IBM's Jazz Integration Architecture: Building a Tools Integration Architecture and Community Inspired by the Web

Scott Rich
Distinguished Engineer, IBM
4205 S. Miami Blvd.
Durham, NC 27703
srich@us.ibm.com

ABSTRACT

In this paper, we describe our experience in the Jazz project, beginning from a "Classical" repository- and Java-centric design and evolving towards an architecture which borrows heavily from the architecture of the Web. Along the way, we formed an open community to collaborate on adapting this architecture to various tool domains. Finally, we discuss our experience delivering the first generation of tools built and integrated using these techniques.

Categories and Subject Descriptors

D.2.12 [**Software**]: Interoperability

H.3.5 [**Information Systems**]: Information Storage and Retrieval – *On-line Information Systems: Web-based services*

K.6.3 [**Computing Milleux**]: Management of Computing and Information Systems – *Software Management: software development*

General Terms

Management, Performance, Design, Standardization

Keywords

Representational State Transfer, REST, Software Development, Semantic Web.

1.Introducing the Jazz Project

In 2005, IBM launched the "Jazz" project, with a mission of building new generation of software development tools and also producing a future tools platform. The platform was built on standard middleware, and intended to enable new tools to be built with common collaborative services.

Starting from their experience with the Eclipse and WebSphere projects, the Jazz developers built their platform upon the same technologies: java and OSGi. In order to provide productive development of new tools on the platform, Jazz offered an easy way to build new Jazz services and clients. Java libraries provided "free" marshaling of compliant Java interfaces, leveraging the EMF modeling framework. The server exposed a set of services, derived form Java service interfaces, and compatible client interfaces were offered in Java.

This architecture was very productive and powerful for the Jazz developers, and enabled the first round of platform and tool development to be done relatively quickly.

2.New Clients Arrive on the Scene

In 2006, the Jazz project started to look beyond Java clients and think about Web-based tools. The project adopted an AJAX-style for its browser clients, and began to provide some services to enable those clients. Among those services was a "modeled REST" service capabiliy. This allowed a service to be built in Java which was document-oriented, in that it conformed to a basic pattern of getting out putting a structured document, but these document shapes were still derived from Java interfaces. The advantage to Web clients was that the modeled REST services could provide a JSON marshaling of their service in addition to XML.

This pattern proved productive, and the Jazz Web UI's were developed in the classic SOA style. The document structure was agreed, the interface specified, and the UI and service developers could work in parallel until integration. UI developers tested on mock JSON data while the real service was being implemented.

Around this time, the Jazz project started to receive some positive feedback on this document-centric model. Several spontaneous integrations appeared from teams that did not have direct support from Jazz developers. In each case, the story was the same, they had used Firebug to trace conversations between one of the Jazz Web UI's and the server, and had written a client to emulate this conversation. These clients were appearing in browsers, in new Eclipse clients, and even in Visual Studio environments, emphasizing the importance of a non-Java client story.

3.Traditional Data Models Begin to Breakdown

While the client story was evolving to support a broader set of client technologies, similar learning was happening with the Jazz data architecture. The original Jazz data architecture was a fairly classical extensible repository design. OSGi bundles could be installed into the server to declare data models for new tools. The server took care of creating the necessary tables and translating resource access and queries into relational queries. This system

WWW 2010, April 20–24, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

worked well, for Java tools, and provided great performance and scalability, but it was not flexible enough for may tool domains. As new tools adopted the Jazz platform, they all had requirements for customization and extensibility of their core data model. Work item customization, custom requirements structure, and extensible test plan formats are examples of the immediate requirements. The Jazz repository offered a limited extensibility architecture, where additional properties could be associated with a modeled resource, but it was never designed to scale to extreme levels, and the complexity of generating the queries for extension values meant that there were limitations on the capability which could be offered for extensions versus modeled data.

4.Integrations Exceed Implementations by 10x

While the initial focus of the Jazz platform was upon providing implementation technologies for a new set of tools being built from scratch, there was also a goal of providing integration capabilities for existing tools. Over time, the Jazz team came to realize that the number of new tools to be built would always be dwarfed by the number of existing tools. The integration aspects of the architecture started to get more and more attention.

The arrival of the Telelogic tools into Rational in 2008 drove the point home. The new Jazz tools now had a completely new set of tools within Rational they had to integrate with. These tools were built on a variety of technologies, but had to be integrated into Jazz in a first class way.

5.Adopting the Architecture of the Web

As it became clear that integrations, rather than implementations, had to be the focus of Jazz enablement, the Jazz team began to look for inspiration from existing integration architectures. Martin Nally[3] led an investigation of the architecture of the Web as a candidate. Clearly the Web supported massive scaling, was extensible in many ways, and provided a powerful model for linked data.

The Jazz team applied the architecture of the Web to reshape the Jazz integration architecture. Tools were tasked with providing fundamental Web capabilities for their tool data:

- Tools should provide stable URLs for important resources
- Tools should publish resource formats, and maintain compatibility going forward
- Tools should Publish specifications for common services, like query
- Tools should enable OAuth for authorizing tool-to-tool communications

6.Going Open

The Jazz team was encouraged by their early experience applying this architecture, and decided that there was even greater value in an external adoption of these techniques. To this end, the Open Services for Lifecycle Collaboration project was chartered at open-services.net. The goal of this project was to form working groups to apply these ideas to different tool domains, producing integration specifications that could be provided and consumed by tools across the industry.

Change Management was selected as the pilot domain. Rational had a need for an integration specification in this area to enable integrating our new and existing change tracking systems with quality and requirements management. Working with partners and customers, a specification was produced for a RESTful Change Management service provider.

One of the daunting problems of any integration specification is agreeing on a common definition of a resource format. Many previous efforts at standardizing software development artifacts have struggled to achieve this. In the case of the OSLC Change Management specification, there was a breakthrough which enabled this specification to reach agreement very quickly, and deliver a very compact resource definition.

The Change Management workgroup realized that trying to completely specify one definition for a Change Request, for example, was pointless. All of the participating tools supported extensive customization, there was no single resource structure, even within one tool. Rather than try to describe a metamodel for capturing custom change request definitions, the team decided to rely on the implementing tool to implement awareness of its own customization model, as well as its model of permissions and process. This would be achieved by specifying services which allowed an integrating tool to request a delegated user interface for presenting a CM resource, or picking or creating a new resource.

This breakthrough allowed the CM specification for a Change Request[4] to be expressed on a single page! Only the identifier and title are required of all change request resources. The spec allows any integrating tool to be guaranteed enough information to create what is essentially a rich hyperlink, but no more. Beyond that, the specification teaches a tool how to rely on the providing tool for its full change management capability.

These concepts that allowed the Change Management workgroup to achieve quick success have since been adopted by the working groups in many other OSLC domains: Requirements Management, Quality Management, and Architecture Management.

7.Indexing Opaque Resources as RDF

Tools building on the Jazz platform still needed help dealing with dynamic and customizable resources which they were storing. The Jazz repository was enhanced to offer indexing of resources in XML, proprietary document formats, and even images, producing queriable RDF data. Tools continued to store their resources in their native format, although some started adopting RDF itself to have a consistent resource and query model. The Jazz platform's query service provided SPARQL[2] query execution over the index graphs. Tools can choose to expose SPARQL query natively in their APIs, but most provide a simplified query service using query parameters or a POST query syntax to support the most common query use cases.

The RDF approach to index data is proving to be a good fit for extensible and customizable resources. The Jazz repository does not need to be extended in order to support new tools and resource types, or to allow query over a new user-defined custom resource format. Tools can perform their own indexing of non-RDF resources, or they can provide hints to the platform to enable indexing of XML data.

8.Performance observations

As always, performance and scalability of the repository was a key concern as the Jazz team adopted RDF and SPARQL. The previous relational implementation for custom data was performing adequately, but had many limitations.

Initially, the platform adopted XQuery as a query language over indexed XML data. XQuery was powerful but proved very hard to optimize. Tool developers struggled to code queries carefully to avoid queries which performed very poorly at large scales.

The Jazz team began looking at RDF query solutions. They first experimented with relational-backed RDF query implementations, benchmarking some IBM internal technology and the Jena SDB implementation. These solutions showed reasonable query performance at small scales, but struggled to perform queries and bulk loads at large scales(tens of millions of triples).

Finally, the team benchmarked Jena's TDB, a file-based implementation. This approach finally yielded the right balance of query responsiveness at low scale and did not degrade dramatically when the triple store grew into the millions. Jena TDB was adopted in 2009 as the RDF store and SPARQL provider for the Jazz platform.

9.Shipping our First Generation of Web-oriented Tools

In 2009, we delivered our first round of tools built on and integrated with these technologies

A new requirements definition tool was built from scratch and using all of the integration and implementation techniques provided by Jazz. One new and one existing change management tool were integrated as defect providers. And finally a new test tracking system was integrated to provide integration with test data.

Using the integration architecture, the teams were able to create traceability links between resources in the requirements, testing, and development domains. The integration is primarily among Web-based tools, but rich clients can also participate by hosting Web UIs as appropriate. The hyperlinks between the resources enable users to navigate the web of resources, blurring the lines between tools. The links allow queries to be executed which cross domains, to answer important integration questions such as "What are the requirements with failing tests?", and "What are the defects blocking testcases?".

10.Conclusions

The experience of building the Jazz platform and integration architecture has been an educational one. The architecture which we initially defined for new tool implementations turned out to be powerful, but incomplete without a flexible and open integration architecture. Adopting the principles of the World Wide Web: RESTful resource style, semantic data in RDF and SPARQL query, has allowed us to build powerful integrations between our tools and others. Taking these ideas to the Open Services community has allowed us to expand the domains and tools available to integrate in this style. And finally, applying these concepts to our own integrations has allowed us to produce a set of tools which provide richer traceability across the application lifecycle.

11.REFERENCES

[1] Roy T. Fielding , Richard N. Taylor, Principled design of the modern Web architecture, ACM Transactions on Internet Technology (TOIT), v.2 n.2, p.115-150, May 2002 [doi>10.1145/514183.514185]

[2] E. Prud'hommeaux, A. Seaborne (Eds.), SPARQL Query Language for RDF, W3C Recommendation, 15 January 2008.

[3] M.Nally, Michael O'Connor, "Martin Nally on Jazz, integration, SOA"(podcast), June 2008, http://www.ibm.com/developerworks/podcast/rsdc/nally-060308txt.html

[4] J. Wiegand, "The Case for Open Services", May 2009, http://open-services.net/html/case4oslc.pdf

[5] OSLC CM workgroup, "Change Management Specification V1: Change Management Resources Definition", May 2009, http://open-services.net/bin/view/Main/CmResourceDefinitionsV1