Sonatype

The Benefits of a Repository Manager

Introduction

This paper outlines how the use of a repository manager enables development teams to reduce build times, improve control, and increase collaboration. In a world where 80% of a typical application is now assembled from components, repository managers have become a "must have" technology for modern software development organizations.

Repository managers are a foundational step in a broader trend towards component lifecycle management (CLM). CLM generally, and repository managers specifically, help organizations gain control and improve collaboration in agile, component-based software development.

The key primary benefits of a repository manager include:

- Reduces build times by proxying cloud-based repositories and enabling local access to components.
- Improves collaboration by providing a central location to store and manage common components used among developers and teams.
- Improves control by providing a mechanism to observe and control component usage.

This paper provides a detailed argument for the use of a repository manager, and its benefits across the software development lifecycle.

When you don't use a Repository Manager

Here are some common negative behaviors when you don't use a repository manager:

- All of your developers download components directly from public repositories. New developers spend an hour downloading a massive library of dependencies from the Central Repository. Proprietary or vendor libraries are passed around, from developer to developer. If you don't use a repository manager, developers likely pass JARs around as an email attachment with some ad-hoc instructions.
- The source control repository is used to store everything from source code to binary builds. Because there is no repository designed to store binaries, developers start to use tools like Subversion to keep track of binaries. As time passes, the Subversion repository becomes an ad-hoc file system for files not appropriate for in a software configuration management (SCM).
- The continuous integration server depends on public repositories. When you change your build or add a new dependency, your CI system downloads dependencies from the public repo. It depends on the availability of this public resource to run builds.
- Production deployments have to run the entire build, from start to finish, to generate binaries for deployment. When a build is tested and then ultimately pushed to production, the build and deployment scripts checkout source code, run the build, and deploy the resulting binaries to production systems.
- Since there is no established mechanism for publishing source or binary artifacts, sharing source code with external partners means granting them access to your SCM.

The general theme in all of these behaviors is that either your systems depend on public resources, or they all depend on the SCM system as a central collaboration point. Using a repository manager provides an optimal solution for managing components without the shortcomings of other approaches.

Caching and Collaborating

Caching artifacts locally

The first, and most obvious, benefit is that a repository manager will proxy and cache artifacts from a remote repository. Downloading your dependencies takes much less time and your builds don't rely on Internet access. Instead of waiting for artifacts to be downloaded over the public Internet, your build will download artifacts from a local server. Since you avoid the public Internet, this process is sped up significantly. The build that once took 15 minutes to download dependencies will now take less than a minute. A repository manager also insulates you from network failures or brown-outs by placing required components behind your firewall.

Get those JARs out of Subversion

The Oracle JDBC driver, a proprietary JAR from a vendor, is the kind of one-off, third party library that is not going to be made available on a public repository. Without a repository, the most obvious solution is to just check these JARs into source control and store them right next to your code. When you run your build, branch your project, and release your code you are just passing around these binaries as a part of your project. While this approach may work for very limited use cases, you are overloading the source control system to perform a task it wasn't designed for.

While Subversion, and many other modern SCM tools, will gladly version binaries, you are storing a static, unchanging artifact in source control. Every time you checkout out source control, you are checking out binaries, and if you are unlucky enough to work somewhere that stores all libraries in source control, you will often be asked to check out a large repository of JARs. Your SCM is going to carefully keep track of changes and version files that will never change.

This is both inefficient and unnecessary. If you use a repository manager, you can upload third party libraries to a repository that stores artifacts on the repository manager. Then you can mix these third party libraries into a public repository group that will combine the contents of public repositories with your own repositories. In other words, you can declare a dependency on a custom, manually uploaded third party JAR as easily as you would declare a dependency on a library stored in the Central Repository.

Collaboration

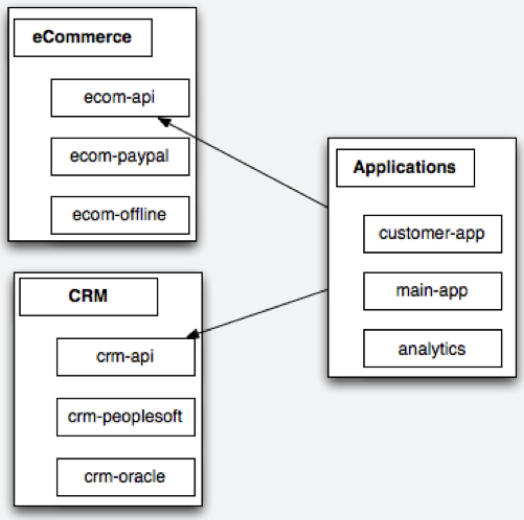
Collaboration is best illustrated with a hypothetical organization. Imagine you work at an organization with three development groups of about 30 developers.

There is an eCommerce group, which is responsible for writing systems that interact with banks and service providers like PayPal.

There is a customer relationship management (CRM) group, which has the unenviable job of merging two disparate CRM solutions from Peoplesoft and Oracle and wrapping those systems in an easy to use Java API.

Then there is a third group, the web applications group, which has the primary responsibility for wrapping these two back office systems in a slick web interface.

These relationships are captured in the following diagram:



Imagine that the eCommerce group and the CRM group have to assemble the entire external API into a single project: ecommerce-api and crm-api. These APIs contain all of the logic required to connect to a set of internal services hosted by each group. As the eCommerce group and CRM groups innovate and offer new services, they will cut releases of these new APIs and publish those releases to the repository manager.

When the web applications group is ready to start using a new version of the eCommerce or CRM API they can simply define a dependency on a version of this artifact. The build for the web applications group will then download the appropriate version of the eCommerce of CRM API from the corporate repository manager.

In this way, the repository manager is a central collaboration point for different groups within the same company. Without the use of a repository manager, you will often see disparate groups forced to check each other's code out of SCM and build it from scratch just to generate client libraries. In the organization that has embraced repository management, these libraries are published as binaries for clients (in this case other workgroups) to consume from the repository manager.

When you collaborate using the SCM, it is very difficult to scale. Every workgroup is forced to use the same set of tools for build management and you will very often see the entire IT operation having to synchronize releases. When you share build artifacts using the repository manager, you decouple workgroups from one another and allow your internal teams to collaborate using a more adaptive, open-source model.

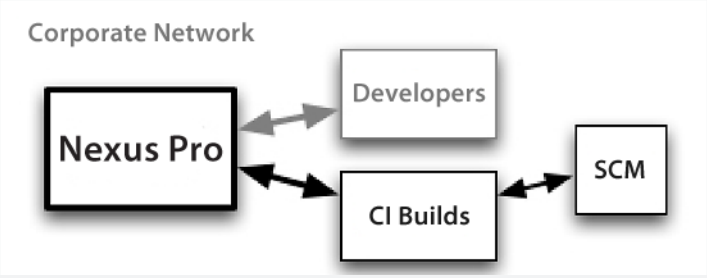
If the eCommerce group needs to innovate, they can do so without affecting the source code of the web applications group. They can publish new releases to the repository manager, and the web applications group can decide when and how they are going to consume these releases by changing the version of a dependency in a build.

When you collaborate using the SCM, all of your workgroups are joined at the hip by a common codebase. When you use a repository manager you allow different workgroups to innovate and create at their own pace.

Continuous Build Deployment

Next we look at how a repository manager works in concert with a continuous integration server like Hudson, Jenkins or Bamboo.

A CI server is an established fact of modern development infrastructure. It is a server that waits and watches, keeping a vigilant eye on your source control system and jumps into action every time it sees a code change. When code changes, your CI system is usually configured to run the entire build, execute all of your unit and integration tests, and send out an email to every developer if it identifies a defect or a failed test. It does this so that you will have an easier time identifying where a particular problem was introduced to the source code.



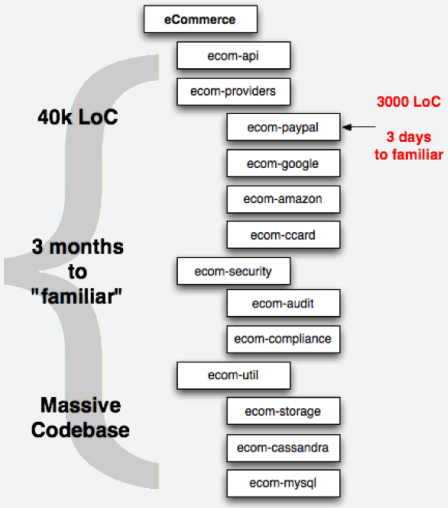
If John checks in some bad code, the CI system runs the build immediately, and about 30 minutes later, everyone in the group receives an email with the subject header "John just broke the build". It is a great way to identify errors, and it is also a great way to motivate developers to test locally before committing to a source control system, as no one likes to be the reason for a build failure email.

Running a CI server is more than "just a good idea". Once your system reaches a certain level of complexity you can't scale a system without committing to continuous integration and testing. If you don't have continuous integration, you end up having to put all development on hold each time you want to perform a release. If you don't build, test, and deploy your system on a regular basis integration becomes a time consuming nightmare. This is especially true if your development effort spans multiple systems and multiple development workgroups. You run a CI system because building, testing, and deploying your system should be automatic: it should be as trivial as pressing a button.

The concept of a CI server is only slightly more established than a repository manager, and very often you will see that an organization has identified the need for a CI server before they've identified the need for a repository manager. If you are coding a complex system, there is a very good chance that you are already running a CI server. The most popular servers out there are Hudson, Jenkins, Bamboo, and CruiseControl. While the connection between CI servers and repository managers might not be immediately obvious, when used together they can introduce some new possibilities for the way you develop your systems.

Continuous Publishing

When you have a system to continuously build your code, you also have a system that can continuously publish SNAPSHOT artifacts to a repository manager to enable a more granular approach to development.

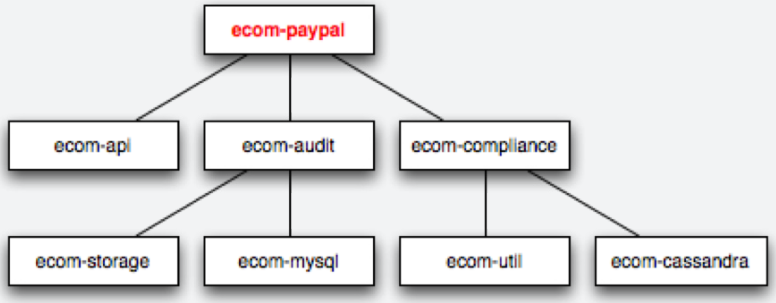


Assume you have a new programmer starting tomorrow. Instead of throwing him at the entire 40,000 lines of code, you would like to be able to give that developer a small, easy-to-digest task.

You want this developer to add support for PayPal's Adaptive Payments API in your eCommerce system. That's it. You don't want them to be distracted by the overwhelming scope of the project, and you certainly can't afford for them to take a three-month voyage through your project's code before they start contributing to the effort.

Deadlines are tight, and you don't have enough people on your team. It is important that new hires start programming as soon as they walk in the door.

In the case of the ecom-paypal project, assume that the dependency graph looks like this:



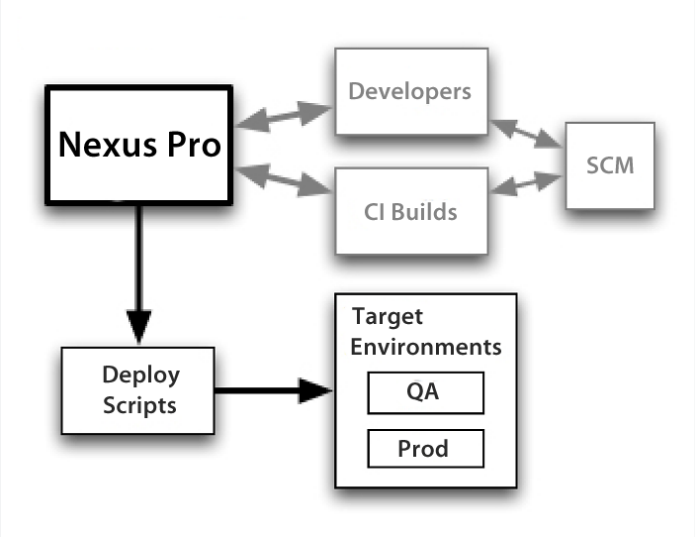
Without a repository manager, trying to build a new version of ecom-paypal in isolation is going to generate errors because you are forced to check-out the entire codebase to build and install all of the dependencies.

When you have a repository manager and a CI server, you can configure your CI server to publish SNAPSHOT artifacts (in-progress SNAPSHOT binaries) to your repository manager. This will allow you to just check out a single, isolated portion of a much larger multi-module project. Now, when you run the ecom-paypal module's build in isolation, Maven will just download and use the most recent SNAPSHOT.

With a repository manager you can work on specific components of a larger multi-module project. This ability to divide and conquer your codebase comes in very handy when you need a consultant to take a look at a specific problem, or when you need to look at a coding problem in isolation. When you continuously publish build artifacts to a repository manager, you move away from the single monolithic project build and toward a project layout and architecture that lends itself to modularization.

Deployment

The following diagram shows the process of pushing code to production and automating a deployment:



Using a hypothetical example, assume that your organization runs a fairly complex application that consists of a few back office systems, a CRM system, and a web application. When your company needs to deploy one of these systems to production, the result is a highly orchestrated symphony of activity. Servers need to be taken offline in a coordinated manner across a distributed network of machines. Downtime needs to be planned and communicated to the right individuals, and backups need to be created before any new code is pushed to a production system.

In other words, any time you need to push new code to production, it is a real challenge to make sure that everyone is coordinating in just the right way. A few days before a production deployment, the development team freezes a particular branch and tags a release. This release is then installed in a QA environment. The QA team tests the release candidate, and if all goes well, this release is then deployed to production.

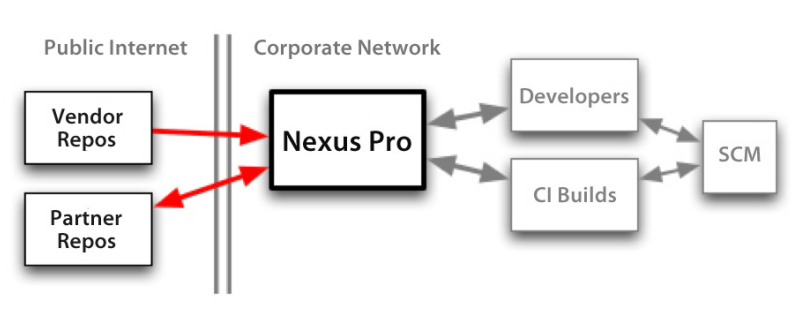
Without a repository manager, your operations teams has to know how to checkout a release tag from SCM, they have to run the entire build to generate a few binary artifacts, and then they need to write some scripts to automate a deployment to production.

With a repository manager, you can publish a release candidate binary to a hosted repository. This release candidate binary can then be used by a series of deployment scripts (or a tool like ControlTier or Puppet). You can have a deployment script that will publish the deployment to a QA environment, and then you can have a deployment script that takes the same, certified binary and publishes it to production.

The main difference between these two approaches is that the second approach adds some isolation between the SCM system and the deployment script. If your production deployment depends directly on your build, you have to recompile and repackage your entire system every time you do a deployment. This is often a challenge because the people that run the build system for deployment are seldom the same engineers that develop your system. If your operations team has to check out source code from an SCM just to build a production network, you are introducing some risk into your process.

Instead, use the repository manager as a way to share binaries with operations. Instead of certifying tags in an SCM, your QA testers should be certifying binaries. This way you can be certain that the systems you are deploying to production have been tested.

External Partners and Vendors



While the Central Repository contains a huge number of libraries, there are still some pretty important things missing.

Anything covered by a non-open-source license, like the Oracle JDBC drivers or anything proprietary, isn't going to be made available from the the Central Repository. Very often there are going to be commercial libraries that you will need to download and install in a third party repository.

While this works, it requires some manual work to upload the artifact.

Vendors and Repository Management

When your vendors embrace repository management, there's no reason why you would have to manually install any third party JARs in your corporate repository. Instead, vendors who need to provide binaries to customers would provide them via authenticated public repositories.

These vendor-run repositories, protected by authentication credentials, would allow customers to synchronize local, corporate repository managers with commercial software vendors. This would allow vendors to capitalize on a new "always connected" model for software delivery.

Both the customer and the vendor have fewer moving parts to worry about, and the task of delivering software becomes very simple.

Once the customer sets up an authenticated proxy repository to connect to a vendor's repository, the vendor can deliver new updates and software directly to a customer's repository.

Instead of having to send the customer a packaged set of binaries, the vendor can simply tell the customer to update a version number in a dependency. The build will then interrogate the customer's repository and then automatically download the repository from the vendor.

The vendor receives additional benefits from this setup:

They can keep track of which customers requested which artifacts, they can control which artifacts a customer has access to, and they can use the repository manager as a way to enforce licensing or provide metered, on-demand access to software components.

Partners and Repository Management

In today's increasingly connected world, it is very common for one company to provide an API for partners. Sites like Twitter and Google provide APIs for the general public, and other companies provide partner-specific APIs for collaboration. If you provide an API for a partner company, you can expose libraries and interfaces using a public-facing, authenticated repository.

Suppose that you work at a financial institution that needs to partner with another financial institution. The two institutions have agreed to collaborate with one another using a simple set of REST services. While these REST services are very well documented, you also want to make sure that your partner is invoking these REST services according to a set of standards. To make it easier to consume these services, you've created a simple client library for your partner. To make it even easier for your partners to use this library, you've published versions of this library to a public-facing, authenticated repository. Using the security features of a modern repository manager, you can create partner-specific library versions and limit access to just the intended recipients. You can also maintain detailed audits of when a particular API was delivered to a particular partner.

As more and more organizations start to adopt the best practice of running a repository manager, these same organizations will be creating ad-hoc, private relationships to emulate the ease of the Central Repository for a corporate environment. Partners, vendors, and consortia of companies will create repositories to facilitate code sharing and cooperation.

Sonatype Nexus Pro and Sonatype CLM

Sonatype Nexus Professional (Nexus Pro) is the industry's most widely adopted repository manager. More than 20,000 organizations and hundreds of thousands of developers rely on Nexus Pro every day to improve their software development process.

Sonatype Nexus Pro is a core foundation of a component lifecycle management strategy. Nexus Pro provides increased control of components in the repository and enables effective collaboration among teams. Many organizations wish to extend component management across the entire software development lifecycle. Sonatype CLM builds on Nexus Pro to enable visibility and control from design, through development and build, an in to production.

For more information about Sonatype Nexus Pro or Sonatype CLM, please visit www.sonatype.com

Sonatype

12501 Prosperity Drive • Suite 350
Silver Spring, MD 20904
1.877.866.2836 • www.sonatype.com