIs are

```
"rdf:urn:test:example:more"
"rdf:http://www.test.com/Test#Data"
```

14.3.1.3 Variable Interpolation Both extended and simple variables are used only inside XML attribute values. In the content part of a template rule, variable names are replaced with content when content is generated.

If a rule's query finds a solution, content generation will follow. When that happens, variable names are simply replaced with their values in the strings they contain. If the variable name has no ground value as a result of the query (possible if a `<binding>` tag is used), then it is replaced with a zero-length string.

14.3.1.4 Special URIs and Namespaces The template system uses a few special URIs and namespaces. The URI scheme `rdf` is used to represent RDF data that originates from inside the Mozilla Platform itself. The currently implemented URIs in this scheme are

```
rdf:addressdirectory rdf:bookmarks rdf:charset-menu
rdf:files rdf:history rdf:httpindex rdf:internetsearch
rdf:ispdefaults rdf:local-store rdf:localsearch
rdf:mailnewsfolders rdf:msgaccountmanager
rdf:msgfilters rdf:smtp rdf:subscribe
rdf:window-mediator
```

There are two special values for this `rdf` URI scheme. The URI

```
rdf:null
```

means use a data source that contains zero facts. Such a data source usually has facts added later via JavaScript. The URI

```
rdf:*
```

is Mozilla-specific notation that means "match any and all predicates." The use of asterisk ("`*`") is inspired by the use of asterisks in CSS2—in CSS2, * also means "match all." This URI should be seen as a special case of a simple template variable. It does not identify one fixed resource. There is also an old notation for this special URI:

```
...
```

This notation is identical to the ellipses (three dots) character, except that it consists of three separate, single, full-stops (periods). This old notation means the same thing as `rdf:*`, but it should not be used any more.

Table 11.3 lists a set of XML namespaces that Mozilla uses for the RDF facts managed by the platform. If templates use Mozilla-internal data sources, then these namespaces can be used to identify predicates/properties within those data sources.

It is common in XML to use the `xmlns` attribute to provide a shorthand alias for a long namespace URL. Within the template system, in nearly all cases, the full URL, not an alias, must be used. This is not a consequence of any XML standard; it is just the way the Mozilla Platform works. XML namespaces do not perform alias substitution inside attribute values, so there is no help from XML itself. That leaves detection of those names up to the platform. Mozilla does not know how to detect or expand `xmlns` aliases or relative URLs inside an attribute value, so full URLs are required. The only place where an `xmlns` alias can be used is as an attribute of the `<rule>` tag. In that case, the alias is used in the attribute name, not in the attribute's value, where it is handled by standard XML parsing.

14.3.2 The Base Tag

The topmost template tag is the parent tag of a `<template>` tag. It is called the base tag of the template. It is an ordinary XUL tag like `<tree>` or `<box>`. This tag must carry part of the template configuration information if the template is to work.

The special attributes that can be added to such a topmost tag are

```
datasources flags coalesceduplicatearcs allownegativeassertions
     xulcontentgenerated ref containment
```

The `datasources` attribute states what RDF data are to be used in the template. It is a space-separated list of RDF file names like `test.rdf`, and named data sources like `rdf:bookmarks`.

Because the `datasources` attribute takes one or more arguments, it is always a composite data source (with interface `nsIRDFCompositeDataSource`).

If the XUL document is installed in the chrome, or is otherwise secure, then the internal data source `rdf:local-store` is automatically added to the start of the data sources list. This data source adds the current user profile's `localstore.rdf` configuration information. This is important because it is common to work on the data sources of a template from a script, and so the application programmer must remember that this data source is present.

The `flags` attribute is used to optimize the performance of the template query process. It applies only to recursive queries and accepts a space-separated list of keywords. Two keywords are currently supported:

- `dont-build-content`. This keyword is specific to tree-based templates. It tells the standard template builder to drop responsibility for sending generated content to the display. Instead, the tree builder is responsible for that. The template-building system still generates RDF-based content, but it acts as a view that the tree builder uses. Chapter 13, Listboxes and Trees, describes the different builders. The benefit of this system is that generating content is put off until it needs to be displayed.

This is efficient for systems where generating content is expensive, such as querying a directory server. It also prevents the template system from "flashing" content to the screen once before the query has generated its own content.

- `dont-test-empty`. This keyword tells the query processor not to examine containers to see if they are empty. This is a performance optimization that saves doing a test that can be expensive. It also allows the template system to handle specialized hierarchical data where testing for empty is impossible. Dynamic network discovery is a practical case where such a situation might exist. In that field, it is impossible to answer the question: "Is the number of network elements out there zero?" It is impossible because the code must wait forever to be sure that no answer arrives. `dont-test-empty` is a good choice for templates based on the `rdf:null` data source.

`coalesceduplicatearcs`, `allownegativeassertions`, and `xulcontentgenerated` are also performance tweaks and further modify how queries work.

- The `coalesceduplicatearcs` attribute, when set, affects the facts that can be extracted from a template's set of data sources. Most flaglike attributes in Mozilla are set to `true` when specified, but this attribute is set to `false`. It affects JavaScript access to facts, not the results of template queries. If this attribute is not set, identical facts inside the template's data sources will be reported only once, not once for each copy. If it is set to `false`, then all facts are reported, duplicate or not. Template data sources go faster if this attribute is set.
- The `allownegativeassertions` attribute, when set, also affects the facts that can be extracted from a template's set of data sources. Most flaglike attributes in Mozilla are set to `true` when used, but this attribute can also be set to `false`. It also affects JavaScript access to facts, not the results of template queries. If this attribute is not set, a fact that is stated (*asserted*) both positively and negatively will never be reported because the two facts cancel each other out. RDF documents contain only positively asserted facts. Negatively asserted facts can be made using JavaScript only. If this attribute is set, no cancelation is done, and all facts are reported. Template data sources go faster if this fact is set.

The `xulcontentgenerated` attribute is applied to any content tag in the template, or any generated content tag produced by the template. It is listed here because it is also a performance optimization. Experiment with this attribute only after you have a full understanding of templates.

- The `xulcontentgenerated` attribute can be set to `true`. It affects when template query and content generation occur. If a template does lazy building, then at any time, some of the possible content will be gen-

erated and some may not. If a DOM operation (like adding a child tag) were attempted on a tag with incomplete lazy content, confusion could result. Where would such a child tag be put when the final set of children is yet to be determined by the query? Mozilla handles this by forcing the template to build that tag's children completely before starting the DOM operation. The `xulcontentgenerated` attribute is a programmer-supplied hint that advises there is nothing to rebuild at this point in the XML tree. This speeds up DOM operations and saves unnecessary processing.

The `ref` attribute states the starting point for the template query. It holds the full URI of a fact subject. That subject should be the name of an RDF `<Seq>`, `<Alt>`, or `<Bag>` container.

`ref` and `containment` are also discussed in the following subtopic.

14.3.2.1 `ref` and `containment`: Container Tests The `ref` and `containment` attributes can be used to specify a starting point for a template query that doesn't use an official RDF container tag. This is useful for plain RDF facts that form a simple hierarchy but that don't use `<Seq>`, `<Bag>`, or `<Alt>`. Such pretend containers require a little explanation.

Consider a normal RDF container. An example is shown in Listing 14.9.

Listing 14.9 RDF container fragment equal to three facts.

```
<Description about="urn:eg:ownerA">
  <prop1>
    <Seq about="urn:eg:ContainerA">
      <li>
        <Description about="urn:eg:item1" prop2="blue"/>
      </li>
    </Seq>
  </prop1>
</Description>
```

The predicates in this example are deliberately kept simple. Listing 14.9 is equivalent to three facts:

```
<- urn:eg:ownerA,     prop1,  urn:eg:containerA ->
<- urn:eg:containerA, rdf:_1, urn:eg:item1 ->
<- urn:eg:item1,      prop2,  blue ->
```

The first fact has the RDF container as its subject. This is the fact that "owns" the container. The second fact states that `item1` is a member of the container. The third fact records some useful color information about `item1`. This is all normal RDF. These three facts result directly from the `<Seq>` syntax in Listing 14.9.

Suppose that a program was presented with these three facts rather than with the RDF markup. How could it tell if a container were present? In this simple example, detecting the predicate `rdf:_1` is enough to tell that

containerA is an RDF container. Inside Mozilla, a different test is used (involving the `rdf:instanceOf` predicate), but in this example, testing for `rdf:_1` will do.

Now suppose that a single change is made to these three facts. Suppose that the predicate `rdf:_1` is changed to `rdf:member` (or anything else). Then the three facts would be

```
<- urn:eg:ownerA,     prop1,  urn:eg:containerA ->
<- urn:eg:containerA, rdf:member, urn:eg:item1 ->
<- urn:eg:item1,      prop2,  blue ->
```

Listing 14.10 shows an RDF fragment that could produce these slightly different facts. This fragment is just a series of nested facts.

Listing 14.10 RDF non-container fragment equal to three facts.

```
<Description about="urn:eg:ownerA">
  <prop1>
    <Description about="urn:eg:ContainerA">
      <rdf:member>
        <Description about="urn:eg:item1" prop2="blue"/>
      </rdf:member>
    </Description>
  </prop1>
</Description>
```

Is there still a container in these three facts? After all, the three facts are very similar in both before and after cases. Purists would argue that the answer is *No*. Mozilla says that the answer is *Maybe*.

It is easy to argue that a container does exists. First, the organization of the three facts hasn't changed. Second, the choice of `rdf:member` as a predicate suggests that `item1` belongs to *something*. Third, any query built for a `<Seq>` tag might work just as well with the new predicate as it did with the old.

The point is this: When RDF markup is boiled down to simple facts, whether containers exist or not is just a matter of opinion. It is easy to create an RDF document with no RDF container tags, but still think of containers as being present. There is no reason why Mozilla should work with one arrangement but not with the other. Mozilla and templates work with both arrangements.

That brings us back to the `ref` attribute. It can be set to "`urn:eg:ContainerA`", and that resource will be used as a starting point for a template query, regardless of whether the original RDF looks like Listing 14.9 or 14.10. An RDF `<Seq>`, `<Bag>`, or `<Alt>` is not necessary.

There is one catch to this flexible use of `ref`. Mozilla must still determine whether the `ref` URI can act like a container. By default, there are three ways that the `ref` URI can pass this test:

1. The URI is a `<Seq>`, `<Bag>`, or `<Alt>` tag's `about` value.

2. A fact exists with `ref` as subject and `http://home.netscape.com/NC-rdf#child` as predicate.
3. A fact exists with `ref` as subject and `http://home.netscape.com/NC-rdf#Folder` as predicate.

These tests are the Mozilla equivalent of testing for `rdf:_1`.

If you don't want to use the `child` or `Folder` predicates in your facts, then you don't need to. You can use your own predicate. To do that, create the RDF content as you see fit, and in the XUL template code, add the `containment` attribute to the template's base tag.

The `containment` attribute can be set to a space-separated list of predicates. These predicates will be added to the list of container tests. These predicates will be tested for just like the items in the preceding list. For example, setting

```
containment="http://www.test.com/Test#member"
```

means that the RDF `<Description>` tag in this code will be considered a container by Mozilla, provided that `xmlns:Test="http://www.test.com/Test#"` is declared somewhere:

```
<Description about="urn:foo:bar">
  <Test:member resource="urn:foo:bar:item1"/>
<Description>
```

In summary, template queries work even if official RDF container tags aren't present, but in that case, you must tell the template system what predicates are used to implement the unofficial container.

14.3.2.2 Attributes Specific to `<tree>`

If the `<tree>` tag is used for a template, then some extra attributes apply:

```
flex="1" statedatasource flags="dont-build-content"
```

Trees do not have a default height. If a `<tree>` template does not have `flex="1"`, the template content often does not appear. Always use `flex="1"` on a `<tree>` template.

The `statedatasource` attribute is set to a named data source that is used to store the current state of the displayed tree. If the user has opened or closed a number of subtrees in the display, then information about which subtrees are open or closed will be persisted to the nominated data source. Currently, this is used only in the Classic Mail & News client, in the pane where the Mail and Newsgroup folders are listed.

If `statedatasource` is not set, then the data source named in the `datasources` attribute is used instead.

The "`dont-build-content`" value for the `flags` attribute is also tree-specific. It is described under "The Base Tag."

14.3.2.3 Sort Support for `<template>` Siblings The template system allows a column of data to be sorted. This feature works with XUL menus, listboxes, and trees.

The sorting process uses several attributes. These can be set either on the base tag of the template or on an individual `<listcol>` or `<treecol>` tag. These attributes are

```
resource resource2 sortActive sortDirection sortResource
     sortResource2
```

- `resource` holds a template variable. This attribute is set by the XUL document author on the column to be sorted. The data that are used for the sort key is specified by this attribute. The template variable it names represents a fact predicate/property associated with each row's solution to the template query. The data used as the sort key are the object/value of that predicate. In other words, this attribute specifies a property whose value per-row is to be sorted.
- `sort` is an alternate syntax for resource. It is also used to detect the sort column in a tree or listbox. Use `resource` and `sortActive` instead of this attribute.
- `resource2` is a secondary predicate for the sorting system. The sort performed on the resource values is not a stable sort. This means that after sorting, a secondary column of information might be very disordered. The `resource2` attribute causes a second column to be sorted if values in the primary column are equal. A third or subsequent column may still be disordered.
- `sortActive` can be set to `true` and indicates whether the data are currently sorted. Mozilla automatically sets this attribute on the specific column and on the tree or listbox. It can be set by the application programmer as well. It is also used to detect the column to be sorted, so it should always be set if `sort` is not set.
- `sortDirection` may be set to `ascending`, `descending`, or `natural`. This attribute may be set by the XUL document author or automatically by Mozilla after a sort. Mozilla will set it on both the specific column sorted and on the base tag.
- `sortResource` and `sortResource2` are the same as `resource` and `resource2`. Mozilla sets these attributes. The secondary sort criteria can be specified by the application programmer using JavaScript, but not directly using XUL.
- `sortSeparators` can be set to `true`. If that is done, then special processing occurs for bookmarks. A sort will not move an item across a bookmark separator if this attribute is set and the `rdf:bookmarks` data source is used.

14.3.3 <template>

The <template> tag holds all the details of a template that are not specified in the base tag. It holds a set of rules, each of which is a query-content pair. The <template> tag has no special attributes of its own. The only tag that the <template> tag can hold is the <rule> tag. It can have any number of <rule> tags as children.

If no <rule> tags appear in the template, but other content does, then the content is assumed to be the content of a single rule that uses the simple-rule syntax.

A <template> tag can contain simple and extended rules.

14.3.4 <rule>

The <rule> tag defines a single template query and the content that is generated to pretty-print the results of the query. A set of <rule> tags makes a query list, so the first <rule> in the list that is satisfied by a set of facts will pretty-print the template variables ground to those facts.

A rule can be expressed in either simple syntax or in extended syntax. If the first child tag of the <rule> tag is a <conditions> tag, then the rule is in extended syntax. In all other cases, simple syntax applies. A <rule> tag may be a singleton tag with no content.

14.3.4.1 The Standard Fact Arrangement All simple syntax queries and many extended syntax queries rely on the facts in an RDF file being arranged a particular way. This arrangement is used repeatedly. It is the case where the RDF data has a three-step, two-fact arrangement. These steps consist of a container term, its item terms, and their property-value terms. Listings 14.1 and 14.2 and the accompanying discussion gives an example of this arrangement.

This standard arrangement also has an equivalent in JavaScript:

```
var container = {item1:{p1:v1,p2:v2}, item2:{p1:v1,p3:v3} }
```

In this literal, container collects two items. item1 and item2 are objects, each with a set of properties pN. Each property has a value vN. There can be any number of items, each with any number of properties. The whole point of this structure is just to make a set of objects and their interesting properties easy to get at. In RDF, Listings 14.2, 14.5, 14.7, and 14.9 are all correct examples of this structure. An RDF equivalent to this JavaScript is shown in Listing 14.11.

Listing 14.11 Required RDF structure for an RDF rule using simple syntax.

```
<Seq about="container">
  <li resource="item1"/>
  <li resource="item2"/>
</Seq>
<Description about="item1" p1="v1" p2="v2"/>
<Description about="item2" p1="v1" p3="v3"/>
```

To make the RDF syntax neat, the <Seq> tag is usually wrapped up inside a <Description> tag, not shown in Listing 14.11. RDF has flexible syntax, and there are several other ways of expressing this same structure. Also, <Seq> can be replaced with <Bag> or <Alt>. A <Description> tag that acts like a container can be used instead.

All facts processed with simple syntax queries must have this standard arrangement.

14.3.4.2 Simple Syntax for <rule> The simple rule syntax has no special-purpose tags.

All the content of the <rule> tag is generated each time the rule's query finds a solution. The content of the <rule> tag can contain simple template variables embedded in any XML attribute value. The content is generated out as for the <action> tag, except for two minor differences: The simple syntax requires that the uri attribute be set to rdf:*, and the simple syntax cannot use the <textnode> tag.

When simple syntax is used, the <rule> tag has several special attributes:

```
type iscontainer isempty predicate="object" parsetype
```

All these attributes determine whether the rule's query will be successful. Part of the rule's query comes from the assumption that the RDF facts are in the standard fact arrangement. These attributes make up the rest of the query.

The type attribute comes from the official RDF XML namespace http://www.w3.org/1999/02/22-rdf-syntax-ns#. It is usually quoted as rdf:type, with that namespace included somewhere in the XUL document using xmlns. rdf:type can be set to a full predicate name, such as http://home.netscape.com/NC-rdf#version. This attribute is used to test for the existence of a predicate. rdf:type therefore tests to see if any objects in the container have a given property. If they don't, the rule fails for that query solution candidate.

The iscontainer attribute can be set to true or to false. The default is don't care., which cannot be set explicitly. It tests to see if a given fact subject is a container, using the container tests described earlier under "ref and containment: Container Tests." If the test fails, the rule fails for that query solution candidate.

The isempty attribute can be set to true or to false. The default is don't care, which cannot be set explicitly. It tests to see if a given URI has any kind of content. In the true case, it tests to see if container-like URIs have any contained items, or if item-like URIs have no properties. If the test fails, the rule fails for that query solution candidate.

The *predicate="object"* attribute stands for any pair of predicate and object names. Because predicate names are often long, they are usually shortened by adding an xmlns declaration at the top of the XUL document. If that is done, then a *predicate="object"* pair like

```
NC:version="0.1"
```

will test whether a fact exists with a version predicate in the NC namespace (perhaps `http://home.netscape.com/NC-rdf#`) and whether it has an object literal of "`0.1`"—in other words, whether an object in the container has a version property set to "`0.1`". The following reserved names cannot be used for the predicate part because they have other uses:

```
property instanceOf id parsetype
```

These three attributes, plus any number of *`predicate="object"`* tests, can be stated in the one `<rule>` tag. They are boolean ANDed together and make up all the query of the rule. If none of these attributes is present, then every fact in the container is a solution to the rule's query. If any of these attributes is present, then the rule fails for a given solution candidate if any of these attributes does *not* match.

The final attribute, `parsetype`, can be set to `Integer`. If this is done, all *`predicate="object"`* pairs in the `<rule>` tag will have their object part interpreted as an integer during the query. Any values that are not integers will cause the whole rule to be ignored. If this attribute is not used, all such parts will have their object part treated as a string. The query system has no support for integer mathematics like addition. It just performs comparisons on a string or integer basis.

The query of a simple rule can usually be expressed as an extended rule. Listing 14.12 is an example of simple syntax that uses several of the special attributes:

Listing 14.12 Simple template query showing all options.

```
<rule iscontainer="true"
      isempty="false"
      rdf:type="http://home.netscape.com/NC-rdf#version"
      NC:title="History"
>
```

The equivalent extended syntax rule is shown in Listing 14.13.

Listing 14.13 Simple template query showing all options.

```
<rule>
  <conditions>
    <content uri="?uri"/>
    <member container="?uri" child="?item"/>
    <triple subject="?item
            predicate="http://home.netscape.com/NC-rdf#version"
            object="?version"/>
    <triple subject="?item"
            predicate="http://home.netscape.com/NC-rdf#title"
            object="History"/>
  </conditions>
```

The extended syntax can do everything the simple syntax can do, except for the options `iscontainer="false"` and `isempty="true"` (the opposites of the preceding example). These two options are not possible in the extended syntax. The extended syntax can check for the existence of facts but not for the absence of facts. These simple tests can still be done with extended syntax by stating two rules instead of just one. The first rule tests for particular facts, but generates no content; the second rule catches the remaining cases, where that content is missing, and generates whatever content is required.

14.3.4.3 Extended Syntax for `<rule>`

When extended rule syntax is used, the `<rule>` tag has no special attributes. This syntax is the most flexible and powerful template syntax. "Common Query Patterns" later in this chapter provides recipes for most common uses. This topic covers the syntax options.

The extended rule syntax consists of a `<rule>` tag with two or three immediate children. The `<conditions>` tag must be the first child and contains the query of the rule. The `<bindings>` tag is the optional middle child and allows extra variables to be created. The `<action>` tag holds the content to generate each time the query finds a solution. These tags are described under their individual headings.

The extended rule syntax uses extended syntax variables, and these variables may be used in siblings of the `<template>` tag. A common use is to put them in the column definition tags of listboxes and trees.

If the query is recursive, then only the `<conditions>` part of the rule dictates the recursive behavior.

14.3.5 `<conditions>`

The `<conditions>` tag contains a template query described using the extended template syntax. This tag has no special attributes of its own. If it is present, it must be the first child tag of `<rule>`. The recursive nature of a recursive query is fully specified by the content of the `<conditions>` tag.

This tag can contain `<content>`, `<triple>` and `<member>` tags. Its first child must be a `<content>` tag, and there can be only one `<content>` tag per condition. The `<conditions>` tag must also contain at least one `<member>` or `<triple>` tag.

The `<conditions>` tag can also contain a `<treeitem>` tag. If the template is based on a `<tree>` tag, then `<treeitem>` may be used instead of `<content>`. In that case, it is written and acts exactly the same as a `<content>` tag. There is no special reason for this variation; it is just a leftover requirement from the early days of templates. `<treeitem>` is deprecated in favour of `<content>`, but for older versions `<treeitem>` must be used if the template is based on a `<tree>`.

When these tags are used, their order should be the same as the drilldown order used by the template query processor. That order is also the same as the hierarchical arrangement of the facts being examined.

14.3.5.1 <content> The <content> tag must be the first child of the <conditions> tag. It has only one special template attribute:

```
uri
```

This attribute is set to an extended template variable. That variable is then ground to the value supplied to the ref attribute in the template's base tag. That value is the starting point for the rule's query. If the query is recursive, the uri attribute of the parent query is used instead of ref. The <content> tag always has the same form, so by convention it is usually written exactly like this:

```
<content uri="?uri"/>
```

The uri attribute is also used on tags that are children of the <action> tag. The variable used in the <content> tag's uri attribute must *never* be used as the value of uri attributes appearing inside <action> content. If it is used in that way, no solutions to the template query will be found, and no content will be generated. The variable used in the <content> tag's uri attribute may appear elsewhere in the <action> tag's content.

If the template is based on a tree, then for versions less than 1.5, the <treeitem> tag is used instead, exactly like this:

```
<treeitem uri="?uri"/>
```

The <treeitem> tag can also be part of the generated content. In that case, it can also appear in the <action> tag, but any uri attribute must have a variable different from ?uri as its value.

14.3.5.2 <triple> The <triple> tag represents one hop (one RDF arc) in the query process. Each <triple> tag used increases the number of hops by one, and increases the number of facts required for a solution by one. The <triple> tag is used to link facts using variables and to select facts.

The triple tag has the following special attributes, which must always be present:

```
subject predicate object
```

- subject may be set to an extended template variable or a URI.
- predicate must be set to a URI that is a property/predicate name.
- object may be set to an extended template variable, a URI, or a literal value.

Variables may not be used in the predicate attribute.

It is possible to construct a set of <triple> tags that drills down into the RDF facts and then drills back up somewhere else. It is also possible to construct a set of <triple> tags that create a cycle. Neither of these things are handled correctly in Mozilla. A <triple> tag with zero variables is useless.

14.3.5.3 <member> The <member> tag is used to find the contained items of a container tag. It has the special attributes

```
container child
```

The container attribute is the URI of the container tag. The child attribute will be matched to the object of facts that have the container URI as their subject. In all uses of this tag, both attributes are present, and both are set to extended template variables, like so:

```
<member container="?uri" child="?item"/>
```

Because the containment relationship inside an RDF container is just a single fact (with rdf:_1 and so on as the predicate), an ordinary <triple> tag can also be used to find these contained items. The <member> tag is just a specialized version of the <triple> tag.

The advantage <member> has over the <triple> tag is that no predicate needs to be supplied. The <member> tag is free to apply a container test using the several predicate possibilities described earlier under "Ref and containment: Container Tests." To do the same job with the <triple> tag would require a separate query that hard-coded each URI that can be used as a container test URI.

Although the container attribute for a <member> tag often holds the URI starting point of the query, it can be any URI or extended syntax variable used in the query.

14.3.6 <bindings>

The <bindings> tag is the optional part of an extended syntax query. It has no special attributes of its own and can contain only <binding> tags. <bindings> and <binding> appear elsewhere in Mozilla, particularly in XBL. There is no connection between any other use and the template use. Explained in SQL terms, this part of an extended query provides support for outer joins and null values.

The <bindings> tag might better be called the <extra-groundings> tag, since there are no bindings in the XBL (or XPConnect or IDL or XPIDL) senses. A *binding variable* is just an additional variable that may be ground by a given query. The same meaning applies to the <binding> tag.

The <bindings> section of the rule is where extra variables can be set without further restricting the results of the <conditions> query. This is done very simply. In the <conditions> section, all variables must be ground if a solution to the query is to be found. Such a solution is passed to the bindings section. The query processor also tries to ground all the variables stated in the <bindings> section. If only some of the extra variables can be ground, then the processor just gives up and sets the rest to the empty string. This means that the <conditions> solution is never rejected by the <bindings> section. In effect, the <bindings> section is similar to an RDBMS outer

join—if something matches, it is reported; otherwise, you get the rest of the query's matched data only.

The variables in the <bindings> section must all be extended template variables. Variables from the <conditions> section may be reused here, but the reverse is not possible. For the <bindings> section to make any sense, at least one variable from the <conditions> section must be used.

14.3.6.1 <binding> The <binding> tag is identical to the <triple> tag in attributes and in purpose. It differs from <triple> only in the rules used to ground its variables, as noted under <bindings>.

14.3.7 <action> and Simple Syntax Content

The <action> tag is used in a <rule> using the extended query syntax. It provides the content that is generated for each query result. The content of simple syntax <rule> tags is generated the same way as <action> content is. That generation system is described here.

The content of an <action> tag consists of ordinary XUL content with embedded extended query variables. The template system substitutes a value for every variable when generating content. Therefore, variables are used in the <action> content, whereas they are defined and ground in the <conditions> content (they are automatically defined and ground for simple syntax queries). The template system also adds an id attribute to one tag each time content is generated.

The content of <action> cannot include a <script> tag.

The content of the <action> tag should be organized as shown in Listing 14.14.

Listing 14.14 Example <action> content hierarchy.

```
<action>                        // simple syntax: <rule>
  <box id="A">
    <box id="B">
      <box id="C" uri="?var"> // simple syntax: uri="rdf:*"
        <box id="D"/>
        <box id="E">
          <box id="F"/>
        </box>
      </box>
    </box>
  </box>
</action>
```

This example is used to explain what works and how it works. In the general case, tags may be nested to arbitrary depth within <action>, and any ordinary XUL tag may be used. The special case of <tree>-based templates is discussed separately.

The content of <action> is divided into the tag containing the uri attribute, that tag's ancestor tags, and that tag's descendant tags. In Listing 14.14, C contains uri; A and B are ancestors; and D, E, and F are children.

Ancestor tags A and B are normally generated as content just once, no matter how many solutions the query finds. If template variables (extended ones for <action> content, simple ones for <rule> content) are used in A or B, then they will have the value of the first solution found by the template. If tags A or B have sibling tags, those siblings may not be generated reliably. It is recommended that such sibling tags be avoided.

Tag C is the start of the repeated content because it contains the uri attribute. Both tag C and all its content will be generated as a unit zero or more times. There are two triggers required before the content will be repeated once. First, a solution must be found to the query. Second, the value of the uri attribute on tag C must be different in that new solution. The easiest way to ensure this is to set that value to a variable quoted in the object of some tuple. One example is the child part of a <member> tag: Use ?item in this case:

```
<member container="?seq" child="?item"/>
```

In the most general case, the uri attribute should therefore be set to a subject or object that is different for every solution found by the query. This unique value is put into the id attribute of each generated copy of tag C during generation. Tag C should not have sibling tags; those sibling tags may not be generated reliably. It is recommended that such sibling tags be avoided.

If lazy building is required, then tag C must be one of these XUL tags:

```
<menu> <menulist> <menubutton> <toolbarbutton> <button> <treeitem>
```

Tags D, E, and F are ordinary content tags that are generated once per query solution. They may have siblings or extensive content of their own. If these tags contain template variables, those variables will be substituted in for each query solution found.

14.3.7.1 Content for <tree>-Based Templates

Content for <tree>-based templates can break the rule that ancestor tags of the uri tag are generated only once.

If a template is based on a <tree> tag, then each <treeitem> in the final tree might have a <treechildren> tag as its second child tag. That tag could contain any number of subtree tags for that item. Fortunately, the content held in <action> need only represent a single tree row. The template content builder that processes <tree> templates is intelligent enough to duplicate this content for subtrees. In other words, it is smart enough to detect the need for recursive queries. The <action> content must be constructed carefully if this arrangement is to work. Listing 14.15 shows the correct construction.

Listing 14.15 Example <action> content hierarchy for a <tree> template.

```
<action>                    //simple syntax: use <rule>
  <treechildren>
```

```
    <treeitem uri="?var">  //simple syntax: use uri="rdf:*"
      <treerow>
        ... any normal treerow content ...
      </treerow>
    </treeitem>
  </treechildren>
</action>
```

All the tags in this arrangement can have attributes added. An `id` should not be added to the `<treeitem>` tag. The `?var` variable usually matches a variable for the object of a `<member>` or `<triple>` tag.

It is possible to vary this syntax a little, and even to nest templates inside other templates—for example, so that a `<treeitem>` comes from one RDF source but its `<treechildren>` sibling tag comes from another. Such nested syntax needs to be explored carefully because it is not widely used or tested.

Tree-based templates use recursive queries. For each recursion of the query deeper into the tree of facts, the `<action>` content above the `uri` tag is duplicated. This means one further copy of the `<treechildren>` tag stated in Listing 14.15. This new copy is then the starting point for all the solutions found by that new query.

14.3.7.2 `<textnode>` Simple and extended template variables can appear in XML attributes' values only. Some XUL tags generate content from text between their start and end tags, like `<Description>`.

The `<textnode>` tag provides a way to put variables in plain, non-attribute content. Suppose that the variable `?var` contains the string `"red"` at some point. The line

```
<tag><textnode value="big ?var truck"/></tag>
```

is the same as the line

```
<tag>big red truck</tag>
```

`<textnode>` can be placed anywhere inside the `<action>` tag. `<textnode>` cannot be used in templates that have the simple query syntax.

`<textnode>` concludes the discussion of XUL template tags.

14.4 COMMON QUERY PATTERNS

Here are many of the common queries used in templates.

14.4.1 Containers

The most obvious template queries exploit the standard fact arrangement, and that means using an RDF container. The simple query syntax supports this arrangement.

14.4.2 Single-Fact Queries

Single-fact queries (see the prior discussion) can be specified using the extended query syntax only; they cannot be specified using the simple query syntax. A single-fact query may return zero or more solutions. The only unknown that can be queried for is the object of the single fact. The subject and predicate must be known in advance and hard-coded in the query.

Such a query can be implemented with either a `<member>` or a `<triple>` tag. The `<member>` variation is

```
<rule>
  <conditions>
    <content uri="?uri"/>
    <member container="?uri" child="?data"/>
  </conditions>
  <action>
    <description uri="?data" value="?uri ?data"/>
  </action>
</rule>
```

In this case, if the predicate of the single fact is not a containment predicate (e.g., `rdf:_1`, or a special Mozilla value like `child`), then the containment attribute on the template must specify that predicate. Several predicate alternatives can be listed in that attribute.

The `<triple>` variation is

```
<rule>
  <conditions>
    <content uri="?uri"/>
    <triple subject="?uri"
      predicate="predURI"
      object="?data"/>
  </conditions>
  <action>
    <description uri="?data" value="?uri ?data"/>
  </action>
</rule>
```

"`predURI`" is the predicate to use, expressed as a literal. If predicate alternatives are required, use more than one rule of this kind.

14.4.3 Property Set

A *property set* is a query whose purpose is to retrieve a number of information items attached to a given fact subject. A typical example is treating a fact subject as a JavaScript object and retrieving the property values of that object. A second example is treating a fact subject as the name of a record and retrieving all the data items in that record. These semantic approaches are encouraged by the RDF standard's use of the terms *property* and *value*.

If the subject with associated properties is a member of an RDF container, or a container-like tag, then the simple query syntax is designed with exactly this use in mind:

```
<rule>
  <box uri="rdf:*">
    <label value="rdf:http://www.test.com/Test#Prop1"/>
    <label value="rdf:http://www.test.com/Test#Prop2"/>
    <label value="rdf:http://www.test.com/Test#Prop3"/>
    <label value="rdf:http://www.test.com/Test#Prop4"/>
  </box>
</rule>
```

If the subject with the associated properties is not a member of a container, then the extended query syntax must be used. In that case, the starting point is the same as for the Single Fact Query pattern, using the `<triple>` version. At least one property must be known to exist.

```
<rule>
  <conditions>
    <content uri="?uri"/>
    <triple subject="?uri" predicate="p1" object="?v1"/>
    <triple subject="?uri" predicate="p2" object="?v2"/>
    <triple subject="?uri" predicate="p3" object="?v3"/>
    <triple subject="?uri" predicate="p4" object="?v4"/>
  </conditions>
  <action>
    <description uri="?data" value="?v1 ?v2 ?v3 ?v4"/>
  </action>
</rule>
```

Here, the `p` again stands for full predicate URIs, like `http://www.test.com/Test#Prop1`. There is one `<triple>` for each property in the needed set. If some of the properties may not exist (or could be null), then the matching `<triple>` tags for those properties can be changed to `<binding>` tags and moved to the `<bindings>` section.

14.4.4 Singleton Solution

A single solution is any query that returns just one solution. This usually occurs when the application programmer's knowledge of the facts being queried ensures that only one solution ever exists. That situation requires no special syntax; it is just a consequence of design.

There is, however, another possibility. It relies on the `uri` attribute used inside an `<action>` tag. Because `<action>` is required, this possibility only applies to the extended syntax.

It is possible for a query to find multiple solutions but to generate content for just one. This can happen only when the query is a multifact query. An example illustrates this case. Suppose that the following RDF content exists:

```
<Description about="urn:example:root">
  <link1>
    <Description about="urn:example:child">
      <link2 resource="urn:example:X"/>
      <link2 resource="urn:example:Y"/>
    </Description>
  </link1>
</Description>
```

All the resources of these facts can be recovered with this query:

```
<rule>
  <conditions>
    <content uri="?uri"/>
    <triple subject="?uri" predicate="p1" object="?child"/>
    <triple subject="?child" predicate="p2" object="?res"/>
  </conditions>
  <action>
    <description uri="?res" value="?uri ?child ?res"/>
  </action>
</rule>
```

In this query, `p1` and `p2` should be replaced with suitable URIs for `link1` and `link2`.

Since the variable `?res` changes for each of the two solutions that will be discovered (one with X and one with Y), two copies of the `<action>` content are generated. If, however, the `<description>` tag is changed to the following line, only one copy is generated:

```
<description uri="?child" value="?uri ?child ?res"/>
```

One copy is generated because the `?child` variable can be ground to only one value. The one copy that is generated usually matches the first solution in the RDF document, but that order is not guaranteed.

14.4.5 Tree Queries

All template queries are equivalent to navigating a tree because the query system uses a depth-first solution strategy. Template queries for trees appear complex because (a) they are often recursive, and (b) extensive syntax is required. Disguised by this syntax is the fact that tree-based templates require very simple queries only. Listing 14.16 illustrates this point.

Listing 14.16 Simple extended syntax tree query.

```
<tree flex="1" datasources="test.rdf"
      ref="http://www.example.com/test.rdf">
  <treecols>
    <treecol id="colA" primary="true"/>
  </treecols>
  <template>
    <rule>
```

```
      <conditions>
        <content uri="?uri"/>
        <member container="?uri" child="?item"/>
      </conditions>
      <action>
        <treechildren>
          <treeitem uri="?item">
            <treerow>
              <treecell label="?item"/>
            </treerow>
          </treeitem>
        </treechildren>
      </action>
    </rule>
  </template>
</tree>
```

Listing 14.16 is a straightforward and minimal use of the tree-template combination. Dark text is template markup; light text is tree markup. Although there is a lot of syntax overall, the template specification is very simple. The <rule> section is no more than a single-fact query. The results of that query are used in exactly one place. The code seems complex because the markup overheads of the XUL template and XUL tree syntaxes are interleaved. If the <tree> tag is replaced with a <box> tag and the generated content reduced to a <description> tag, then this template is no more than Listing 14.17.

Listing 14.17 Simple extended syntax tree query.

```
<box datasources="test.rdf"
     ref="http://www.example.com/test.rdf">
  <template>
    <rule>
      <conditions>
        <content uri="?uri"/>
        <member container="?uri" child="?item"/>
      </conditions>
      <action>
        <description uri="?item" value="?item"/>
      </action>
    </rule>
  </template>
</box>
```

Clearly this is a very simple template.

This syntax can be further reduced to the simple syntax if the RDF content follows the standard fact arrangement. If the query is recursive, that change is a little more complex than it first appears. The standard fact arrangement requires two facts if a simple syntax query is to extract a single RDF property value. Those two facts have as subject terms the container iden-

tifier and the container member. Both facts must be repeated down the tree of possible solutions that the query searches, since each query step consumes two facts. That means a tree explored using the simple syntax must be twice as big as a tree explored more economically with the extended syntax. In Listing 14.17, only one fact is required per step down the tree.

14.4.6 Unions

There are several ways to generate a set of query solutions that is the union of two or more queries:

- The `datasources` attribute can refer to multiple RDF files or data sources so that the facts examined by a single query are a union of several fact sets.
- The `containment` attribute can add several different container predicates to the one query.
- Multiple `<rule>` tags can be used to ensure that more than one type of solution is found from a given set of facts.
- A set of `<binding>` tags can be used to explore several subtrees of a given query at the same time.

14.4.7 Illegal Queries

This query cannot be handled by the template system:

```
<rule>
  <conditions>
    <content uri="?uri"/>
    <triple subject="?uri" predicate="a" object="?v1"/>
    <triple subject="?v1"  predicate="b" object="?v2"/>
    <triple subject="?v2"  predicate="c" object="?v3"/>
    <triple subject="?v4"  predicate="d" object="?v3"/>
  </conditions>
</rule>
```

In this query, the last `<triple>` works backward from the `?v3` variable, rather than using it as a subject. This effectively decreases the depth that the query penetrates into the fact store by one. That is not implemented; all `<conditions>` content must expand the query in the forward direction only.

14.5 TEMPLATE LIFECYCLE

Here is a summary of the processing steps that templates go through.

The first part of the process is the initial generation of XUL content.

1. Mozilla loads the XUL document containing the template. XUL content surrounding the template is layed out as normal, leaving a spot for the template-generated content.

2. All tags, including template tags, are formed into a DOM 1 content tree.
3. When the template part of the content is detected, the browser starts loading facts into memory via the nominated data sources. This is done asynchronously, like images that load inside an HTML Web page.
4. The template tags are examined for rules, and the rules are compiled into an internal, private representation.
5. After the rules are known and the data source is fully loaded, the template query system starts searching for solutions. If the application programmer has added observers, those observers are advised of progress during the search.
6. If lazy building is in effect, the search process might mark parts of the search "don't do this now; maybe look at it later."
7. As each solution is found, content is generated for it from the template rules. This content is added to the XUL document. This is done asynchronously, like images loading inside an HTML Web page.
8. When all nonlazy solutions have been found, initial content generation is complete.

The second part of the process involves interactive response from the template after the initial load is complete.

1. The generated XUL content can react to events and other interactions exactly like ordinary static XUL content. It can exploit event handlers and respond to changes in layout and general-purpose scripting.
2. If the template includes lazy building, then a user clicking on a twisty can cause query processing to explore the RDF graph further. The preceding steps 5-8 are repeated in this case.
3. If the user performs drag-and-drop actions or clicks on sort direction icons, then a `<listbox>`- or `<tree>`-based template must respond to those actions, possibly by updating the fact store. Drag and drop requires extra scripting by the application programmer.
4. If the template is told to rebuild, then the preceding steps 2-8 are repeated, except that the existing generated content is replaced if necessary, rather than created from nothing.

If the template is read-only, so that no fact changes are made, then closing the Mozilla window or removing all the template DOM subtree has no effect on the source of the RDF facts. Such templates cannot damage RDF documents.

If the template includes scripting logic that writes changes to the underlying set of RDF facts, then those changes will be permanent only if (a) the facts come from an RDF file or the Bookmarks file and (b) that file is explicitly written out by a piece of script using the `nsIRDFRemoteDataSource` interface.

14.6 SCRIPTING

Here we will look at how to control and enhance templates from a script and further explore some of the advanced tree concepts discussed in Chapter 13, Listboxes and Trees. Chapter 16, XPCOM Objects, contains extensive discussion of the RDF and data source objects that complement the template system.

14.6.1 Guidelines for Dynamic Templates

Templates expressed in XUL alone are relatively easy because they don't ever change, and they generate content once only. Templates expressed in a combination of XUL, JavaScript, and AOM and XPCOM objects can be very challenging because full functionality is the exception rather than the expected. Here is some collected wisdom:

- The `dont-build-content` flag, and the other performance flags have no effect when a template as a whole is rebuilt. They affect only the progress of any recursive part of a template query, and the way multiple data sources are merged.
- The `ref` template attribute or property can be changed any time, and the generated content will be automatically rebuilt.
- Changes to any part of the template's specification other than `ref` require a manual call to `rebuild()`.
- The `nsIRDFCompositeDataSource` that holds all the data sources for a given template shouldn't be used to assert new facts (either by `Assert()` or by other means). Always extract and work on one of the data sources that contributes to the composite.
- For applications, the most reliable data sources for asserting facts are the `in-memory-datasource` (an initially empty data source that usually replaces a value of `rdf:null` for the `datasources` attribute) and the `xml-datasource` (used for externally stored RDF files, and for the URLs stated in the `datasources` attribute). The `xml-datasource` will only assert facts for URLs with a `file:` scheme. It will not assert facts for `chrome:` or `resource:` schemes, or for any other scheme.
- If you are making extensive fact changes to a data source, remove it from the template first, make the changes, and then add it back. That saves many complex updates to the appearance of generated content. See also the `beginBatchUpdate()` methods for trees in Table 13.3.
- The `Flush()` method is supported only for data sources based on `file:` URLs.
- The `Refresh()` method is supported for `http:` and `file:` URLs. When using this method to retrieve a file from a Web server, test equipment must be set up correctly. All HTTP caching effects between the server and Mozilla must be removed or else results may be confusing.

14.6.2 AOM and XPCOM Template Objects

The template system adds JavaScript properties to the objects representing template tags. Properties are added to both the template specification tags and the generated template content tags. These properties are in addition to the properties that exist for the base tag of the template. Table 14.1 lists these properties.

Table 14.1 JavaScript properties added by the template system

Property name	Base tag?	<template> tag?	Rules tags	Generated tags
database	Useful	Null	Null	Null
builder	Sometimes Useful	Null	Null	Null
resource	Useful	Useful—holds tag id	Null	Useful—holds uri value or null
datasources	Useful	Empty string	Empty string	Empty string
ref	Useful	Empty string	Empty string	Empty string

In this table, "Useful" means that the property gains a useful value; "Null" means that the property is set to null; "Empty string" means that the property contains a zero-length string.

- The `database` property is an object that contains references to the data sources used in the template. It implements the `nsICompositeDataSource` XPCOM interface. It exists even if the sole stated data source is `rdf:null`. For chrome-based templates, it always contains the `rdf:local-store` data source.
- The `builder` property is an object used to control the query and content generation process of the template. It can be seen as a highly specialized (and very limited) layout or rendering tool. It implements the `nsIXULTemplateBuilder` interface. It is useful for `<listbox>` and `<tree>`-based templates only.
- The `resource` property is an object that contains a compiled RDF URI. That URI can be a subject, predicate, or object URI. If the tag is `<template>`, it holds the template `id` as a non-URI resource. If the template is a generated tag that has a `uri` attribute, then it holds the value of the specified extended template variable. This object implements the `nsIRDFResource` interface.
- The `datasources` property holds the value of the `datasources` XUL attribute.
- The `ref` property holds the value of the `ref` XUL attribute.

The `database` object contains the `AddDataSource()`, `RemoveDataSource()`, and `GetDataSources()` methods, which are used to manage the data sources that feed facts into the template. The `builder` object contains primarily the `rebuild()` method, which is used to completely re-create and refresh the displayed contents of a template. The `resource` object and `datasources` and `ref` attributes are all available merely for convenience. None of these objects may be replaced by the application programmer.

In addition to these AOM objects are the objects that support builder observers, views, and delegates. They are discussed next.

14.6.3 Builders

Template builders are builders—a piece of code inside the Mozilla Platform that generates the content of a tree. Builders cannot be implemented by an application programmer, but they can be modified.

The simplest use of a builder is to re-create and redisplay a templated `<tree>`. Only one line of code is required:

```
treeElement.builder.rebuild()
```

There are no options. This technique works only on templates whose base tag is a `<tree>` or `<listbox>`. `rebuild()` does not work on other templated tags.

When the tree is rebuilt, the open and closed states of any subtrees will be preserved if lazy building is at work. To stop this from happening, use the `statedatasource` attribute on the `<tree>` tag, and then take that data source away just before rebuilding. To take it away, remove the data source from the databases' property, then remove the tree attribute using a DOM operation, and then call `rebuild()`.

Recall there are two builders used in Mozilla; the XUL content builder and the XUL template builder. The latter deals with templates. This template builder is based on this component and interface:

```
@mozilla.org/xul/xul-template-builder;1 nsIXULTemplateBuilder
```

This interface contains the `rebuild()` method. This builder cannot be modified or replaced by the application programmer.

Inside Mozilla, this template builder has a specialization (a subclass) that is specific to templated trees: the XUL tree builder.

```
@mozilla.org/xul/xul-tree-builder;1 nsIXULTreeBuilder
```

This builder cannot be replaced by the application programmer either. It can, however, be enhanced. It can accept the registration of an object with the `nsITreeBuilderObserver` interface. Although this interface acts somewhat like an observer, it is also very similar to a delegate object. Which of these two design patterns is most accurate is a question of opinion as much as it is of fact.

This `nsITreeBuilderObserver` interface is a subset of the `nsITreeView` interface that is used for custom views. It is used to enhance user interactions with the tree, like column clicking and drag-and-drop gestures. In the Mozilla application, two examples of such an observer exist: one in the Bookmark Manager and one in the email folders pane of Classic Mail & News. Both implement drag and drop for their respective trees, and their JavaScript code is a suitable example to study.

This interface is used for drag and drop because of the way that `<tree>` tags hold their content separate to the rest of the XUL document (see Chapter 13, Listboxes and Trees). The standard drag-and-drop techniques described in Chapter 6, Events, can't be used for a `<tree>`. In fact, there is no (simple) way to do drag and drop for a `<tree>` that is not a template.

If an object with the `nsITreeBuilderObserver` interface is used, it should be lodged as an observer with the tree's DOM object using the `addObserver()` method of `nsITreeView`.

All the Mozilla tree builders also support the `nsIRDFObserver` interface, which means the whole tree can in turn act like an observer. This interface can be added to a data source object (interface `nsIRDFDataSource`), and it will then be advised each time new facts relevant to the template's queries appear.

14.6.4 Views

If the template is based on a tree, then properties relating to views are also available. Chapter 13, Listboxes and Trees, explains that a view is an automatically created object that is used by a tree builder to perform the actual generation of tree content.

A custom view object completely replaces the content that would be displayed from a template-based RDF file. If the data source for the `<tree>` template is `rdf:null`, then a custom view can do the whole job of populating the tree. The simplest way to set this up is to use a `<tree>` tag with a normal `<treecols>` section, plus one of these three options for the remaining content:

```
<children/>
<template><treechildren/></template>
<template> ... normal set of template rules ... </template>
```

Such a tree will display its ordinary content first, then when the view is changed and tree rebuilt, the view will take over and determine the content from that point onward. If the attribute `flags="dont-build-content"` is added, then the ordinary content will not appear at all.

14.6.5 Delegates

A delegate is a very general term in object-oriented design, and many design patterns and software objects have some delegate-like features. In Mozilla, delegates are objects that are used to tie RDF fact URIs to an external system,

so that each URI has other baggage associated with it. That other baggage might be vital data of some kind.

In Mozilla, delegates are used to tie RDF facts that are about email information to related information stored on a remote mail server, like an SMTP, IMAP, or LDAP server.

Time and space do not permit a discussion of delegates in this chapter. A starting point is to look closely at the `nsIRDFResource` and `nsIRDFDelegateFactory` interfaces, as well as at the XPCOM components whose contract ID's begin with

```
@mozilla.org/rdf/delegate-factory;1?key=
```

14.6.6 Common Scripting Tasks

Here is a brief list of strategies for common template scripting tasks. There is extensive examination of scriptable RDF objects in Chapter 16, XPCOM Objects. In particular, tricks that use the `rdf:null` data source and other internal data sources are discussed there.

To change a template's data source, use `databases.getDataSources()`, iterate through the provided list to find the right one, use `databases.removeDataSource()` with that data source as the argument, and then rebuild. Or remove that data source from the `<tree>` tag and rebuild.

To change the root URI for a template query, use `setAttribute("ref",newURI)` on the base tag. The template will rebuild automatically. Setting the `ref` property has the same effect.

To change rules, use DOM operations or `innerHTML` to change the template tags directly in the XUL, and then rebuild by calling `rebuild()`. A better solution is to create all possible rules and disable the one you don't need. Do this by placing a catch-all rule before the rules that should be disabled. The catch-all rule finds solutions for all cases it receives, so the remaining (disabled) rules are never considered.

To change facts in a data source, extract the specific data source using `databases.GetDataSources()`. Use the `Assert()` or `Change()` methods of the `nsIRDFDataSource` interface. The template will be rebuilt automatically in some cases; to be sure, call `rebuild()`.

To change the results of a query, step back a little. You can't change the solutions that result from the template queries because they are generated by the query. You need to change the query or the original RDF data so that the query solutions become different. To change the facts in the fact store, use the `@mozilla.org/rdf/rdf-service;1` component and other RDF components to construct facts and pieces of facts, and then use the data source object's `Assert()` method to make those facts true. Then rebuild the template.

To change generated content, use normal DOM operations. (The changes will be lost if a rebuild occurs.) To make such changes permanent, change the `<content>` part of a rule rather than the generated content, and `rebuild()`.

14.7 STYLE OPTIONS

The template system has no related style enhancements, but some new tricks with the existing styles are possible.

Styles can be attached to content in a template, and the style and class attributes can benefit from the use of template variables. This allows some style information to be supplied by the RDF data that drives the template. It also creates an architectural problem because it allows presentation information and raw data to be mixed in one set of RDF facts. If this needs to be done extensively, it is better to use two RDF data sources and keep presentation facts in one and raw data facts in the other.

Styles can also be applied by using `<rule>` tags as data-driven style selectors. Two rules can generate near-identical content, but differ slightly in their queries. The near-identical content can just have different style information.

Finally, styles can be applied to the generated content of a template, using JavaScript DOM operations. If the template is rebuilt, these styles may disappear.

14.8 HANDS ON: NOTETAKER DATA MADE LIVE

This "Hands On" session is about using templates to get real data into an application.

In this session, we'll replace some of our NoteTaker XUL code with templates and then test it out with real Web note data.

To begin with, we have the RDF data model of Chapter 12, Overlays and Chrome. Later on we'll put this file in the user's profile, where it belongs. For now, we'll just store it in the chrome with the rest of the NoteTaker tool. It will be in:

```
chrome://notetaker/contents/notetaker.rdf
```

Templates are needed three times in the NoteTaker tool:

1. The form fields on the NoteTaker toolbar should be loaded with data from the current note. We'll see that this takes two templates.
2. The listbox of keywords in the Edit dialog box needs to be loaded with the existing keywords for the current note.
3. The tree of related keywords in the Edit dialog box needs to be loaded with all related keywords that match the keywords in the current note.

The last use requires a special technique; the tree is (very roughly) in a master-detail relationship with the listbox and needs to be coordinated against the contents of the listbox.

Data also appear in the HTML-based note displayed on the current Web page. HTML does not support templates, so we must extract the needed data

from RDF using a nontemplate system. That is done in Chapter 16, XPCOM Objects.

14.8.1 Creating Test Data

Before building templates, we need some test data. To that end we create a NoteTaker RDF document containing two notes and their related information. The first note is for a Web page named `test1.html`, which has four keywords. The second note is for a Web page named `test2.html`, which has two keywords. Listing 14.18 shows the full data for these two notes, plus an extra keyword relationship.

Listing 14.18 Test data for NoteTaker XUL templates.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmnls:NT="http://www.mozilla.org/notetaker-rdf#">
  <Description about="urn:notetaker:root">
    <NT:notes>
      <Seq about="urn:notetaker:notes">
  <li resource="http://saturn/test1.html"/>
  <li resource="http://saturn/test2.html"/>
      </Seq>
    </NT:notes>
    <NT:keywords>
      <Seq about="urn:notetaker:keywords">
        <li resource="urn:notetaker:keyword:checkpointed"/>
        <li resource="urn:notetaker:keyword:reviewed"/>
        <li resource="urn:notetaker:keyword:fun"/>
        <li resource="urn:notetaker:keyword:visual"/>
        <li resource="urn:notetaker:keyword:cool"/>
        <li resource="urn:notetaker:keyword:test"/>
        <li resource="urn:notetaker:keyword:breakdown"/>
        <li resource="urn:notetaker:keyword:first draft"/>
        <li resource="urn:notetaker:keyword:final"/>
        <li resource="urn:notetaker:keyword:guru"/>
        <li resource="urn:notetaker:keyword:rubbish"/>
      </Seq>
    </NT:keywords>
  </Description>

<!-- details for each note -->
  <Description about="http://saturn/test1.html">
    <NT:summary>My Summary</NT:summary>
    <NT:details>My Details</NT:details>
    <NT:top>100</NT:top>
    <NT:left>90</NT:left>
    <NT:width>80</NT:width>
    <NT:height>70</NT:height>
    <NT:keyword resource="urn:notetaker:keyword:test"/>
    <NT:keyword resource="urn:notetaker:keyword:cool"/>
  </Description>
```

```
  <Description about="http://saturn/test2.html">
    <NT:summary>Good place to list</NT:summary>
    <NT:details>Last time I had a website here, my page also appeared on
          Yahoo
    </NT:details>
    <NT:top>100</NT:top>
    <NT:left>300</NT:left>
    <NT:width>100</NT:width>
    <NT:height>200<NT:height>
    <NT:keyword resource="urn:notetaker:keyword:checkpointed"/>
    <NT:keyword resource="urn:notetaker:keyword:reviewed"/>
    <NT:keyword resource="urn:notetaker:keyword:fun"/>
    <NT:keyword resource="urn:notetaker:keyword:visual"/>
  </Description>

<!-- values for each keyword -->
  <Description about="urn:notetaker:keyword:checkpointed"
          NT:label="checkpointed"/>
  <Description about="urn:notetaker:keyword:reviewed" NT:label="reviewed"/
          >
  <Description about="urn:notetaker:keyword:fun" NT:label="fun"/>
  <Description about="urn:notetaker:keyword:visual" NT:label="visual"/>
  <Description about="urn:notetaker:keyword:breakdown"
          NT:label="breakdown"/>
  <Description about="urn:notetaker:keyword:first draft" NT:label="first
          draft"/>
  <Description about="urn:notetaker:keyword:final" NT:label="final"/>
  <Description about="urn:notetaker:keyword:guru" NT:label="guru"/>
  <Description about="urn:notetaker:keyword:rubbish" NT:label="rubbish"/>
  <Description about="urn:notetaker:keyword:test" NT:label="test"/>
  <Description about="urn:notetaker:keyword:cool" NT:label="cool"/>

<!--sufficient related keyword pairings -->
  <Description about="urn:notetaker:keyword:checkpointed">
    <NT:related resource="urn:notetaker:keyword:breakdown"/>
    <NT:related resource="urn:notetaker:keyword:first draft"/>
    <NT:related resource="urn:notetaker:keyword:final"/>
  </Description>
  <Description about="urn:notetaker:keyword:reviewed">
    <NT:related resource="urn:notetaker:keyword:guru"/>
    <NT:related resource="urn:notetaker:keyword:rubbish"/>
  </Description>
  <Description about="urn:notetaker:keyword:fun">
    <NT:related resource="urn:notetaker:keyword:cool"/>
  </Description>

<!-- single example of a cycle -->
  <Description about="urn:notetaker:keyword:cool">
    <NT:related resource="urn:notetaker:keyword:test"/>
  </Description>
  </Description>
</RDF>
```

The combination of RDF documents and XUL templates requires five times as much care as <tree>. If just one little thing is incorrect, the template will produce nothing, and there will be no clues why that happened. We start by carefully testing our data using the advice in "Debug Corner." If you haven't got a text editor with automatic XML syntax checking, then proceed this way: Directly load the RDF document into the Classic Browser to catch all XML syntax errors. Then change the file extension to .xml and load it again, to catch tag-ordering problems. Finally, check it by eye against the format that was decided in Chapter 11, RDF. When all that looks correct, we have a fair, but not perfect, level of confidence that the test data are right.

14.8.2 Simple Templates for Form Elements

The first templates we'll code are the ones in the NoteTaker toolbar. That toolbar requires one value for each of the two textboxes, and a set of values for the dropdown menu. We must use a template that produces one solution for the textboxes and another template that produces a set of zero or more solutions for the dropdown menu.

The dropdown menu is easiest. Looking at the structure of the notetaker.rdf file, we see that we need the label property of the individual keywords. The keywords are collected in the urn:notetaker:keywords RDF container, which is a <Seq> tag. This layout matched the standard fact arrangement required for the simple (template) query syntax, so the template should be easy to create.

We test the template on a simple document first, rather than add it to the toolbar right away. That way we avoid complexities with the menu's popup content. Because templates and RDF give us little feedback, we must proceed step by step. Listing 14.19 is the simple, template-free test document that we'll use.

Listing 14.19 Plain XUL document suitable as a basis for testing templates.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <vbox>
    <description value="Static content"/>
    <hbox>
      <description value="Repeated content"/>
    </hbox>
  </vbox>
</window>
```

The details of the needed template follow:

- Use this RDF file: datasources="notetaker.rdf"
- Use this query starting point: ref="urn:notetaker:keywords"

- This simple variable is always required on the repeated content for the simple syntax: `uri="rdf:*"`
- This property will be used as a simple syntax query variable: `rdf:http://www.mozilla.org/notetaker-rdf#label`

Merging that information into Listing 14.18 yields Listing 14.20.

Listing 14.20 Plain XUL document templated for the menu query.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <vbox
    datasources="notetaker.rdf"
    ref="urn:notetaker:keywords">
    <description value="Static content"/>
    <template>
      <hbox uri="rdf:*">
        <description value="Repeated content"/>
        <description
        value="rdf:http://www.mozilla.org/notetaker-rdf#label"/>
      </hbox>
    </template>
  </vbox>
</window>
```

The `"Static content"` content is outside the `<template>` tag, and so we should always see it. The `"Repeated content"` content will appear once for each solution found by the query. The third description tag displays the values of the sole variable ground by the query. This simple query page results in Figure 14.6.

Now we have the query working. In the preparation of this "Hands On" session, numerous subtle syntax errors had to be ironed out to reach this point; there is nothing wrong with the template system, except possibly a shortage of debugging tools. We can now modify the template for the toolbar. Listing 14.21 shows the dropdown menu both before and after the template is installed.

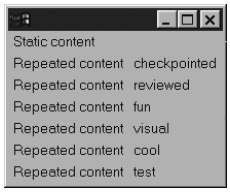


Fig. 14.6 Simple template-generated output using a test page.

Listing 14.21 Templated popup menu for the NoteTaker toolbar.

```
<?xml version="1.0"?>

<menulist editable="true">
  <menupopup>
  <!-- static menu items removed -->
  </menupopup>
</menulist>

<menulist id="notetaker-toolbar.keywords" editable="true">
  <menupopup datasources="notetaker.rdf" ref="urn:notetaker:keywords">
    <template>
      <menuitem uri="rdf:*"
        label="rdf:http://www.mozilla.org/notetaker-rdf#label"/>
    </template>
  </menupopup>
</menulist>
```

The results of these changes are shown in Figure 14.7.

If there are no keywords at all, the menu will contain no items and will lay out poorly in the toolbar. To fix this, we could add a dummy `<menuitem>` tag above the `<template>` tag, or just make sure that this initial `notetaker.rdf` contains at least one keyword. We'll do the latter.

The other template required in the toolbar must retrieve a single solution, since there's only one summary `<textbox>`, ever. To see what that template's query might be, we examine the structure of the `notetaker.rdf` file. The note's URL is a member of a named RDF container (`urn:notetaker:notes`), and it has a property summary. So the standard fact arrangement, required for a simple template query, is present. Perhaps we can use that simple syntax.

In fact, we can't use the simple syntax, because we want to be choosy about which members of the sequence are retrieved. We only want one member, which is the note for the currently displayed URL. Therefore we must resort to the extended syntax. In the extended syntax, we don't have to start

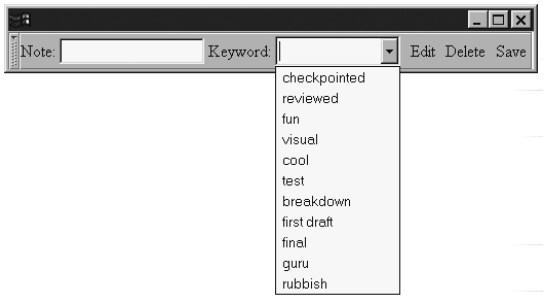


Fig. 14.7 Template-generated menu popup for the NoteTaker toolbar.

at the top of a sequence, so we can be more creative with our query. We can start it directly at the URL of the current note. If we do that, an example of the query fact would be:

```
<- http://saturn/test1.html, http://www.mozilla.org/notetaker-
     rdf#summary, ?summary ->
```

Using the advice in this chapter under "Common Query Patterns," we create the template for the keyword textbox as shown in Listing 14.22:

Listing 14.22 Templated textbox for the NoteTaker toolbar.

```
<box datasources="notetaker.rdf" ref="http://saturn/test1.html">
  <template>
    <rule>
      <conditions>
        <content uri="?uri"/>
        <triple subject="?uri"
          predicate="http://www.mozilla.org/notetaker-rdf#summary"
          object="?summary"/>
      </conditions>
      <action>
        <textbox id="notetaker-toolbar.summary" uri="?summary"
          value="?summary"/>
      </action>
    </rule>
  </template>
</box>
```

This code replaces the single tag `<textbox/>` in the existing toolbar. We've surrounded the textbox with an invisible `<box>` just to get the template working.

This template code has a fixed URL as the query start point. We'll shortly make this value dynamic using a script and `setAttribute()`. That concludes the XUL changes to the NoteTaker toolbar. We now turn to templates required for the Edit dialog box. The `<listbox>` and `<tree>` tags in that dialog box both need templates.

14.8.3 An Extended Syntax Template for `<listbox>`

The `<listbox>` tag for this dialog box is the easier task, so that's first. To learn what the query should be, we examine the structure of the `notetaker.rdf` file again. We see that a note is part of a sequence and contains a `keyword` property, so it is in the standard data arrangement. The problem is that the URL for the note is a known, fixed quantity rather than being every note in the sequence. This is a similar situation to the `<textbox>` on the toolbar, so we'll need an extended query.

Another reason for an extended query is to dig out the textual strings for the keywords. Those strings are properties of each keyword rather than prop-

erties of the note. We can easily retrieve them by extending the query further into the set of facts—one additional <triple> tag will link the keywords found on the note to the labels on the keywords. The template we require looks like Listing 14.23.

Listing 14.23 Templated listbox for the NoteTaker toolbar.

```
<listbox id="dialog.keywords" rows="3" datasources="notetaker.rdf"
          ref="http://saturn/test1.html">
  <template>
    <rule>
      <conditions>
        <content uri="?uri"/>
        <triple subject="?uri"
          predicate="http://www.mozilla.org/notetaker-rdf#keyword"
          object="?keyword"/>
        <triple subject="?keyword"
          predicate="http://www.mozilla.org/notetaker-rdf#label"
          object="?text"/>
      </conditions>
      <action>
        <listitem uri="?keyword" label="?text"/>
      </action>
    </rule>
  </template>
</listbox>
```

Except for the addition of one <triple> tag, this code is effectively the same as the code for the toolbar <textbox>.

We could also use a template for the Edit panel of the Edit dialog box. There are good reasons for not doing that, so we'll leave that panel as it is for now. One of those good reasons is that we don't know which note is the best one to display when the URL loads.

14.8.4 An Extended Syntax Template for <tree>

The last template we consider is for the <tree> of related keywords.

To gain some ideas what the query might be for this tree, we return to the notetaker.rdf file. We will need the label property to display the keyword. We will also need the related property to discover keywords related to a given keyword. Finally, we want the level 0 (zero) of the tree to be the keywords of the current note's URL. We'll use that URL as the top of the query, but we'll display only the children of that URL, not the URL itself.

The facts we seem to need for a full, recursive query go like this:

```
<- current-page-url, keyword, ?keyword -> // level 0
<- ?keyword, label, ?text ->

<- ?keyword, related, ?keyword2 ->        // level 1
<- ?keyword2, label, ?text2 ->
```

```
<- ?keyword2, related, ?keyword3 ->       // level 2
<- ?keyword3, label, ?text3 ->

... and so on ...
```

This set of facts doesn't match the standard fact arrangement, so we can't use the simple query syntax. This set of facts also represents a recursive query, so we really need to use a `<tree>` tag for testing.

What will the recursive query be? We need to concentrate on the narrower question: What is the query needed to probe one level deeper into the tree of solutions? There appear to be two facts per level, so it might seem that each step in the recursive query is a two-fact query. If we examine the `notetaker.rdf` fact structure, it appears that there is only one fact required per level of the query because keywords are separated by a single predicate/property. If we draw the query as an RDF diagram, we can quickly see that one fact is the correct approach. Figure 14.8 shows this diagram.

Clearly, the recursive behavior requires one step along the (vertical) arrows only. So the recursive query must be a one-fact query. The other arrows are side information required at each level but not contributing to the recursive behavior.

We consider first this recursive fact represented by vertical arrows. The problem is that sometimes it is labeled with `keyword` and sometimes it is labeled with `related`. In other words, there are two alternatives for the recursive query:

```
<- current-page-url, keyword, ?keyword ->
<- current-keyword-urn, related, ?keyword ->
```

Somehow the template query must accommodate both cases. One solution is to use a separate `<rule>` for each possibility. In our case, the `con-`

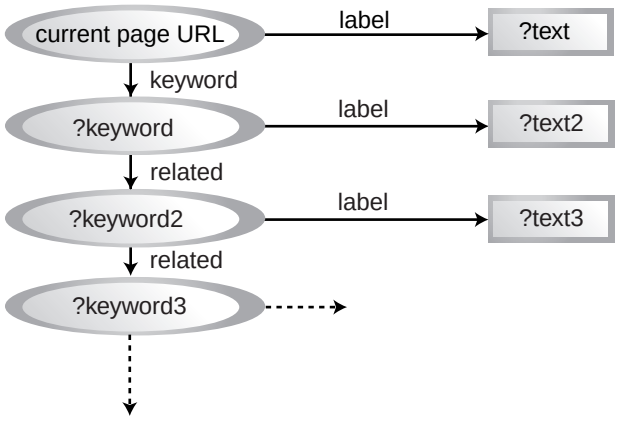


Fig. 14.8 RDF diagram for a recursive template query.

tainment template attribute is a simple solution. We are fortunate that, given any URI, either there will be facts with the related property, or there will be facts with the keyword property, but not both, which means that we can use multiple container predicates at once. We will list both the related and keyword predicates in this attribute's value.

To see why we can use containment, consider the progress of the query. When the query is discovering the top level of the tree, the keyword predicate will find solutions and the related predicate won't, because that's the way the facts are arranged in the RDF file. When the query is discovering solutions deeper in the tree, the related predicate will find solutions and the keyword predicate won't. This is exactly what we want. Listing 14.24 shows the resulting template.

Listing 14.24 Recursive tree template for the NoteTaker Edit dialog box.

```
<tree id="dialog.related" flex="1"
      datasources="notetaker.rdf"
      ref="http://saturn/test2.html"
      containment="http://www.mozilla.org/notetaker-rdf#related http://
          www.mozilla.org/notetaker-rdf#keyword"
      hidecolumnpicker="true"
  >
  <treecols>
    <treecol id="tree.all" hideheader="true" primary="true" flex="1"/>
  </treecols>
  <template>
    <rule>
      <conditions>
        <content uri="?uri"/>
        <member container="?uri" child="?keyword"/>
      </conditions>
      <action>
        <treechildren>
          <treeitem uri="?keyword">
            <treerow>
              <treecell label="?keyword"/>
            </treerow>
          </treeitem>
        </treechildren>
      </action>
    </rule>
  </template>
</tree>
```

The query contains a <member> tag to match the use of the containment attribute. Because this is a recursive query, we can't test it with a plain <description> tag. We must use one of the tags that explicitly supports recursive queries, in this case <tree>.

We have therefore managed to build the recursive query. All that is needed now is to add the ancillary information from Figure 14.8. That is easy

to do using a <binding> tag. We add this content just after the </conditions> tag:

```
<bindings>
  <binding subject="?keyword"
    predicate="http://www.mozilla.org/notetaker-rdf#label"
    object="?text"/>
  </bindings>
```

Because <binding> tags don't assist with the recursive process, we haven't damaged the query; we've just enhanced the information it retrieves. Finally, we must change the <treecell> to report the label that the binding discovers, rather than the URN of the keyword:

```
<treecell label="?text"/>
```

With these changes, we've completed the tree's template, and all the required templates for the NoteTaker tool. Even though the <listbox> and <tree> tags have template-generated content, our event handlers from Chapter 13, Listboxes and Trees, continue to work; they can browse the DOM structure for template-generated content as easily as they can a DOM structure built from static XUL tags.

We do have one problem with the <tree> template. It does not display related keywords as well as the custom view experiment did in Chapter 13, Listboxes and Trees. It only displays what's in the RDF file. We could improve the output by adding the other keyword-keyword combinations to notetaker.rdf, or we could somehow modify the template so that the facts used in the template receive special processing before display, rather than being read straight from the file. We'll consider that latter possibility in Chapter 16, XPCOM Objects.

14.8.5 Coordinating and Refreshing Template Content

The final feature we'd like to implement is keeping the displayed data up to date. That means tying the data in the toolbar, Edit dialog box, and HTML note to the RDF facts about the URL of the Web page currently displayed in the browser. Keeping the HTML note up to date is too hard for the technology we've explored so far, but the other updates are easy. Our plan follows:

1. Update the <textbox> on the toolbar every time the current note changes.
2. Update the dropdown menu on the toolbar every time a note is saved or deleted, in case the total set of keywords changed as a result of that save or delete.
3. Update the Keywords pane of the dialog box every time it is opened.

To do this, we make the following small changes for the toolbar:

1. Update the JavaScript version of the current note to include a URL.
2. Provide a utility function that updates the toolbar display.

3. The timed checks that detect a newly displayed Web page need to update the toolbar `<textbox>`.
4. A `notetaker-delete` command is needed on the delete button on the toolbar.
5. A `notetaker-save` command is needed on the save button on the toolbar.
6. The `notetaker-open-dialog` command must update the dropdown menu on the toolbar when the dialog box closes.

First, we must implement the existing note object as a full JavaScript object and then give it methods `clear()` and `resolve()` and a `url` property. `clear()` removes any current note data; `resolve()` turns a supplied URL into the URL of an existing note and records the results. For this chapter, these changes are trivial, but we'll need to expand on them later. The code for the new note object is shown in Listing 14.25.

Listing 14.25 Basic JavaScript object for a NoteTaker note.

```
function Note() {}  // constructor

Note.prototype = {
  url : null,
  summary : "",
  details : "",
  chop_query : true, home_page : false,
  width : 100, height : 90, top : 80, left : 70,
  clear : function () {
    this.url = null;
  },
  resolve : function (url) {
    this.url = url;
  },
}

var note = new Note();
```

Second, we implement a utility `refresh_toolbar()` function that updates the toolbar templates. Some of this function we'll have to modify in later chapters. Listing 14.26 shows this function.

Listing 14.26 Template rebuilding code for the NoteTaker toolbar.

```
function refresh_toolbar()
{
  var menu = window.document.getElementById('notetaker-toolbar.keywords');
  menu.firstChild.builder.rebuild();

  var box = window.document.getElementById('notetaker-toolbar.summary');
  box.parentNode.setAttribute('ref', note.url);
  box.parentNode.builder.rebuild();
}
```

The `rebuild()` method causes the template-generated contents to be removed and re-created. In the case of the dropdown menu, the contents will change only if the underlying RDF file changes. In the case of the textbox, the template itself is modified, and so the query is different each time it is rebuilt.

Next, we look at the timed checks. This is all run from the `content_poll()` function, so let's rewrite it slightly so that it updates the toolbar as well as the displayed note. Listing 14.27 shows the new version.

Listing 14.27 Poll the displayed Web page and make all required updates.

```
function content_poll()
{
  var doc;
  try {
    if ( !window.content ) throw('fail');
    doc = window.content.document;
    if ( !doc ) throw('fail');
    if ( doc.getElementsByTagName("body").length == 0 ) throw('fail');
    if ( doc.location.href == "about:blank" ) throw('fail');
  }
  catch (e) {
    note.clear();
    refresh_toolbar();
    return;
  }

  if ( doc.visited ) return;

  note.resolve(doc.location.href);
  display_note();
  refresh_toolbar();
  doc.visited = true;
}
```

If there's no suitable URL present, then clear the current note and the toolbar. Otherwise, find the new current note and update the toolbar, note, and current page.

We can't yet complete the remaining tasks 4 and 6 because we don't know how to modify RDF files yet—we only know how to read them. We can, however, do the template update part of the nominated commands. We update the `action()` function to call `refresh_toolbar()` wherever it's needed. Listing 14.28 shows this trivial code.

Listing 14.28 Full list of NoteTaker toolbar commands with template rebuilding.

```
function action(task)
{
  if ( task == "notetaker-open-dialog" )
  {
    window.openDialog("editDialog.xul","_blank","modal=yes");
    refresh_toolbar();
  }
```

```
  if ( task == "notetaker-display" )
  {
    display_note();
  }

  if ( task == "notetaker-save" )
  {
    refresh_toolbar();
  }
  if ( task == "notetaker-delete" )
  {
    refresh_toolbar();
  }
}
```

That completes management of the toolbar templates. For the dialog box, we need an equivalent effect for the templates in the Keywords pane. We choose to put this in the `notetaker-load` command. In the `action()` function that services that command, we add a call to `refresh_dialog()` and implement `refresh_dialog()` as shown in Listing 14.29.

Listing 14.29 Template rebuilding for NoteTaker Edit dialog box.

```
function refresh_dialog()
{
  var listbox = window.document.getElementById('dialog.keywords');
  listbox.setAttribute('ref', window.opener.note.url);
  listbox.builder.rebuild();

  var tree = window.document.getElementById('dialog.keywords');
  tree.setAttribute('ref', window.opener.note.url);
  tree.builder.rebuild();
}
```

This is identical to the template updates for the toolbar drop-down menu.

14.9 Debug Corner: Template Survival Guide

When templates are phrased and used correctly, everything works. There are almost no syntax-related bugs or unstable features to blame failure on. Most problems come from syntax mistakes in the application code.

On Microsoft Windows, be sure to check that the Mozilla Platform has been fully shut down when the last window is closed. If a badly formed template is used, the Mozilla process can linger on and be reused in later tests, which is very confusing. When this happens, the most common symptom is that recent changes to the XUL code or JavaScript appear to have no effect. It is safest to start each test with a new instance of the platform. If you leave the Mozilla splash screen in place, you can use this as a hint to remind you when a new instance has started.

A second platform flaw involves recursive queries. Only attempt to get these working with a <menu> or <tree> widget; other results are unpredictable and can cause crashes.

The combination of RDF documents and XUL templates requires five times as much care as the <tree> tag discussed in Chapter 13, Listboxes and Trees. If just one little thing is incorrect, the template will produce nothing at all, and there will be no clues why that happened. The right approach is essential.

14.9.1 Building Up a Template

When creating a template, always start with test data stored in an external RDF file. Mozilla has full-featured support for a file-based RDF source. External RDF files are easy to view. You can use the techniques described in "Debug Corner" in Chapter 11, RDF, to run the data through Mozilla once. This lets you confirm that it is loading the way that you expect. Load the file directly into a browser so that Mozilla can report any syntax errors. Load the file as an .xml file so that you can see any indentation problems. Alternately, try out one of the RDF validation tools recommended by the W3C.

After gaining confidence with some test data, the next step is to prove that the query output can be displayed in a XUL window.

If the template query is not recursive, then create a template using a simple <vbox> tag as the base tag. Dump something out into a <description> or <label> tag without any fancy features in the rules. After you have seen something appear, you can be sure that you have at least the query root and part of the query structure right. Even if your goal is to use a deeply nested tree structure, try displaying the top-level items by themselves in a <vbox> first.

If your ultimate source of facts is an internal data source, read the advice in Chapter 16, XPCOM Objects, on these data sources and extract into a test page as much information as possible from that data source. Some data sources can be manipulated directly from a template; others need a script. You need to be familiar with the exact content generated, or your template queries won't work.

Next, build up your queries. The template system is flexible and can handle some changes to the order of content inside <template>, but making those changes is not recommended. For the least headaches, stick closely to the order of tags suggested in this chapter. Even though such order can be varied a little, doing so can confuse the internal rule creation system about what template variables are what. Swapping things around can stop data from appearing.

If the template is to be a recursive one, then you can't easily test the recursion outside of a <menu> or <tree> tag; you can only test the first level of the recursion. To test the recursion through multiple levels, use a <tree> tag, not a <menu> tag. It is not possible to build a recursively generated set of menus out of a single <rule>—at least two rules are required. One rule is for

menu items that are <menu>s; the other is for menu items that are <menuitem>s. So for recursive queries, the <tree> tag is the simplest test vehicle.

Only after the queries are yielding results should you think about the real widget you want to use as the base tag. In the case of <tree>, always start with the recommended content described in this chapter, and always make sure that the tree has a primary column. You can alter the other details later. Carefully review the flags and options that can be added to the base tag—some are vital. Automatically add `flags="dont-build-content"` and `flex="1"` attributes to trees, until you can think of a good reason for omitting them.

After your template works properly, you might want to script it so that it has dynamic behavior. "Guidelines for Dynamic Templates" contains very important wisdom. Outside of those guidelines, very little is currently supported.

Finally, note the remarks in "Data Sources" in Chapter 16, XPCOM Objects, about data sources. If your final RDF source is internal, not external, then the data source used needs to be carefully assessed for available functionality. Most internal data sources have unpublished formats for the RDF facts they supply. As a last resort, examine how they are used in the chrome by the already working Mozilla applications.

14.10 SUMMARY

Mozilla's template system extends RDF with a query system and with formatted output of the query results. Those results and output can change dynamically to match dynamic changes to the information underlying the query. Templates work on top of the XUL language and are integrated with the content layout system. They can be used to create dynamic GUI interfaces or just to display changing data in a fixed-sized widget. Templates are not static—they can easily be refreshed or updated if conditions change.

The template system is a technical challenge to learn. It has unfamiliar concepts, it has its share of quirks and stumbling blocks, and it provides very little feedback to the programmer. Awkwardly, the data sources that feed data into the template system are a motley group of individuals whose quirks require further effort to appreciate.

For all those version 1 problems, templates are very powerful tools. An RDF file stored at any URL in the world can be read and displayed using a very brief and concise set of XUL tags. Such simplicity augers well for the future data processing capabilities of the XUL language and the Mozilla Platform. It is likely that templates are only an early step in the data processing capabilities of the platform.

JavaScript and the DOM can be used to generate new content in a XUL document. RDF and templates achieve a similar end, but by a different route. A third way exists to generate content in a XUL document. That method involves XBL, the subject of the next chapter.