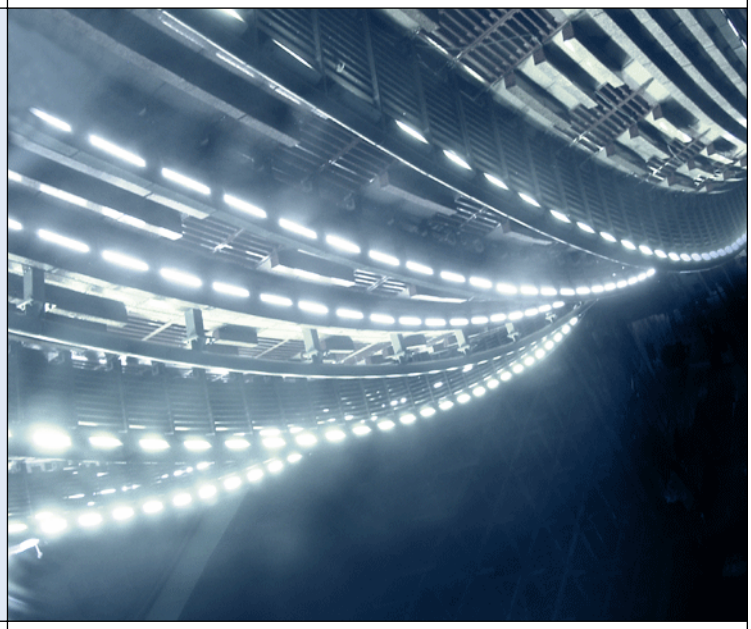
Day

JSR 170 Overview

Standardizing the Content Repository Interface



Roy T. Fielding, Ph.D.
Chief Scientist, Day Software

13 March 2005

Table of Contents

1 Introduction

The World Wide Web is the most pervasive software system ever developed. The Web uses a simple, standardized interface to encompass information from all over the world regardless of how that information is created, stored, and processed behind the interface.[1] As a result, Web-based services can be implemented on any form of computer system, whether or not they are connected to the Internet.

The Web architecture has a simplifying effect that transcends the complexity of traditional network-based software. Unfortunately, those simplifying principles are often ignored when it comes to developing applications behind the Web interface. The .NET and J2EE™ platforms, in particular, emphasize development of software services through the use of complex, service-specific interfaces based on proprietary APIs. Software developers are encouraged to make each interface specific to the object being manipulated, resulting in applications that are fragile when changed and incomprehensible to those tasked with maintaining them. IT departments often find that it is less expensive to develop a new application from scratch than it is would be to adjust their existing applications as user requirements change over time.

Application development doesn't have to mirror the complexity of its applications. The Web has shown that complex goals can be achieved using a very simple interface based on content-centric design and a commitment to standardization. The same architectural principles that made the Web successful can be applied to application development within servers.

A content-centric interface eases the task of application integration by focusing on the uniform nature of content rather than the specific controls of any given application. To illustrate the difference between content-centric and control-centric interfaces, consider the task of integrating with a word processing application. A control-centric approach would be to look at the functionality provided by the command menus of the word processor, such as "Format/Paragraph…," and provide method interfaces that replicate the commands and data-entry dialogs of the word processor. A content-centric approach would be to focus on the data managed by the application: in this case, a sequence of paragraphs with associated formatting.

The control-centric approach is able to take advantage of the unique behavior and functionality built into the word processor, but that comes at great cost: the API will consist of hundreds of actions that are tied to a specific version of one vendor's word processor. In contrast, the data-centric approach limits functionality directly obtainable via the API, but enables an unlimited number of additional tools to be applied to the common data model, eventually surpassing the functionality that could be provided by any single vendor.

The clearest differentiation between the two approaches, however, can be seen when multiple applications are integrated to form a combined application. The interface points of a control-centric architecture grow as an arithmetic progression (*order* n^2), whereas the interface points of a data-centric architecture are strictly linear in growth (*order n*). The application integration task simply does not scale with control-centric architectures.

Following the Web's architectural principles in designing a content-centric interface does not imply we are limited to the protocols and data formats that make up the Web interface (*e.g.*, HTTP and HTML); instead, we can simply learn from the design principles of the Web and its focus on uniform identifiers, standard methods, and extensible representation types.[2] Interactions between Web clients and servers consist of course-grained messages exchanged over high-latency networks. In contrast, application development within a server consists primarily of fine-grain interactions upon local data stores. Therefore, what is needed is a simplifying architecture that promotes content-centric design via a uniform interface, and yet one that is as suitable for tiny data interactions as it is for multi-gigabyte data transfers. We refer to that interface as a Content Repository API.

A content repository is a generic application data "super store." In addition to being adept at handling both small and large-scale data interactions, a content repository is expected to manipulate and store structured and unstructured content, binary and text formats, metadata, and relationships that vary dynamically. Support for advanced content services are also desirable, such as uniform access control, locking, transactions, versioning, observation, and search. In some cases, a content repository will be embedded within the same server as the application, while in others it will be based on separate servers for the sake of availability and load balancing.

2 Advantages of a Standard API

Interface standards clearly benefit application developers, project managers, and the community that supplies them with tools, training, and infrastructure. The software industry has seen this pattern of innovation, standardization, and rapid adoption many times in its history. Consumers of enterprise software find it in their best interests to prefer vendors that produce standards-based architectures, rather than vendors that leave them stranded on a software island.

Application developers can confidently use a standard interface as the basis of their applications without fear of becoming subject to the whims of any single vendor (*i.e.*, vendor lock-in). The interface allows an application to be ported from one implementation of a content repository to another, based on whatever system provides the most value for that application. Furthermore, the standard promotes a shared vocabulary for the developers, making it easier to quickly express design ideas and potential implementations, and eliminating the need to learn dozens of proprietary APIs that are specific to each application.

Software project managers need standards to reduce risk in project planning. Using a common interface promotes reuse of both experience and code, reducing future costs and making it easier to estimate future projects based on past experience. Likewise, adopting technology that can be supported by multiple vendors reduces the risk that a project will become dependent on any single supplier. A content repository standard will make it easier to compare repositories from multiple vendors, achieving a better fit for current applications, while at the same time providing a greater return on investment by not being limited to a single purpose.

Availability of a standard encourages the development of independent documentation, consulting, and training programs. Software development tools with built-in support for the interface soon follow, which in turn promotes use of the interface for additional applications. A multitude of applications creates a market opportunity for infrastructure software development that far exceeds the value of any single application, thus becoming a focus for performance optimization.

Day Software, with over a decade of experience building Web applications and enterprise content management software, developed a uniform interface for content-centric J2SE/J2EE™ application development as part of its Communiqué product platform. However, just defining the interface is not enough. In order to encourage the same social network-effects that enabled the Web to be implemented across so many different platforms, the content repository interface must be standardized.

3 Content Repository API for Java Technology (JCR)

The Content Repository API for Java Technology (JCR) is an ongoing effort to define a standard repository interface for the J2SE/J2EE™ platforms.[3] The goal of a content repository API is to abstract the details of application data storage and retrieval such that many different applications can use the same interface, for multiple purposes, without significant performance degradation. Content services can then be layered on top of that abstraction to enable software reuse and reduce application development time.

A traditional application uses multiple data stores during its operation. For example, a typical email application will store its configuration in a property list, its address book in a table, messages within indexed files (folders), message properties in separate tables, and search indices in a binary hash. In most cases, each of those storage formats would have their own interface. The application developer would spend a significant portion of the development effort designing, creating, and maintaining those interfaces.

In contrast, a content repository API separates the issues of content storage and efficient retrieval. The application developer defines how the content is identified via the interface, writes the content, and then uses the built-in services of the API to perform efficient retrieval in a variety of modes: individual reads, traversals of related data, hierarchical search, and full database query. The real storage format is separated from the application interactions, allowing the most appropriate storage subsystem to be selected based on observing the actual performance of the application, rather than by making a premature decision early in the application's design. The application developer doesn't have to worry about parsing file formats, maintaining search indices for text content, managing transactions, or exporting data between applications; content services like those can be provided by a repository API without being specific to the application.

Designing an API such that it can be independent from both applications and underlying data stores is a challenge. JCR has met that challenge through a generic, hierarchical data model based on extensible node types and content properties, levels of functionality to distinguish

read-only from read/write repositories, and optional functionality for higher-level content services.

3.1 Repository Model

The heart of JCR is its data model for the repository interface, which we refer to as the repository model. The repository model defines how data stored within the repository is identified and structured from the point of view of the client. When the client wishes to perform an operation on the data, it expresses those operations in terms of their effect on the repository model. When the client requests a save operation, or commits a transaction that includes a save, the repository translates the modifications that the client made to its repository model into actual data storage actions corresponding to its storage subsystems.

The repository model consists of an unbounded set of named workspaces, with each workspace containing a virtual hierarchy of items in the form of a tree of nodes and properties. Nodes provide names and structure to the content while properties contain the content. The easiest way to visualize a JCR workspace is through comparison with the Unix file system structure, which consists of a tree of directories and files. However, there are some distinct differences.

- A repository may contain many workspaces, each with its own name and root node.
- Although each workspace is independent in the sense that node hierarchy and content within that workspace are not directly affected by changes in other workspaces, there does exist a correspondence relationship between nodes in different workspaces. Node correspondence enables an application to track changes within other workspaces and perform comparisons, a feature commonly required by collaborative applications involving multiple users.
- A node is typed using namespaced (extensible) names, which allows content to be structured according to standardized constraints. For example, some node types may be similar to a directory, consisting only of a collection of child nodes, while other node types may be closer to a file (*e.g.*, consisting of a set of child properties for the content, a creation date, last-modified date, owner, etc.), and still others may consist of a combination of nodes and properties.
- A node may be versioned through an associated version graph of past changes.

It is important to note that the data model for the interface (the repository model) is rarely the same as the data models used by the repository's underlying storage subsystems. For example, a client may traverse a catalog of items stored for an e-commerce store, make some minor changes to the item descriptions and availability dates, and save those changes as if it were working on individual files in a file system tree. The repository, however, may have as its backing storage a hierarchical database for the item descriptions, a separate relational database for the availability dates, and a backup system for old versions: the client's changes are made persistent by copying the current state to the versioned backup and then performing set operations on the several database records corresponding to the content changed by the client.

The repository knows how to make the client's changes persistent because that is part of the repository configuration, rather than part of the application programming task, and thus the application developer doesn't need to worry about how the data is actually stored and the multitude of potential interfaces for those storage subsystems.

Of course, the complexity and work of building interfaces between the repository and its underlying storage subsystems doesn't just disappear. Instead, it becomes isolated within the repository and thus manageable from the point of view of the customer. The repository implementation can be purchased from an existing vendor, such as Day Software, or constructed as a separate project. With JCR, the customer has the ability to upgrade to more complex forms of repository as the application matures and its actual needs become clear. For that reason, JCR includes two levels of required functionality for a compliant implementation of a repository, along with a separate list of optional content services that may be accessed through the standard API but need not be implemented by every repository.

3.2 Level 1 Functionality

JCR Level 1 provides for the simplest of repositories: those that only support read-only access. This makes it possible for a significant number of basic implementations to be deployed quickly, particularly in those cases where content is stored on a legacy platform and the needs of the application are supplied by read-only access. In addition, Level 1 includes support for export via XML, allowing the content to be migrated to other platforms or to a Level 2 repository when that is eventually desired.

Level 1 includes the following major functionality:

- **Initiating a session with a workspace (login)**. A client connects to the repository by calling a login method with a workspace name and credentials. If the login is successful, the client receives a session tied to the specified workspace that is filtered according to the client's credentials; *i.e.*, the client can access any node or property within the workspace for which their access is authorized by the session login and can determine if it has permission to perform a given action prior to performing it. A repository can define its own access control system or make use of an external mechanism, such as the Java Authentication and Authorization Service (JAAS).[6]
- **Retrieval and traversal of nodes and properties**. Once a session is obtained, a client can retrieve items within the workspace by traversing the tree, by directly accessing a particular node, or by traversal of a set of query results (discussed below). Traversing the tree consists of retrieving the root node, requesting its children, and then traversing its children in turn. Direct retrieval of a node can be accomplished by requesting a path, such as "/customer/acme/Irvine/address", or by requesting the unique identifier (UUID) of a referenceable node. JCR provides methods for access via traversal of parent/child relationships, iteration over a set, hierarchical path, and unique identifier because each method has its advantages and disadvantages, with the "best" form of access depending on the structure of the content and the operations desired by the client.

- **Reading the values of properties**. All content in the repository is ultimately accessed through properties. JCR does not distinguish between "real" data and metadata, though it does allow node types to designate a primary child item that can be indirectly retrieved without knowing its name. Property types define the expected format (and possible conversions) for the content, with built-in types provided for String, Binary, Date, Double, Long, Boolean, Name, Path, and Reference. The latter two supply content indirection via a repository path or UUID.
- **Export to XML**. JCR Level 1 supports two mappings of the JCR data model to XML: the system view and the document view. The system view mapping provides a complete serialization of workspace content to XML without loss of information, meaning that the complete content of a workspace can be exported. The advantage of the system view is that any valid repository content can be expressed in XML. The disadvantage is that the resulting format is somewhat difficult for a human to read. The document view, in contrast, is designed to be more human-readable, though it achieves this at the expense of completeness. The document view's value lies primarily in making XPath queries easier to write and understand.
- **Query facility with XPath syntax**. XPath is a search language originally designed for selecting elements from an XML document.[4] XPath provides a convenient syntax for searching JCR content because the repository model's tree of nodes and properties is analogous to an XML document's tree of elements, element attributes, and element content. XPath query expressions can be defined and executed as if they were being applied to an XML document view of the current workspace, returning a table of property names and content matching the query.
- **Discovery of available node types**. Every node in a JCR repository must have one and only one primary node type. The primary node type defines the names, types and other characteristics of the properties and child nodes that this node is allowed (or required) to have. A node may also have one or more "mixin" types that mandate additional characteristics to those enforced by its primary node type (*e.g.*, more child nodes, properties, and their respective names and types). JCR Level 1 provides methods for discovering the node types of existing nodes and reading the definitions of node types that are available in the repository.
- **Transient namespace remapping**. The name of a node or property may have a prefix, delimited by a single ':' (colon) character, indicating the namespace of the item's name. Namespaces in JCR is patterned after XML namespaces: the prefix refers to a namespace, identified by a URI, which acts as a qualifier to minimize name collisions.[5] Every compliant repository has a namespace registry that maps each namespace prefix to its corresponding URI, including several built-in namespace prefixes that are reserved by JCR. In Level 1 repositories, the prefix assigned to a registered namespace may be temporarily overridden by another prefix within the scope of a particular session.

The goal of JCR Level 1 is to support the needs of read-only applications with a simple repository implementation that remains compliant with the JCR standard. In this way, application developers can remain within a single API even when the needs of the particular application under development do not justify a full-featured repository. For example, read-only applications typically only need a static configuration. Real content management applications require the bidirectional features of Level 2.

3.3 Level 2 Functionality

JCR Level 2 includes the functionality of reading and writing content, importing from other sources (including other JCR repositories), and managing content definition and structuring using extensible node types. In addition to the Level 1 features described above, a Level 2 repository must support the following major features.

- **Adding and removing nodes and properties**. Level 2 includes methods for adding, moving, copying, and removing items within the session's workspace, to be persisted when the client requests a save, as well as methods for moving, copying, and cloning items between workspaces. A client can also compare its unsaved changes to the current workspace state (*e.g.*, to discover changes that may have been saved by some other client) and either merge the current state into its own changes or discard the changes altogether.
- **Writing the values of properties**. Property values, where most of the content (aside from structuring information) is actually stored, can be set in Level 2 via methods on the node and property objects. Property types are checked and the value is either converted to the defined format for that property or an exception is raised if the value is incompatible with the expected format.
- **Import from XML**. Level 2 allows arbitrary XML documents to be imported into the repository as a tree of nodes and properties. If the XML document is in the form of JCR's system view, then the import is the equivalent of a "restore" operation from backup, completing a round-trip from the XML export functionality of Level 1. Otherwise, the XML import is processed as if it were a document view, adding the XML namespace declarations to the repository's namespace registry and building a tree of JCR nodes and properties that match the structure and names within the XML document.
- **Assigning node types to nodes**. Level 2 provides methods for assigning primary and mixin types to nodes. In some cases, type assignment will be made automatically based on the type definition of the node's parent. For example, if a node type requires that all of its children be of primary type "my:customer", then any child created for that node will default to that primary type and any attempt to add a child with some other primary type will result in an exception. Node types can therefore be used to enforce data type constraints on content.

- **Persistent namespace changes**. Level 2 repositories have the capability to add, remove and change the set of namespaces stored in the namespace registry, excluding the built-in namespaces required by JCR.

The goal of Level 2 is to complete the required functionality for a writeable content repository without creating too much of a burden for implementations. As such, it leaves optional many of the features that are provided by modern content management and transaction systems.

3.4 Optional Functionality

JCR provides a standard interface to additional content services as optional functionality. These features are independent of one another and do not depend on Level 2 functionality; thus, each feature may be individually supported by any Level 1 or Level 2 repository.

- **Locking** allows a user to temporarily lock nodes in order to prevent other users from changing them. This function is typically used to reserve access to a node, since JCR Level 2 automatically prevents conflicting updates through its independent workspaces.
- **Transactions** may be supported through adherence to the Java Transaction API (JTA).[7] JTA provides for two general approaches to transactions: container-managed transactions and user-managed transactions. In container-managed transactions, the transaction management is taken care of by the application server and is entirely transparent to the client of the JCR API. In user-managed transactions, the client of the JCR API may choose to control transaction boundaries from within the application. A JCR implementation must support both of these approaches if it provides the transactions feature.
- **Versioning** allows the state of a node to be recorded in such a way that it can later be viewed or restored. The JCR versioning system is modeled after the Workspace Versioning and Configuration Management (WVCM) API defined by JSR 147.[8] A versioning repository has a special version storage area consisting of version histories: a collection of node versions connected to one another by the successor relationship. A new version is added to the version history of a versionable node when one of its workspace instances is checked-in. Each new version is attached to the version history as the successor of one (or more) of the existing versions. The resulting version history is a directed acyclic graph of node state as it has changed over time.
- **Observation** enables applications to register interest in events that describe changes to a workspace and then monitor and respond to those events. The observation mechanism dispatches events when a persistent change is made to the workspace.
- **SQL search** provides an additional query language beyond the XPath search of Level 1.

Optional features allow for a variety of repository implementations while retaining a single API for application development. Application deployment specialists can select repository capacity and feature sets based on the application's needs, rather than some pre-determined level of functionality, thereby reducing costs and minimizing complexity.

3.5 Non-features

As with any well-designed API, the JCR interface only defines what is needed for interoperability between independently developed systems. It does not define how vendors must implement applications that use the repository, including many of the management applications that repository vendors will need to implement to manage the repository itself. Some of these "non-features" include:

- Managing workspace creation, deletion, and naming. JCR does not provide a means for creating a named workspace, changing the name of a workspace, or deleting a workspace. Administrative tasks are usually specific to a repository's implementation and outside the scope of an application interface, for the same reason that file system creation (disk format) is outside the scope of a file API.
- Managing node types. JCR does not supply methods for defining, creating or managing node types. The wide range of approaches used to type entities in existing repositories makes it very difficult to define a single mechanism for node type configuration. Therefore, JCR limits itself to node type assignment and discovery.
- Managing users and access control lists. JCR is designed to make use of existing access control systems rather than invent one that is particular to the repository API. However, we won't be surprised if some access control implementations decide to use JCR for their own content (just like other applications).
- Workflow. Workflow (*a.k.a.*, business process automation) is an application feature commonly found in advanced content management systems. Many of the features supported by JCR (*e.g.*, observation) are particularly useful to workflow engines, but the engine itself and its associated management functions are not part of the JCR interface.
- Semantic information model. JCR does not assume or require anything about the content exchanged via the interface. Applications may create such models and may constrain themselves (and their node types) to obey those models, but JCR is agnostic.

In addition to the non-features noted above, JCR is also frequently mistaken to be a network protocol or a replacement for the likes of HTTP or WebDAV. JCR's interactions take place within the J2SE/J2EE™ environment and, though they could be mapped to an application-layer network protocol, such is not a goal of the repository API. When WebDAV is used as an alternative interface to a JCR-based repository, JCR's role in relation to the WebDAV protocol is similar to the role of the Java™ Servlet interface in relation to HTTP.

4 Deployment

An API can only be innovative if it gets deployed. Day Software has carefully planned and promoted use of the JCR interface through its leadership in the Java Community Process, collaboration with the Apache Software Foundation's open source communities, and development of its own next generation commercial products based on JCR.

4.1 JSR 170 and the Java Community Process

Industry standards for the Java™ language and J2SE/J2EE™ platforms are created within the Java Community Process (http://jcp.org/). Members of the JCP participate in the proposal and development of community specifications, referred to as Java Specification Requests (JSRs). In February 2002, Day Software proposed the creation of an expert group to define a standard content repository API. The proposal was accepted, with JSR 170 formed as a result and David Nüscheler (Day Software's CTO) appointed as specification lead. The expert group has included members from 21 different companies.

The JSR 170 expert group are now in the final stages of producing a first version of the Content Repository API for Java™ Technology, having published a proposed final draft for public review and anticipating only minor changes before submission for final approval.

The JCR specification is a collaborative product of the specification lead, David Nüscheler (Day Software), the author, Peeter Piegaze (Day Software), and other members of the JSR 170 expert group, including Tim Anderson (Intalio), Gordon Bell (Hummingbird), Geoff Clemm (IBM), David Choy (IBM), Jeff Collins (Vignette), Stefan Guggisberg (Day Software), Stefano Mazzocchi (Apache Software Foundation), James Myers (Pacific Northwest National Laboratories), James Owen (BEA), Franz Pfeifroth (Fujitsu), Corprew Reed (FileNet), Victor Spivak (Documentum), David B. Victor (IBM), as well as many others who contributed with corrections and suggestions.

4.2 Open Source Collaboration

As specification lead, Day Software is responsible for licensing the JCR specification, its reference implementation (RI), and technology compatibility kit (TCK) in a form that allows other organizations to create independent implementations of the standard once it has been accepted. In order to promote deployment and use of the interface, Day Software decided early in the process to license the specifications under the most liberal terms possible, such that open source projects can implement to the specification without concern.

In order to further promote application development and ensure that the JCR reference implementation and TCK receive the open review, testing, and collaboration that only a true open source community can provide, Day Software licensed the initial RI and TCK implementations to The Apache Software Foundation (http://www.apache.org/) for use in the Jackrabbit project.

- http://incubator.apache.org/projects/jackrabbit.html

Jackrabbit is currently under incubation and will most likely remain so until the final JCR specification is released and the JCP's constraints on independent implementations are lifted. Several Apache veterans have been members of the JCR expert group since its inception and many other Apache projects are investigating use of JCR as their primary content repository interface. Day Software continues to participate in the Jackrabbit project and will use the code within the official (binary) RI and TCK releases, thereby allowing developers to beta test against the open source versions as well as the official versions.

4.3 Commercial Products

JCR's design is based on experience gained from Day Software's *Communiqué* family of products for enterprise content management. Future versions of *Communiqué* will continue that leadership through its combination of the first full-featured implementation of a JCR-based repository along with the complete suite of applications and tools needed for enterprise content management.

In addition, Day Software has produced a standalone release of its JCR implementation, *Content Repository Extreme (CRX)*, designed for embedded applications and OEM licensing. *CRX* allows for the storage, retrieval and management of content across large-scale enterprises, while protecting technology investment in content and applications, thus providing a content infrastructure that is future-proof and sustainable.

In addition to the complete feature set of the JCR API, *CRX* provides a number of useful tools beyond the specification to help create and maintain repositories.

- User tools that enable easy browsing of the repository (*Content Explorer*), import of content in original form or with automated decomposition into hierarchical nodes (*Content Loader*), and export of content into XML or ZIP format (*Content Zipper*) for dissemination of content to remote systems and transfers between repository implementations.
- Administrative tools for user, node type, and namespace management, as well as cluster management for high availability installations.
- Integration with existing Internet standards, such as remote access via WebDAV and external user authentication via LDAP.

Other commercial implementations, tools, and applications based upon JCR are anticipated once the specification has been formally approved.

5 Conclusions

The Content Repository API for Java™ Technology (JCR) is poised to revolutionize the development of J2SE/J2EE™ applications in the same way that the Web has revolutionized the development of network-based applications. JCR's interface designers have followed the guiding principles of the Web to simplify the interactions between an application and its content repository, thus replacing many application-specific or storage-specific interfaces with a single, generic API for content repository manipulation.

JCR is a boon for application developers. Its multipurpose nature and agnostic content model encourages reuse of the same code for many different applications, reducing both the effort spent on development per application and the number of interfaces that must be learned along the way. Its clean separation between content manipulation and storage management allows the repository implementation to be chosen based on the actual performance characteristics of the application rather than some potential characteristics that were imagined early in the application

design. JCR enables developers to build full-featured applications based on open source implementations of a repository while maintaining compatibility with the proprietary repositories that are the mainstay of large data centers.

Application owners will gain control over their investment in repository infrastructure by leveraging the market created by an industry standard interface. Purchasing managers can compare repository implementations on the basis of tested performance and standard feature sets, rather than artificial benchmarks. And, when their needs eventually change, the content can be migrated from one repository to another, using a standard XML format, without changing any of the applications that have been built to create, manage, and manipulate that content.

Most significantly, however, JCR holds great promise for eliminating the information silos created by application-specific storage mechanisms and proprietary APIs. Content management systems based on the JCR interface, such as Day Software's Communiqué, will be able to manage all of the content within all of the applications that make use of the repository, thus unifying information access, workflow, and presentation across the entire enterprise.

References

[1] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

[2] W3C Technical Architecture Group. *Architecture of the World Wide Web, Volume One*. W3C Recommendation, 15 December 2004. <http://www.w3.org/TR/webarch/>

[3] Day Software. *Content Repository API for Java™ Technology Specification*. Java Specification Request 170, version 0.16.1, 24 December 2004. <http://www.jcp.org/en/jsr/detail?id=170>

[4] DeRose, S. and J. Clark, eds. *XML Path Language (XPath), Version 1.0*. W3C Recommendation, 16 November 1999. <http://www.w3.org/TR/xpath>

[5] Bray, T., Hollander, D., and A. Layman, eds. *Namespaces in XML*. W3C Recommendation, 14 January 1999. <http://www.w3.org/TR/REC-xml-names>

[6] Sun Microsystems. *Java Authentication and Authorization Service (JAAS)*. <http://java.sun.com/products/jaas/>

[7] Sun Microsystems. *Java Transaction API (JTA) Specification*. <http://java.sun.com/products/jta/index.html>

[8] IBM. *Workspace Versioning and Configuration Management (WVCM) API*. Java Specification Request 147. <http://www.jcp.org/en/jsr/detail?id=147>

Author

Roy T. Fielding is chief scientist at Day Software. Dr. Fielding is best known for his work in developing and defining the modern World Wide Web infrastructure as architect of the current Hypertext Transfer Protocol (HTTP/1.1), co-author of the Internet standards for HTTP and Uniform Resource Identifiers (URI), co-founder the Apache HTTP Server Project, and former chairman of the Apache Software Foundation. His research interests include the World Wide Web, software architecture for network-based applications, application-layer network protocols, collaborative software development methods, and global software engineering environments. His dissertation, *Architectural Styles and the Design of Network-based Software Architectures*, defines the REST architectural style as a model for the design principles behind the modern Web architecture. Dr. Fielding was honored with the 1999 ACM Software System Award—the computing society's most prestigious award for software—for his work on the Apache HTTP server project. He has also been honored by MIT Technology Review as a member of the first TR100 (the top 100 young innovators for 1999) and by the O'Reilly Open Source 2000 with the Appaloosa Award for Vision. He continues to serve as a member of the Apache Software Foundation, an elected member of the W3C Technical Architecture Group, and an external advisor for the University of California's Institute for Software Research. Dr. Fielding received his Ph.D. in Information and Computer Science from the University of California, Irvine.

Day Software is a leading provider of integrated content, portal and digital asset management software. Day's technology Communiqué offers a comprehensive, rapidly deployable framework to unify and manage all digital business data, systems, applications and processes through the web. Communiqué's content-centric architecture, and its innovative ContentBus, turns the entire business into a virtual repository, bringing together content from any system, regardless of location, language or platform.

Day is an international company, founded in 1993, and listed on the SWX Swiss Exchange (SWX: DAYN) since April 2000. Day's customers are some of the largest global corporations and include Audi, DaimlerChrysler, Deutsche Post World Net, General Electric, Intercontinental Hotels Group, McDonald's, UBS and Volkswagen.

For more information

Day Software AG
Barfüsserplatz 6
4001 Basel, Switzerland
T +41 61 226 55 85

E-Mail info@day.com