Haystack: A Platform for Authoring End User Semantic Web Applications

Dennis Quan, David Huynh, and David R. Karger

MIT Computer Science and Artificial Intelligence Laboratory
200 Technology Square, Cambridge, MA 02139 USA
{dquan,dfhuynh,karger}@ai.mit.edu

Abstract. The Semantic Web promises to open innumerable opportunities for automation and information retrieval by standardizing the protocols for metadata exchange. However, just as the success of the World Wide Web can be attributed to the ease of use and ubiquity of Web browsers, we believe that the unfolding of the Semantic Web vision depends on users getting powerful but easy-to-use tools for managing their information. But unlike HTML, which can be easily edited in any text editor, RDF is more complicated to author and does not have an obvious presentation mechanism. Previous work has concentrated on the ideas of generic RDF graph visualization and RDF Schema-based form generation. In this paper, we present a comprehensive platform for constructing end user applications that create, manipulate, and visualize arbitrary RDF-encoded information, adding another layer to the abstraction cake. We discuss a programming environment specifically designed for manipulating RDF and introduce user interface concepts on top that allow the developer to quickly assemble applications that are based on RDF data models. Also, because user interface specifications and program logic are themselves describable in RDF, applications built upon our framework enjoy properties such as network updatability, extensibility, and end user customizability – all desirable characteristics in the spirit of the Semantic Web.

1 Introduction

One reason underlying the initial success of the World Wide Web is the facility with which people can author Web pages and post them online. Web browsers proved to be an easy client-side platform on which to develop, due to the simplicity and forgiving nature of HTML syntax and the quick turnaround time of the edit-debug process of authoring HTML content. HTML was also sufficiently expressive as a layout language that creative page designs could be realized. Early adopters found a whole new medium in which to express and share their thoughts, designs, and artwork. As HTML matured, programming languages such as JavaScript were called upon to provide support for implementing client-side dynamic content, making HTML even more expressive.

Perhaps an even more important reason for the Web's success is the fact that HTML-based content is extremely easy to navigate. Using the almost ubiquitous Web browser, content located virtually anywhere in the world, regardless of the server on which it is hosted, can be browsed with point-and-click simplicity.

D. Fensel et al. (Eds.): ISWC 2003, LNCS 2870, pp. 738–753, 2003.

In contrast, the Resource Description Framework (RDF) [2], the corresponding standard language for the Semantic Web [3], enjoys none of these properties. Composition of RDF is difficult in its XML form, as is evidenced by the creation of several alternate syntaxes for RDF [5]. Separate from the syntax is the conceptual difficulty of crystallizing knowledge in terms of ontologies, a more complicated process than copying and pasting pieces of hypertext. Furthermore, there are no standard approaches to visualizing RDF, and the generalized approaches of graph visualization and key/value pair editing employed by many projects do not provide the intuitive interface presented by the Web [17].

For the Semantic Web to develop organically, various kinds of users must be able to participate in its growth. User interfaces must be constructed to facilitate the creation and distribution of RDF-encoded information and to visualize extant RDF metadata on the Semantic Web in an intuitive fashion. Developers will need tools for producing such user interfaces that give them easy access to RDF data and user interface components that are specially designed to handle the generality of RDF's data model.

An example of a user interface that gives normal humans the ability to interact with RDF is Haystack [1]. Haystack brings the Semantic Web to end users by leveraging key Semantic Web technologies that allow users to easily manage their documents, e-mail messages, appointments, tasks, etc. The Haystack user interface is capable of visualizing a variety of different types of information; meanwhile, the interface gives few clues to the notion that the underlying data model is represented in RDF. Presenting information in a manner familiar and intuitive to users is key, as few users are familiar with ontological vocabulary and descriptive logic. Additionally, users are unlikely to accept a system that requires them to explicitly shuttle information between their current systems and an RDF representation. In other words, end user Semantic Web applications need to be developed in such a way that users need not even be aware that the Semantic Web is involved!

In addition to serving as an exemplar, Haystack has been built as an extensible platform that allows various kinds of functionality to be developed easily and independently and incorporated seamlessly. In this paper we describe our observations on the kinds of tools that are needed by developers of RDF-based client software and demonstrate these key concepts of the Haystack system that can be reused by others.

2 Approach

The layers of Haystack's infrastructure are designed to tackle specific aspects of the problem of creating end user Semantic Web applications. Enabling the data layer of the system is Adenine, a new domain-specific programming language we have developed for manipulating RDF data. Like RDF/XML and Notation3 [5], it can be used to record RDF, but unlike them, it can express programming constructs that manipulate such data. Adenine adopts a combination of Python, Notation3, and Scheme [9] syntax in order to conveniently express frequently-used RDF operations. Furthermore, because Adenine can be compiled into an RDF representation, Adenine code and RDF data can be freely intermixed and distributed together.

The basis for the Haystack system is a layer that supports back-end components called services that are responsible for incorporating data from other systems and processing existing data in the background. Haystack's RDF information store holds all RDF data known to the system and serves as a blackboard that coordinates the workings of different services, allowing one service to build on the results produced by other services. Services can be written in a variety of languages, including Java, Python, and Adenine.

We turn our attention to the problems of presenting the RDF information that is managed by services to the user. As mentioned, one important part of the appeal of HTML is its expressiveness and ease in coding layout and presentation. Haystack supports an analogous, extensible user interface ontology called Ozone that exploits the power of RDF to describe on screen presentation. Using Ozone we can construct user interface elements called views that represent resources described in RDF on screen.

Not only do we need to present RDF data to the user, but we also need to give users intuitive tools with which to interact with such data. We allow users to manipulate resources with direct manipulation techniques such as context menus and drag and drop. The actual commands that are exposed by such techniques are specified according to an ontology for declaring operations on RDF data. Operations – akin to menu items and toolbar buttons in existing environments – can be defined to work on specific classes of RDF resources and are written in Adenine.

A special type of operation is object and document creation, which is the explicit means through which the user adds data to the system. We define the notion of a constructor, an adaptation of templates, factories, and other construction paradigms used in object-oriented systems [4], to the Semantic Web. Constructors, like operations, are Adenine functions that set up the basic properties of an object, potentially also displaying a user interface to prompt the user for necessary information in the process. We will show how this abstraction can address the issue of how users create new resources and describe existing resources to the system.

Our contributions can be reused in systems other than Haystack. Adenine, for instance, can code information processing algorithms on Web servers that handle RDF data. Haystack's UI framework can be adapted to serve Dynamic HTML pages built up by nesting HTML representations of pieces of RDF data. However, it is through the Haystack system that we wish to illustrate how the combined use of all of these techniques can ease the development of an environment that brings the benefits of the Semantic Web directly to end users.

3 Related Work

We believe that the availability of tools for prototyping and building programs that both produce content for and render content from the Semantic Web can help to improve the reception of Semantic Web technologies. The current generation of tools represents the first step in this direction in that they expose programming interfaces for manipulating information. Toolkits for generating, processing, and visualizing graphs of RDF data are widely available on most platforms [14] [15]. Tools for editing data according to specific ontologies, such as Ont-O-Mat and Protégé, give knowledge engineers powerful tools for creating and manipulating data that

corresponds to specific schemata [10] [11]. Furthermore, server-side software packages have been developed to aggregate RDF information for presentation to users [13].

Building on these toolkits, Haystack exposes functionality to users for interacting with information at higher levels of abstraction. Rather than exposing information as a series of RDF statements, Haystack concentrates on the concepts that are important to users of that information: documents, messages, properties, annotations, etc. The Placeless Documents project at Xerox PARC [3] similarly developed an architecture for storing documents based on properties specified by the user and by the system. Both Haystack and Placeless Documents support arbitrary properties on objects and a collection mechanism for aggregating documents. It also specified in its schema access control attributes and shared properties useful for collaboration. We have taken advantage of many ideas that arose from this research in developing the user interface paradigms exposed to users in Haystack for working with RDF-encoded information.

4 Adenine Programming Language

In any system built upon an RDF data model, a sizeable amount of code – both in services and in user interface components – is devoted to the creation and manipulation of RDF-encoded metadata. We observed early on that the development of a language that facilitated the types of operations we frequently perform with RDF would greatly increase our productivity. This lead to the creation of Adenine. An example snippet of Adenine code is given below.

```
# Prefixes for simplifying input of URIs
@prefix : <urn:test-namespace:>

:ImportantMethod rdf:type rdfs:Class

method :expandDerivedClasses ;
rdf:type :ImportantMethod ;
rdfs:comment "x rdf:type y, y rdfs:subClassOf z => x rdf:type z"
        # Perform query
        # First parameter is the query specification
        # Second is a list of the variables to return,
        # in order
        = data (query {
                ?x rdf:type ?y
                ?y rdfs:subClassOf ?z
        } @(?x ?z))

        # Assert base class types
        for x in data
                # Here, x[0] refers to ?x
                # and x[1] refers to ?z
                add { x[0] rdf:type x[1] }
```

The impetus for creating this language is twofold. The first key motivation is having the language's syntax support the data model. Introducing the RDF data model into a standard object-oriented language is fairly straightforward; after all, object-oriented languages were designed specifically to be extensible in this fashion. Normally, one creates a class library to support the required objects. However, more

advanced manipulation paradigms specific to an object model begin to tax the syntax of the language. In languages such as C++, C#, and Python, operator overloading allows programmers to reuse built-in operators for manipulating objects, but one is restricted to the existing syntax of the language; one cannot easily construct new syntactic structures. In Java, operator overloading is not supported, and this results in verbose APIs being created for any object-oriented system.

Arguably, this verbosity can be said to improve the readability of code. On the other hand, lack of syntactic support for a specific object model can be a hindrance to rapid development. Programs can end up being much longer than necessary because of the verbose syntactic structures used. This is the reason behind the popularity of domain-specific programming languages, such as those used in Matlab, Macromedia Director, etc. Adenine is such a language. It includes native support for RDF data types and makes it easy to interact with RDF stores and RDF-based services.

4.1 RDF Representation

The other motivation for creating Adenine was to be able to combine executable code with data in the same representation. To achieve this, Adenine is compilable directly into RDF according to the Adenine ontology. The benefits of this capability can be classified as portability and extensibility. Since 1996, bytecode-based virtual machine execution models have resurged as a result of Java's popularity. Their key benefit has been portability, enabling interpretation of software written for these platforms on vastly different computing environments. In essence, bytecode is a set of instructions written to a portable, predetermined, and byte-encoded ontology.

Adenine takes the bytecode concept one step further by making the ontology explicit and extensible and by replacing byte codes with RDF. In other words, instructions are represented as RDF resources, connected by "next instruction" predicates. Execution occurs by following a chain of such instruction resources. Instead of dealing with the syntactic issue of introducing byte codes for new instructions and semantics, Adenine takes advantage of RDF's ability to extend the directed "object code" graph with new instruction node types.

One recent example of a system that uses metadata-extensible languages is Microsoft's Common Language Runtime (CLR). In a language such as C#, *developer-defined* attributes can be placed on methods, classes, and fields to declare metadata ranging from thread safety to serializability. Compare this to Java, where serializability was introduced only through the creation of a new *language keyword* called "transient". The keyword approach requires knowledge of these extensions by the compiler; the attributes approach delegates this knowledge to the runtime and makes the language truly extensible.

In Adenine, RDF assertions can be applied to any statement, such as comments, classifications, authorship attributions, and information about concurrency safety. This fact enables a number of different features, from self-modifying code to automated object code analysis. Most importantly, it means that Adenine can be packaged together with schemas and other ontological metadata and manipulated in the same fashion as other RDF data. In particular, one feature that has proven to be highly useful is the ability to annotate functions with specialized types such as "asynchronous constructor" or "query operator". This feature is used heavily in the implementation of operations, which is discussed later in this paper.

Adenine's RDF representation and its treatment of the RDF triple as a native data type make Adenine very similar to Lisp, in that both support open-ended data models and both blur the distinction between data and code. However, there are some significant differences. The most superficial difference is that Adenine's syntax and semantics are especially well-suited to manipulating RDF data. Adenine is mostly statically scoped, but exposes dynamic variables that address the current RDF containers from which existing statements are queried and to which new statements are written. (An RDF container is simply a data structure that holds RDF statements.) Adenine's runtime model is also better adapted to being run off of an RDF container. Unlike most modern languages, Adenine supports two types of program state: in-memory, as is with most programming languages, and RDF container-based. Adenine in effect supports two kinds of closures, one being an in-memory closure as is in Lisp, and the other being persistent in an RDF container. This affords the developer more explicit control over the persistence model for Adenine programs and makes it possible for services written in Adenine to be distributed.

4.2 Defining Data in Adenine

RDF data is written in much the same way in Adenine as it is in Notation3. Double quotes enclose RDF literals and create instances of the `Literal` class. Angle brackets (`<>`) enclose URIs and create instances of the `Resource` class.

Prefixes can be declared as a convenient way of referring to frequently-used URIs. For example:

```
@prefix test: <http://test.org/>

if (== test:hi-there <http://test.org/hi-there>)
    print 'Success!'
```

The `rdf`, `rdfs`, `daml`, `xsd`, and `adenine` prefixes are predefined with their standard values.[1]

Collections of RDF statements are enclosed within curly braces (`{}`). The tokens within the `{}` operator are of the form:

```
{ [subject] [predicate] [object]
  [subject2] [predicate2] [object2] … }
```

No separator is required between consecutive statements, unlike Notation3. The semicolon (`;`) can be used in the subject field to refer to the last used subject. Expressions within the `{}` operator are handled as follows: Expressions that evaluate to `Resource` or `Literal` objects are used directly. Lists are expressed with the `@()` operator and are expressed as DAML+OIL lists. Other objects are converted into `Literal`'s. The `{}` expression itself evaluates to an object exposing the `IRDFContainer` interface.

---

[1] `rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs: http://www.w3.org/2000/01/rdf-schema#`
`daml: http://www.daml.org/2001/03/daml+oil#`
`xsd: http://www.w3.org/2001/XMLSchema#`
`adenine: http://haystack.lcs.mit.edu/schemata/adenine#`

Anonymous nodes can be created from Adenine using the `${}` operator (the equivalent of the `[]` operator in Notation3). Syntactically, an anonymous node expression has type `Resource` and can be used anywhere a resource is needed. This feature is useful when you need a unique, "anonymous" URI for a set of statements. The following set of statements states that Mary's son is 15 years old and is named "Bob" (`add` is the command used to insert RDF statements into the store):

```
add {
    <urn:person:mary> <urn:person:hasSon> ${
        <urn:person:age> "15" ;
        <urn:person:name> "Bob"
    }
}
```

4.3 Writing Executable Code

The syntax of Adenine code resembles a combination of Python and Lisp. As in Python, indentation levels denote lexical block structure (indentation is ignored within `{}` expressions). Adenine is an imperative language, and as such contains standard constructs such as functions, for loops, arrays, and objects. Function calls resemble Lisp syntax in that they are enclosed in parentheses and do not use commas to separate parameters. Arrays are indexed with square brackets as they are in Python or Java. Also, because the Adenine interpreter is written in Java, Adenine code can call methods and access fields of Java objects using the dot operator, as is done in Java or Python. The execution model is quite similar to that of Java and Python in that an in-memory environment is used to store variables; in particular, execution state is *not* represented in RDF. Values in Adenine are represented as Java objects.

Adenine methods are functions that are named by URI and are compiled into RDF. To execute these functions, the Adenine interpreter is instantiated and passed the URI of the method to be run and the parameters to pass to it. The interpreter then constructs an initial in-memory environment binding standard names to built-in functions and executes the code one instruction at a time. Because methods are simply resources of type `adenine:Method`, one can also specify other metadata for methods, as was mentioned earlier. In the example given, an `rdfs:comment` is declared and the method is given an additional type, and these assertions will be entered directly into the RDF container that receives the compiled Adenine code.

Adenine methods are usually executed by interpretation of a method's instructions from an RDF store. A prototype interpreter has been implemented in Java and is used to run much of Haystack. However, to improve performance, a tool is available for compiling Adenine methods into Java Virtual Machine bytecode. While eliminating some of the dynamic nature of Adenine, translation into Java does provide a significant performance increase.

The top level of an Adenine file is used for data (i.e., `add` instructions) and method declarations and cannot contain executable code. This is because Adenine is in essence an alternate syntax for RDF. Within method declarations, however, is code that is compiled into RDF; hence, method declarations are like syntactic sugar for the equivalent Adenine RDF "bytecode".

Development on Adenine is ongoing, and Adenine is being used as a platform for testing new ideas in writing RDF-manipulating services and user interface

components. More information about Adenine can be found at the following URL off of our website: http://haystack.lcs.mit.edu/documentation/adenine.pdf.

5 Services

In the past, programs that aggregated data from multiple sources, such as mail merge or customer relationship management, had to be capable of speaking numerous protocols with different back ends to generate their results. With a rich corpus of information described in a single format, namely RDF, the possibility for automation becomes significant because services can now be written against a single unified abstraction. In Haystack, *services* encapsulate key pieces of functionality that manipulate RDF data and execute independently of the user interface. Furthermore, services can be written to help users deal with problem such as information overload by extracting key information from e-mail messages and other documents and presenting the user with summaries. In short, services massage data of importance to the user for consumption by the user interface.

Services in Haystack are callable entities that expose a Java interface. (A Java-implemented stub class that calls Adenine methods is also available and frequently used.) The core services are mostly written in Java, but some are written in Adenine and some in Python (these services are hosted by the Jython interpreter). We utilize an RDF ontology derived from WSDL [6] for describing the interfaces to services as well as for noting which server processes hosts which services. As a consequence, we are able to support different protocols for communicating between services, from simply passing in-process Java objects around to using HTTP-based RPC mechanisms such as HTTP POST and SOAP [8]. In other words, Haystack services are in effect Web Services whose implementation implements the `edu.mit.lcs.haystack.server.service.IService` Java interface and where the appropriate WSDL metadata has been entered into the store; the system takes care of exposing services via whatever protocols are supported.

One specific class of service is of great importance in Haystack: the RDF store. RDF stores, as their name implies, hold RDF statements and allow clients to query their contents. As all persistent system state is described in RDF, Haystack uses RDF stores much as modern software uses the file system.

5.1 Core Infrastructure

Sitting at the core of the Haystack system is a service manager, a Java process that is responsible for starting up the services it hosts. At system startup the service manager reads an RDF configuration file to determine where the root RDF store is. The service manager then connects to this root store, much as a UNIX system mounts its root file system at startup, and determines what services should be started based on the values of the `config:hostsService` property of the service manager's resource (all service managers are named by URIs).

All services are run within the context of a root store and a service manager. The root store provides a container for services to persist their state. Furthermore, the service manager is responsible for allowing services to connect to one another. If a

service requests to connect to a service running on the same service manager, the service manager can return a reference to the other service directly; otherwise, the service manager uses the information about the service encoded in the WSDL ontology to construct a proxy.

Because services in Haystack share an underlying store, services can interoperate with each other by treating the store as a "blackboard". Blackboard architectures permit multiple services to attack a problem by allowing services to use information on the blackboard to perform some specific analysis and to pose new information that is derived from that analysis. RDF stores have built-in support for registering events, which allows services to learn when new information (i.e., RDF statements) has been posted to the store. New functionality can be introduced by adding services that perform certain tasks when specific forms of information enter the system.

5.2 Automation

One useful application for services that is core to the Semantic Web is automation. Services are used in Haystack to automatically retrieve and process information from various sources, such as e-mail, calendars, the World Wide Web, etc. Haystack includes services that retrieve e-mail from POP3 servers, extract plaintext from HTML pages, generate text summaries, perform text-based classification, download RSS subscriptions on a regular basis, fulfill queries, and interface with the file system and LDAP servers.

Services are particularly useful for analyzing collections of documents and finding patterns, which can then aid the system when trying to present such a collection to the user. Modern information retrieval algorithms are capable of grouping documents by similarity or other metrics, and previous work has found these automatic classifications to be useful in many situations [19]. Additionally, users can build collections prescriptively by making a query. A service, armed with a specification of what a user is looking for, can create a collection from the results of a query, and it can watch for new data entering the system that matches the query. For example, one service that exists in Haystack automatically filters a user's e-mail for documents that appear to fit in one or more collections defined by the user, such as "Website Project" or "Letters from Mom" [19].

6 Ozone Presentation Ontology

We have defined an ontology called Ozone that can be used to encode page layout and content much like that expressible in HTML. The following code snippet illustrates how a simple page (Fig. 1) can be authored in Ozone:

```
@prefix slide:
<http://haystack.lcs.mit.edu/schemata/ozoneslide#>

= mySlide ${
   rdf:type           slide:Slide ;
   slide:margin              "10" ;
   slide:bgcolor             "lightGray" ;
   slide:color        "#444444" ;
```

Fig. 1. Sample slide

```
    slide:fontFamily           "Arial" ;
    slide:fontSize             "10" ;
    slide:child ${
      rdf:type          slide:Paragraph ;
      slide:children @(
       ${  rdf:type             slide:Text ;
           slide:text   "Welcome to Haystack" ;
           slide:fontSize       "120%" ;
           slide:fontBold       "true"
       }
       ${  rdf:type             slide:Break }
       ${  rdf:type             slide:Text ;
           slide:text   "The current user is:"
       }
       ${  rdf:type             slide:Block ;
           slide:marginLeft     "20" ;
           slide:borderWidth    "1" ;
           slide:child ${
             rdf:type  slide:Paragraph ;
             slide:children @(
               ${  rdf:type            ozone:ViewContainer ;
                   ozone:initialResource (__identity__.getResource) ;
                   ozone:viewPartClass   ozone:InlineViewPart
               }
             )
           }
      }
    )
  }
}
```

The code specifies a new *slide* (analogous to an HTML page) with all margins set to 10 pixels, the background color set to light gray, and the foreground (text) color set to a dark shade of gray as defined by an RGB triple. The text on the page will be in Arial, 10 point. These color and font settings are inherited by all descendant resources of the slide; they can also be overridden by the descendant resources when necessary, as is the case with Cascading Style Sheets.

The sample slide has one child, a `slide:Paragraph` resource (similar to the `<P>` tag in HTML). The `slide:Paragraph` resource has four child resources: two `slide:Text` resources, one `slide:Break` resource, and one `slide:Block` resource. The first `slide:Text` resource redefines its font size and boldens its text. The Block resource is like the `<DIV>` tag in HTML: it allows specification of block-specific attributes such as margins, borders, clearances, drop shadow, etc. Inside the Block resource is a placeholder for a view (discussed later), which renders the name of the current user. The current user is expressed by the Adenine expression `(__identity__.getResource)`, which is embedded within the slide definition.

Note the hierarchical form of the code snippet: in this way, Ozone is very similar to HTML and should be somewhat familiar to HTML programmers who know RDF. Adenine makes it easy to write pieces of code that can both manipulate RDF data and generate Ozone data. This is important when, as in many cases, the Ozone data to be generated depends on data in the RDF store.

More information about Ozone can be found on our website at the following URL: http://haystack.lcs.mit.edu/documentation/ui.pdf.

7 Views as Representations of Resources

Using Ozone we can construct user interface elements called *views* that present information about resources in the RDF store. Specifically, a view is a component that displays certain types of resources in a particular way. A given RDF class may have any number of different views associated with it. Furthermore, views are described in RDF, allowing a view to be characterized according to the RDF classes it supports and how it displays resources (e.g., full screen, in a one line summary, as an applet-sized view, etc.). When a resource needs to be displayed in Haystack in a certain way, such as full screen, a view is chosen that possesses the necessary characteristics.

As components, views enable pieces of user interface functionality to be reused. The developer of a one line summary view for people (perhaps displaying a person's name and telephone number) provides an RDF description to the system that enables developers that need to display summaries of contacts to reuse the component. The best example of reuse can be seen in the case of views that embed views of other resources. For example, a view of an address book containing contacts and mailing lists needs not implement views for displaying contacts and mailing lists; Ozone provides a way for views to specify that a resource needs to be displayed at a certain location on the screen in a certain fashion (e.g., as a one line summary). In this way composite views can be constructed that leverage the specialized user interface functionality of the child views that are embedded.

When a view is instantiated, the system passes the view a *context object* that informs it of the resource to be displayed. The context object also contains a pointer to the parent view's context object, if one exists as a result of a view being embedded within another view. In this way views are made aware of the context in which they are displaying information. For example, if an address book view is displaying a list of people by embedding individual person views, the person view can know not to display the "Add to Address Book" button, since it knows that it is embedded within the address book's view and hence is displaying a resource that is already in the address book.

Also, because the system is responsible for instantiating views and keeping track of where child views are to be embedded within parent views, the system can provide default implementations of certain direct manipulation features for free. A good example is drag and drop: When the user starts to drag on a view, the system knows what resource is being represented by that view, such that when the view is dropped elsewhere in the user interface, the drop target can be informed of what resource was involved instead of simply the textual or graphical content of the particular representation that was dragged.

Take the example of filling in a list of meeting attendees on a form. Instead of retyping or copying and pasting names of people from an address book, a user can drag and drop contacts from an address book into the list. Because the views representing contacts in the address book are associated with the resources they represent and not just the names of the contacts, the identities of the contacts' resources can be preserved. The alternative opens the possibility for ambiguity because information is lost. For example, what if there are two people named "John Doe" known to the system? Specifying the text string alone is not sufficient to disambiguate which John Doe is intended, even though it is clear that the John Doe desired is the one that the user selected in the address book.

8 Operations

Most systems provide some mechanism for exposing prepackaged functionality that can be applied under specific circumstances. For example, in Java one can expose methods in a class definition that perform specific tasks when invoked. In C one can define functions that accept arguments of particular types. Under Windows, one can define verbs, which are bound to specific file types and perform actions such as opening or printing a document when activated through a context menu in the Windows Explorer shell. In general, these mechanisms all permit parameterized operations to be defined and exposed to clients.

In Haystack, the analogous construct is called an *operation*, which can accept any number of parameters of certain types and perform some task. Operations are Adenine methods annotated with key metadata such as parameter types [18]. The operation ontology is best explained in the context of an example. The definition of the "Browse To" operation is given in the following code snippet.

```
@prefix op: <http://haystack.lcs.mit.edu/schemata/operation#>

add { :target
    rdf:type          op:Parameter ;
    rdf:type          daml:ObjectProperty ;
    rdfs:label        "Target" ;
    op:required              "true" ;
    rdfs:range        daml:Thing
}

method :browseTo :target = target ;
rdf:type            op:Operation ;
dc:title        "Browse to" ;
ozone:icon      <http://haystack.lcs.mit.edu/icons/verbs/browseto.gif> ;
adenine:preload "true"
  ozone:navigate target[0]
```

The definition of an operation (e.g. `:browseToOperation`) includes basic information such as its name, an icon, as well as a set of named parameters. Notice that operations are defined using the method syntax; this is possible because operation is a subclass of Adenine method. Parameters (e.g. `:target`) are also given names, but in addition parameters can also be typed, in a variety of different ways. The most basic mechanism for typing is simply specifying an `rdfs:Class` as a parameter's class using the `rdfs:range` predicate. A parameter's type can also be constrained by giving

an Adenine validator method, which given a value verifies that it can be used for that parameter. Finally, parameters can be specified to be either mandatory or optional.

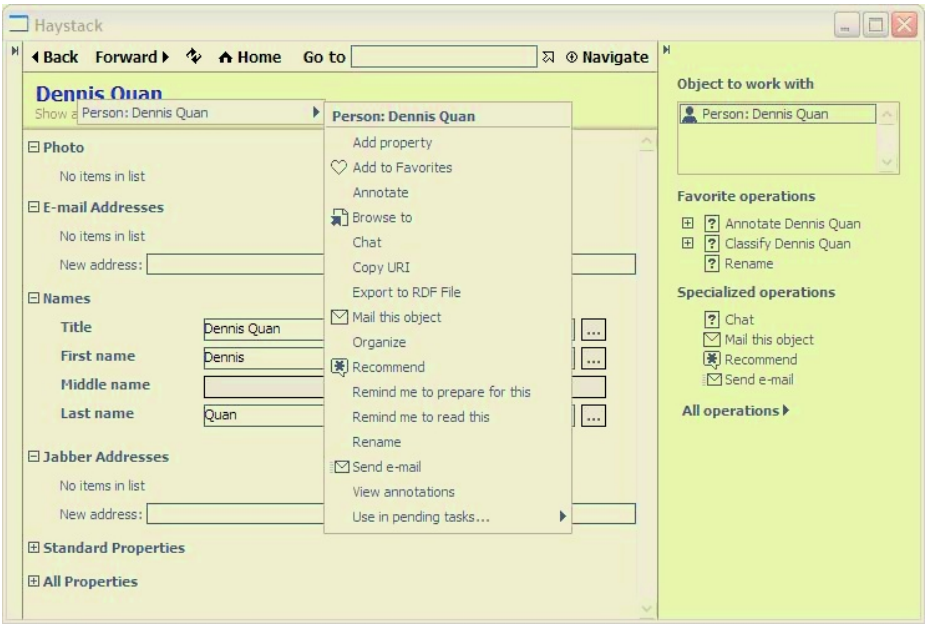

Fig. 2. Operations in Haystack

When an operation is invoked, the values assigned to the operation's parameters are passed to the operation. Parameters can have multiple values; for example, a send mail operation may allow multiple recipients to be specified. To allow for this, the Adenine method receives a list of all values for each named parameter.

The Haystack user interface exposes the operations installed in the system in various ways. Operations are displayed on the tool pane (the right hand pane) in Haystack as well as in context menus (Fig. 2). In fact, operations are also used for commands such as "Shutdown Haystack", where no parameters are needed. In this way, operations can play the roles normally played by menus and toolbars in applications today.

Furthermore, the Haystack framework eliminates the need for developers to create specialized user interfaces for user-performable operations in many cases. When an operation that requires parameters is activated, Haystack checks to see if the target object (in the case of the command being issued from a context menu or the tool pane) satisfies any of the operation's parameters. If there are unresolved parameters, Haystack presents a *UI continuation*, depicted in Fig. 3 [18].

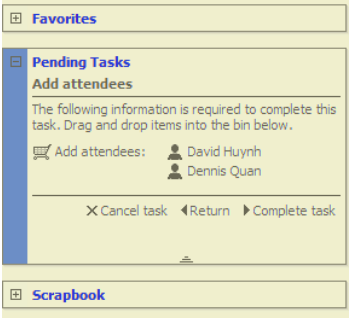

Fig. 3. Sample UI continuation (taken from left hand pane)

Like a dialog box, a UI continuation prompts the user for needed information – in this case, the unresolved parameters. However, unlike most dialog boxes, which are modal, UI continuations are modelessly placed on the left hand pane, allowing the user to use whatever tools in the system he or she is most familiar with to find the information needed to complete the operation. By default, the system takes the user to a convenient place to find the required information, such as in the case of a send e-mail operation, the user's address book. This interface is similar to a shopping cart on an e-business website: the user can drag and drop relevant items into the "bins" representing the operation's parameters. The user can even decide to perform other tasks and come back to the operation later. When the user has finished obtaining the necessary information and is ready to commence the operation, he or she can click the "Done" button on the UI continuation. The system then returns to the state that was present when the operation was initiated (hence the term continuation) and performs the operation. By providing UI continuation functionality, the system frees the developer from needing to design specialized, miniature user interfaces for retrieving information from within modal dialog boxes by reusing the existing browsing environment and at the same time providing the user with a seamless experience.

The operation abstraction allows the functionality of the system to be arbitrarily extended, without special plug-in interfaces or points of extensibility needing to be defined on a per-application basis. Furthermore, developers can declaratively specify new functionality to the system rather than modify monolithic dialog boxes, menus, or toolbars. However, since the UI continuation is displayed using Haystack's view technology, developers are free to customize the display of a UI continuation by defining new view parts.

9 Constructors

The operations ontology is able to describe a large portion of the functionality exposed by an application. However, one particular type of functionality provided by many applications deserves special focus: object creation. Object creation manifests itself in many different forms, ranging from the addition of a text box to a slide in a presentation graphics program to the composing of an e-mail. Applications that support object creation usually expose interfaces for allowing users to choose the appropriate type of object to create or to find a template or wizard that can help guide them through the process of creating the object.

In RDF, the process of creation can naïvely be thought of as the coining of a fresh URI followed by an `rdf:type` assertion. The corresponding choice list for creating objects in RDF could be implemented by displaying a list of all `rdfs:Class` resources known by the system. However, there are many issues not addressed by this solution. The user's mental model of object creation may map onto three distinct activities in the programmatic sense: (1) creation of the resource; (2) establishing some default view; (3) population of the resource with default data. For example, the creation of a picture album from the perspective of the data model is straightforward in that a picture album is simply a collection of resources that happen to be pictures. However, if the user begins viewing this blank picture album with an address book view, he or she may believe that the system has created the wrong object. With respect to the third point, Gamma et al. assert that object creation can come about in various ways,

ranging from straightforward instantiation to creating objects according to some fixed pattern [4].

Furthermore, the classical framing of the object creation problem does not address the user interface implications entailed by certain kinds of instantiations. Some objects can be created without further input from the user, such as empty collections, while some objects require configuration data or other information to be properly initialized, such as a POP3 mail service.

To solve these problems, Haystack makes use of a constructor ontology, which describes resources called *constructors* that create objects. Constructors have type `construct:Constructor`, which derives from `adenine:Method`. (Constructors that are exposed to the user also have type `op:Operation`.) Like all other objects in Haystack, constructors can be browsed to in the user interface and have custom views associated with them. The default view for a constructor's UI continuation simply contains a button that invokes the constructor and browses to the created object. However, for constructors that require a custom user interface to be presented, a custom view part can be provided with specific controls for creating the object. Fig. 4 shows an example of the annotation pane in Haystack, which takes advantage of this functionality. Annotations in Haystack are not limited to text but can be constructed from any kind of object. The annotation pane exposes a drop down list of possible constructors; when the user completes the constructor, the newly created annotation is hooked to the object being annotated.

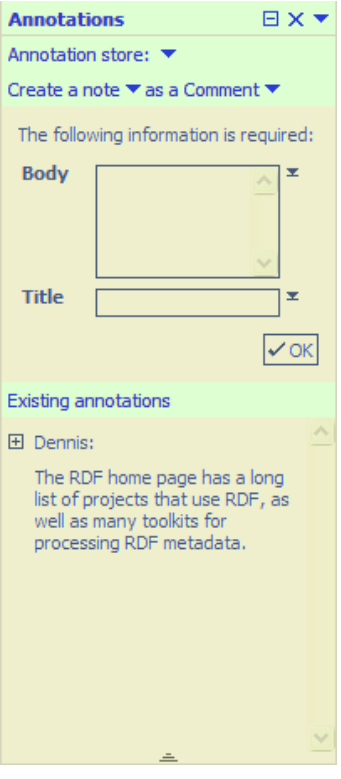

Fig. 4. Annotation UI

10 Conclusion

In this paper we have explored a number of the tools built into Haystack for developing Semantic Web applications for end users. These tools focus on applying RDF technology to improving the developer experience, by allowing developers to declaratively define concepts such as operations and user interface components. Many of these technologies have been built on top of Adenine, which facilitates the manipulation of RDF data and provides syntactic sugar for defining RDF ontologies and user interface designs. We believe these tools have lowered the barrier for creating truly usable and compelling applications that can deliver on the promises of automation and uninhibited data exchange on the Semantic Web.

Acknowledgements. This work was supported by the MIT-NTT collaboration, Project Oxygen, and IBM.

References

1. Huynh, D., Karger, D., and Quan, D.: Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. Proceedings of Semantic Web Workshop, WWW2002. http://haystack.lcs.mit.edu/papers/sww02.pdf
2. Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
3. Berners-Lee, T., Hendler, J., and Lassila, O.: The Semantic Web. Scientific American, May 2001
4. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns. Boston: Addison Wesley, 1995
5. Berners-Lee, T.: Primer: Getting into RDF & Semantic Web using N3. http://www.w3.org/2000/10/swap/Primer.html
6. Christensen, E., Cubera, F., Meredith, G., and Weerawarana, S. (ed.): Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl
7. Dourish, P., Edwards, W.K., et al.: Extending Document Management Systems with User-Specific Active Properties. ACM Transactions on Information Systems, Vol. 18, No. 2, April 2000, 140–170
8. Box, D., Ehnebuske, D., Kavivaya, G., et al. (ed.): SOAP: Simple Object Access Protocol. http://msdn.microsoft.com/library/en-us/dnsoapsp/html/soapspec.asp
9. Abelson, H, Dybvig, R., Haynes, C., Rozas, G., et al.: Revised Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation, Vol. 11, Issue 1, August 2000, 7–105
10. Eriksson, H., Fergerson, R., Shahar, Y., and Musen, M.: Automatic Generation of Ontology Editors. In Proceedings of the 12th Banff Knowledge Acquisition Workshop, 1999
11. Handschuh, S., Staab, S., and Maedche, A.: CREAM – Creating relational metadata with a component-based ontology-driven annotation framework. Proceedings of K-CAP '01
12. Horrocks, I. et al. (ed.): DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html
13. Stojanovic, N., Maedche, A., Staab, S., Studer, R., Sure, Y.: SEAL: a framework for developing SEmantic PortALs. Proceedings of the International Conference on Knowledge Capture, October 2001
14. Pietriga, E.: IsaViz. http://www.w3.org/2001/11/IsaViz/
15. Carroll, J.: Unparsing RDF/XML. Proceedings of WWW2002
16. Huynh, D., Quan, D., and Karger, D.: Haystack's User Experience for Interacting with Semistructured Information. Proceedings of WWW2003
17. Quan, D., Karger, D., and Huynh, D.: RDF Authoring Environments for End Users. Proceedings of Semantic Web Foundations and Application Technologies 2003
18. Quan, D., Huynh, D., Karger, D., and Miller, R.: User Interface Continuations. To appear in Proceedings of UIST 2003
19. Rosen, M.: E-mail Classification in the Haystack Framework, Master's Thesis, February 2003