Haystack: A Platform for Authoring End User Semantic Web Applications

Dennis Quan, David Huynh, David R. Karger
MIT AI Laboratory/LCS, 200 Technology Square, Cambridge, MA 02139 USA
{dquan,dfhuynh,karger}@ai.mit.edu

1. OVERVIEW

The Web fulfilled a practical need of users in allowing them to conveniently author, browse and share content online. Similarly, we believe that Semantic Web technologies will be more quickly proliferated if they can prove to be directly useful to end users. The Haystack project [5] 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, including e-mails, to-do items, news feeds, and web page bookmarks.

The way in which the Haystack interface is constructed gives few clues to the notion that the underlying data model is represented using the Resource Description Framework (RDF)—the standard data representation language of the Semantic Web. Presenting information in a manner familiar and intuitive to users is key, as few users are familiar with ontological vocabulary and descriptive logic. 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 support of this cause, Haystack has been built as an extensible platform that allows various kinds of functionality to be developed easily and independently, and incorporated seamlessly. We wish to make the authoring of RDF-based content and Haystack features that act on such content as easy as authoring HTML web content. In this paper we describe the tools Haystack provides to Semantic Web developers for building end user applications based on RDF.

2. 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 [6]. 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 [2] [4]. Furthermore, server-side software packages have been developed to help users aggregate RDF information for presentation to users [7].

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 important to users of that information: documents, messages, properties, annotations, etc.

WWW 2003, May 20-24, 2003, Budapest, Hungary.
ACM xxx.

3. ADENINE

In a system such as Haystack, a sizeable amount of code—both in agents and in user interface components—is devoted to the creation and manipulation of RDF-encoded metadata. We have developed a language called Adenine that is specifically suited to manipulating RDF through special syntactic and runtime support. Adenine supports standard programming constructs such as for loops and statically-scoped variable bindings, in some respects resembling a version of Python with native support for RDF data types built in.

Like Lisp, Adenine is both a data definition language and an imperative programming language. RDF statements can be described in a syntax similar to Notation3 [1]. Pieces of executable Adenine code are called methods, which are named by URIs and can have RDF properties. Like Lisp functions, Adenine methods are also representable in the data language; the Adenine ontology describes a way to encode the execution of an Adenine method as a series of instruction resources linked by "next instruction" predicates. More information about Adenine can be found on our web site (http://haystack.lcs.mit.edu/) and in previous work [5].

4. AGENTS AND SERVICES

In the past, programs that aggregated data from multiple sources, such as mail merge or customer relationship management software, had to be capable of speaking numerous protocols with different back-ends to generate their results. With a rich corpus of information such as that present in a user's Haystack, the possibility for automation becomes significant because services and agents can now be written against a single unified abstraction. (We will use "service" and "agent" interchangeably in this document, as both are autonomous, running entities that are capable of receiving and sending messages in Haystack.) Furthermore, agents 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.

Agents in Haystack are callable entities that expose methods and maintain state. The core agents are mostly written in Java, but some are written in Adenine and some in Python (these agents are hosted by the Jython interpreter). We utilize an RDF ontology derived from WSDL for describing the interfaces to agents as well as for noting which server processes hosts which agents. As a consequence, we are able to support different protocols for communicating between agents, from simply passing in-process Java objects around to using HTTP-based RPC mechanisms such as HTTP POST and SOAP. In other words, Haystack agents are in effect Web services that implement a specific Java interface and where the appropriate WSDL metadata has been entered into the store; the system takes care of exposing agents via whatever protocols are supported.

5. SLIDE ONTOLOGY

Haystack uses RDF to, among other things, model the user interface. At the bottommost layer is the Slide ontology, which allows

the developer to specify the appearance and formatting of user interface elements such as buttons, paragraphs of text, and tables. Slide is modeled after HTML and, like HTML, is assembled in a tree structure (expressible in RDF since trees are graphs). Adenine is used to hook up executable code to events such as mouse clicks. Individual Slide elements are described as RDF resources and are rendered to the screen by Java components.

There are two key benefits to our RDF representation over HTML. First, we feel that the way in which data that conforms to an RDF schema should be presented is just as important as the schema itself. Slide, in conjunction with Adenine and other user interface concepts discussed later, enables RDF to be used to describe both a schema and its ideal mode of presentation. Second, the Slide ontology is extensible; one needs simply to add an RDF description of a new Slide element to the RDF store in order for the component to be supported. In fact, we have support for writing components completely in Adenine, providing a pure RDF-based solution for distributing presentation logic.

6. VIEWS

Our user interface architecture uses the Slide ontology to present information in terms of *views*. 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 them to be characterized according to the RDF classes they support and by the way they display resources (e.g., full screen, in a one line summary, as an applet, 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 contacts (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; the system 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.

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.

7. 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 that expose pieces of functionality in the user interface. The definition of an operation includes basic information such as its name, an icon, and a set of parameters, which is generated automatically by the Adenine compiler based on the definition of the Adenine method. Parameters are also given names and can have type constraints.

8. CONSTRUCTORS

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) creating the resource; (2) establishing a 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 [3]. Haystack enables this flexibility with constructors—operations that initialize a resource instance in some fashion.

9. ACKNOWLEDGMENTS

This work was supported by the MIT-NTT collaboration, the MIT Oxygen project, a Packard Foundation fellowship, and IBM.

10. REFERENCES

[1] Berners-Lee, T. Primer: *Getting into RDF & Semantic Web using N3*. http://www.w3.org/2000/10/swap/Primer.html.

[2] Eriksson, H., Fergerson, R., Shahar, Y., and Musen, M. Automatic Generation of Ontology Editors. In *Proceedings of the 12th Banff Knowledge Acquisition Workshop, Banff, Alberta, Canada*, 1999.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Boston: Addison Wesley, 1995.

[4] Handschuh, S., Staab, S., and Maedche, A. CREAM—Creating relational metadata with a component-based ontology-driven annotation framework. *Proceedings of K-CAP '01*.

[5] Huynh, D., Karger, D., and Quan, D. "Haystack: A Platform for Creating, Organizing and Visualizing In-formation Using RDF." *Semantic Web Workshop, The Eleventh World Wide Web Conference 2002 (WWW2002)*.

[6] Pietriga, E. IsaViz. http://www.w3.org/2001/11/IsaViz/.

[7] 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 2001*.