Continuous Delivery of Apache Sling Applications

Master's Thesis

by

Artyom Stetsenko

Presented for the Degree of
Master of Science
in
Computer Science

Supervisors:

Prof. Willy Zwaenepoel
Operating Systems Laboratory
École Polytechnique Fédérale de Lausanne

Bertrand Delacrétaz
Principal Scientist
Adobe Research Switzerland

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland

August 15, 2014

Abstract

Every software product naturally gets more and more complex over time. In order to simplify further development, common tasks, such as building and testing, are often automated. One of such automatable aspects of software development is release, with the practice of automated releases known as *continuous delivery*. Many well-known applications, such as Chrome or Microsoft Office, have shifted to this paradigm in the recent years. The purpose of this thesis was to propose and evaluate a continuous delivery mechanism for web applications built on the Apache Sling framework. Specifically, the goals were to develop a mechanism for updating Sling applications atomically and without downtime, and to automate it. The result was a mechanism that achieves both goals, where each update is triggered by a commit of a single text file describing the new application version to a version control system such as Git. Following the commit, the new application version becomes accessible to users within the time it would take to start it from scratch, usually on the order of a few minutes. The different application versions have to be partially interoperable, limiting the proposed mechanism. The mechanism additionally contains single points of failure. Further research and refinements of the mechanism are recommended to lift these limitations.

Acknowledgements

Because this thesis was carried out as an internship at Adobe, it was supervised not by one, but by two sides. It would not have been possible without the help and support from people both from the side of the university and from the side of the company.

I would like to thank Professor Willy Zwaenepoel for accepting to supervise this thesis from the university side, and for providing continuous guidance during the past six months, making suggestions and pointing out potential for improvement. Professor Zwaenepoel did not know me before this thesis, but it did not prevent him from confidently taking on the task of the supervisor.

From the Adobe side I would like to especially thank my supervisor, Bertrand Delacrétaz, for proposing me this thesis topic in the first place, and for his continuous supervision, suggestions, and guidance throughout it, masterfully directing my efforts into the right direction. This thesis would not have been possible without his dedicated involvement. I would also like to extend my gratitude to my formal line manager, Alexander Saar, for supporting me on the administrative level, and to my local point of contact in Basel, Stefan Egli, for his assistance and valuable clarifications on how the different elements in Sling and Oak work. I would additionally like to thank everyone who made it possible for me to start my internship on time and made my arrival and stay in Basel seamless and comfortable.

Furthermore, I would like to thank everyone who made it possible for me to study at EPFL, particularly the Distributed Information Systems laboratory and Professor Karl Aberer personally for offering me the opportunity to work as a research assistant throughout my studies, enabling me to cover the financial aspect of my life in Switzerland. I would like to extend this acknowledgement to my previous professors, teaching assistants, and everyone on the administrative level for making the past three years exceptional.

Finally, I am also deeply grateful to my family and friends for believing in me, supporting me, and for simply being there, encouraging me all along throughout my journey of becoming a Master of Science.

Contents

Figures

Tables

Glossary

AEM	Adobe Experience Manager, a web content management system from Adobe built on top of Sling
Apache	Apache HTTP Server, a general-purpose web server
Apache Sling	Content-driven framework for building web applications in Java
Atomic broadcast	Broadcast primitive in distributed systems that requires that all nodes receive broadcast messages reliably and in the same order
Big flip	Approach to updating distributed systems where a cluster of nodes is updated one half at a time, such that the two halves are not online concurrently during updates
Bundle	A JAR package specifically adapted to be deployed to an OSGi framework
Component	A functional module in OSGi
Config	In this thesis, a particular version of a Sling application
Content repository	Abstract data store defined by JCR that Sling uses as the data tier
Crank file	A file defining an OSGi application that can be started by Crankstart
Crankstart	OSGi application launcher that starts and configures an OSGi framework according to commands in a crank file
Dependency hell	Situation in which the graph of dependencies of a software package includes dependencies on multiple versions of the same third-party package, which cannot be loaded into a system at the same time without proper isolation
Health check	A test in Sling with an expected result, useful for checking whether certain conditions in a running Sling instance are satisfied
Home directory	A directory on the host file system that Sling, OSGi, and JCR use for storing configuration, logs, and other bookkeeping records
HTTP endpoint	A URL under which a Sling instance can be reached
HTTP front-end	A server that maintains a pool of web servers to which it forwards incoming HTTP requests
Jackrabbit	Reference implementation of JCR, used as the default JCR implementation in Sling
JAR hell	Dependency hell specific to the Java platform

JCR	Java Content Repository, specification that defines an abstract model and a Java API for data management and storage
Launchpad	Default Sling application launcher
Minion	In this thesis, one of the Sling instances running a particular version of a Sling application
Oak	An implementation of JCR that focuses on scalability and performance
Online update	An update that satisfies two properties: atomicity and zero downtime
Orchestrator	In this thesis, application managing the continuous delivery mechanism, responsible for spawning and stopping Minions
OSGi	Set of specifications that define a dynamic component model for Java
OSGi framework	Environment, similar to a container, that hosts an OSGi application
Resource	A piece of data available to Sling
Resource resolver	A component responsible for finding (resolving) resources in Sling given their paths
RESTful	Conforming to REST (Representational State Transfer) style of HTTP requests, where requests make use of URIs and HTTP methods (`GET`, `POST`, `PUT`, `DELETE`) in a way that makes each request self-descriptive
Rolling upgrade	Approach to updating distributed systems where each node in a cluster is updated one at a time
Script	A Sling component for handling HTTP requests, similar to a servlet, programmable in languages other than Java
Service	An interface through which OSGi components communicate
Service registry	OSGi module keeping track of all available services
Servlet	A Java component for handling HTTP requests
Sling	See Apache Sling
Virtual resource	A resource in Sling that does not reside in the content repository
Workspace	An element of a JCR repository that contains a tree of data nodes, intended for being used as a branch of the repository content
ZooKeeper	Cluster coordination service that provides services that require atomic broadcast

Chapter 1 Introduction

In the modern computerized world more and more tasks that used to be performed by people are being offloaded to software. With more time on their hands, people are instead focusing on improving the different aspects of this software: adding new features, integrating with other applications, improving scalability. These improvements can be broadly divided into two categories: functional (improvements resulting in new functionality) and non-functional (improvements taking the existing functionality to the next level).

These two categories of improvements meet head-to-head on a critical battlefield: application complexity. Functional improvements inherently increase the complexity of an application, thereby making further functional improvements more difficult. That is why one of the non-functional improvements' primary aims is to decrease complexity and automate whatever possible.

One of such automatable aspects of software development is release. Automating software releases allows for increasing the frequency of releases, which is beneficial because it results in faster delivery of new features and bug fixes to end users. The practice of automated frequent software releases as opposed to long release cycles is generally known as *continuous delivery*. Continuous delivery is becoming commonplace today as indicated by the growing adoption of short release cycles by major applications, prominent examples of which include web browsers such as Firefox and Chrome and suites such as Microsoft Office and Adobe Creative Cloud. Mobile operating systems are designed with built-in continuous delivery mechanisms for apps. This thesis attempts to bring another set of applications up to standard by proposing an approach for continuous delivery in the context of the Apache Sling framework.

This master's thesis was carried out as an internship at the Adobe Research Switzerland office in Basel, Switzerland.

Problem Statement

The objective of this thesis is to explore continuous delivery in the context of Apache Sling applications. Apache Sling [1] is a content-driven web framework—a framework for building web applications. Web applications are hosted on a web server and can be interacted with via a web browser. Because everything on the Internet is expected to be available all the time, web applications are no exception. Continuous delivery in such a context is complicated by the difficulty of performing online updates—updates on a running system without having to shut it down. Sling does not provide online update mechanisms out of the box, and thus

updating Sling applications properly—atomically and without causing downtime—is challenging.

This thesis aims at proposing and validating such a mechanism and automating it, thereby enabling continuous delivery in Sling. As such, the proposition consists of two parts:

1. A mechanism to perform online updates of Sling applications
2. A way to automate this mechanism

Target Audience

This thesis may be of interest to anyone working on continuous delivery or online updates (not necessarily in web applications), particularly in a cluster environment where multiple identical nodes are used to run an application.

Personal Motivation

Web technologies have been of interest to me for a long time. I learned the basics of HTML back in 2003 when I first got hold of a computer at home. The general interest in web technologies has been growing rapidly too as computing started moving into the cloud. This caused speed-ups in the evolution of HTML, CSS and JavaScript, a shift to JavaScript over browser plugins for dynamic effects on web pages, and other similar developments. During my undergraduate years I was working as a web developer for an Internet marketing company in the United States, part-time during school year and full-time during summers, and had a chance to observe the benefits and challenges of developing and deploying web applications. Therefore I found the possibility of writing my master's thesis in this area in the industry compelling and this project interesting.

Company Motivation

Adobe is well known for its design applications collectively known as the Adobe Creative Cloud (formerly Adobe Creative Suite). In recent years Adobe has also been expanding its digital marketing business, focusing on the business-to-business market and offering a suite of applications collectively known as the Adobe Marketing Cloud. One of the applications in the suite is Adobe Experience Manager (AEM, formerly Adobe CQ), which is a web content management system with a focus on marketing and commercial content.

At its core, AEM is a Sling application with custom extensions. Because, like Sling, it does not have a continuous delivery mechanism built in, it has historically been a challenge to upgrade a live setup of AEM from one version to the next. Performing even simple updates such as modifying or customizing existing modules is also a challenge often calling for a trade-off between introducing downtime and temporarily putting the application into an inconsistent state.

As the number of AEM customers grows, it becomes necessary that updates be easier to perform. While the approach presented in this thesis may not be applicable to AEM just yet, the end goal is that it will be eventually incorporated into it.

Scope of the Project

This project is exploratory in nature and its goal is to propose and validate a continuous delivery mechanism in the context of Apache Sling. The implementation provided as part of this project should be treated similarly and may need to be extended, improved, adapted, optimized, or customized before being suitable for production use.

The presented approach introduces some constraints on the way Sling components should be developed. This thesis calls for future work to attempt to lift some of these constraints.

Report Structure

The current chapter introduced the project. The next chapter, “Related Literature”, gives an overview of some research publications dealing with online updates. The “Overview of Apache Sling” chapter that follows gives an overview of the Sling framework, including a description of the state of the art online update options.

The “Improving Continuous Delivery” chapter, the heart of this thesis, comes after that, and gives details on the proposed continuous delivery mechanism. It is followed by the “Evaluation” chapter that gives an overview of performance of the proposed mechanism, which is then discussed in the “Discussion” chapter. The thesis then ends with the “Conclusion” chapter.

Chapter 2 Related Literature

The main challenge of this work lay in the first aspect of the problem: defining a mechanism for updating Sling applications atomically and without downtime. This problem has been known and studied for a long time, with first approaches focusing on dynamically replacing parts of a running program, also known as *hot code swap*, dating back as far as 1983 [2].

Dynamic code replacement is still an active research area today. Because this is inherently a difficult problem, approaches to it usually have significant limitations. The recently published approach for dynamic code replacement in Java, for example, is restricted to method replacement, and as such cannot change interfaces or class layouts [3]. While such limitations may be acceptable for some applications, because of the difficulty of the problem, the effort needed to deploy such approaches in practice would generally outweigh the benefits.

There has also been some research on updating operating systems without the necessity to reboot them. Here approaches are varied from being oriented towards dynamic code replacement and thus calling for modifications to the way existing operating systems handle updates [4] to taking the systems approach to updates and thus being applicable to general-purpose operating systems [5].

With respect to updating distributed systems, two popular approaches that emerged long ago are the *rolling upgrade* and the *big flip*, documented as far back as 2001 [6] [7]. Rolling upgrade calls for each node in a distributed system to be updated one at a time, and thus requires different versions of the nodes to be interoperable. The big flip, on the other hand, entails updating half of the nodes at a time, such that the different halves are never online at the same time during the update—and thus do not need to be interoperable. Both of these mechanisms take a systems approach to updates; both are thus applicable to general updates, and are simple to grasp and apply. Not surprisingly, they remain the industry best practices to this day [8].

Because rolling upgrades require interoperability between application nodes, they also require that nodes of different versions be able to handle user requests destined to different versions. As this may be a problem for some applications, one paper proposes an analytical framework for assessing the risk of these “mixed-version races” [9]. The framework allows the vendors to compare the risk of carrying the update forward with the risk of delaying or entirely cancelling it.

Neither the rolling upgrade nor the big flip is applicable to single-node systems, which led to the proposal of an approach that works on them [10]. This approach does not eliminate

downtime but significantly reduces it by updating the application in a virtual machine while the old version of the application continues its service.

Neither the rolling upgrade nor the big flip also take testing of the application into account (the new version is supposedly tested before being deployed on the cluster). One paper tries to fill in this gap by proposing a staged deployment model of applications [11], where testing happens directly on select user machines. The user machines are clustered according to their environments and initially only selected users from each cluster get access to the updated application. The paper additionally introduces a user-machine testing subsystem that compares the behavior of the application before and after the update, and a reporting subsystem that provides feedback to the vendor.

Finally, Dumitraş and Narasimhan propose an approach similar to the big flip that avoids reducing the original application capacity. It does this by applying the updates in a "parallel universe", on stand-by nodes not used by the application [8]. Termed *Imago*, this approach thus isolates the running version of the application from the update and reduces the risk of failure (the parallel universe can be discarded with no impact on the online system), but requires access to twice the number of resources than needed for the application. The approach is based on the observation that distributed applications have well-defined *ingress points* where users' requests are directed (e.g. an HTTP front-end in case of web applications), and *egress points* where the application communicates with a storage back-end. The application's business logic resides between the ingress and egress points, and this portion of the system architecture can be mirrored, updated, and then, if successful, atomically substituted for the original. An overview of this process is shown in Figure 2.1. The main challenge in this approach is the need to duplicate the storage back-end in order to not have to make the two application versions interoperable. The paper goes in depth discussing an opportunistic mechanism for copying the data while the old version continues

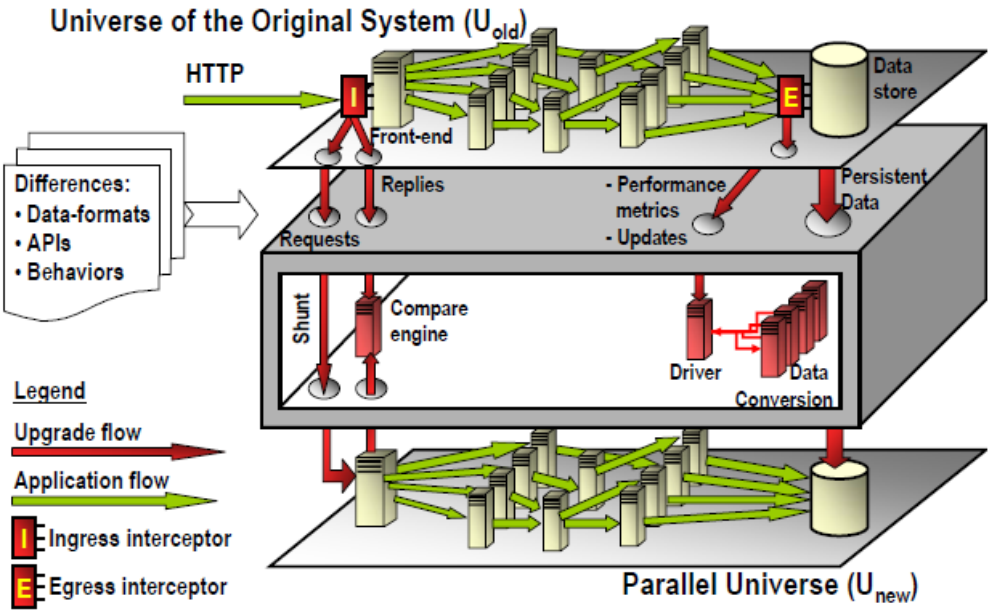


Figure 2.1. An update with Imago. (Image courtesy of the paper; republished with permission from Springer Science and Business Media.)

serving requests that potentially write to the back-end. In order for this data transfer to eventually terminate, the paper suggests that when it is close to the end, either write access be briefly disabled or requests that could potentially have write effects be blocked.

Chapter 3 Overview of Apache Sling

Apache Sling is a content-driven web framework for building web applications. It is aimed at processing HTTP requests in a RESTful way and makes it very easy to do so.

In general, web applications make use of a client–server architecture style known as the *three-tier architecture* that consists of (Figure 3.1):

1. *Presentation tier*, responsible for user interface;
2. *Logic tier*, responsible for user request processing and business logic; and
3. *Data tier*, responsible for storage.

Web applications built in Java make use of *servlets* for the presentation tier, with *JavaServer Pages* (JSP) being a higher-level abstraction of servlets that simplifies the delivery of HTML. Servlets are standard Java classes that are used for processing raw HTTP requests, and it is up to the programmer to make the request mechanism RESTful if needed.

Sling facilitates the approach to building web applications in Java. It provides a content

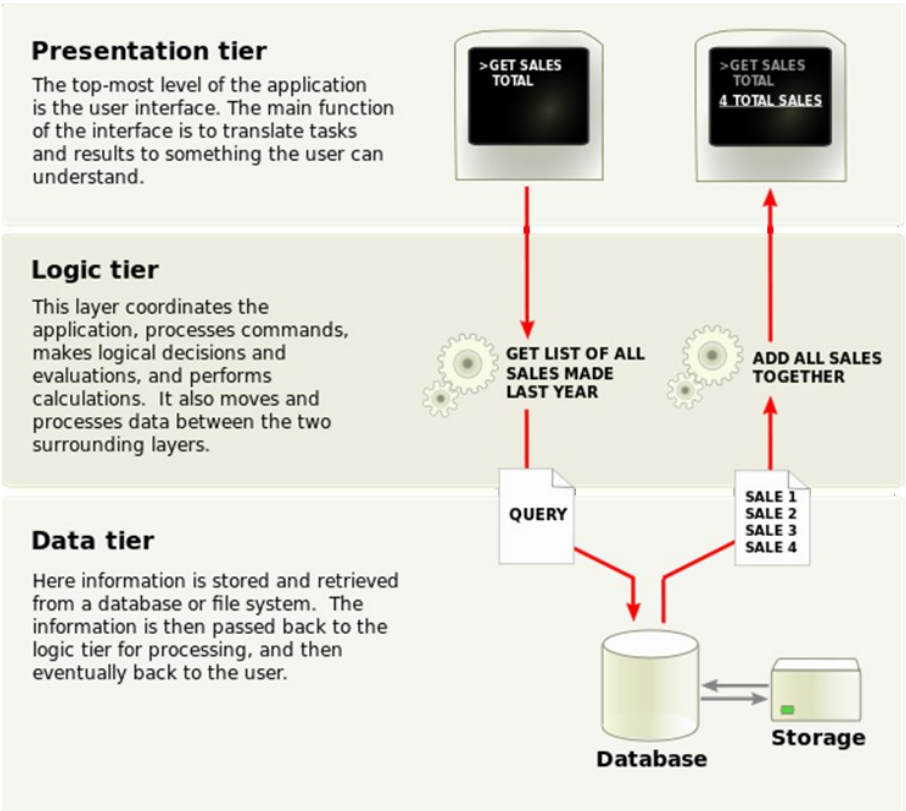


Figure 3.1. The three-tier architecture. (Image courtesy of Wikipedia.)

repository as a standard data tier and RESTful access to it. The content repository is an abstract data store, and multiple actual storage mechanisms (e.g. file system, memory, or database) can be plugged in. For processing user requests, Sling can use either standard servlets or *scripts*, which are an abstraction over servlets and can be written in several languages, including the above-mentioned JSP, server-side JavaScript, Scala, and Groovy. Sling does all this via the *Sling API*, which is an extension of the Servlet API. An implementation of the Sling API, such as Apache Sling, is called a *Sling framework*.

Sling therefore takes care of the data tier and simplifies the implementation of the presentation tier, providing the RESTful mechanism along the way. As such, Sling is well suited to building web applications such as blogs, wikis, and web content management and digital asset management systems.

Architecture

Architecturally, Sling is an OSGi application and is realized as a series of OSGi *bundles* that supply application code. Bundles could be responsible for any of the three tiers of the web application, and users may add their own bundles to implement their business logic or extend Sling. Essentially, although Sling is called a "framework", it is a standard Java application, and applications built on Sling are simply extensions of this application. Because Sling API is an extension of the Servlet API, Sling runs in a servlet container.

Sling uses the Java Content Repository (JCR), an abstract model for data management and storage for Java, for providing the content repository.

A Sling application can thus architecturally be represented as a stack as shown in Figure 3.2.

OSGi

OSGi (Open Service Gateway initiative) is a set of specifications published by the OSGi Alliance that define a dynamic component model for Java [12]. This allows for the development of applications in terms of a set of dynamic and reusable *components*. Components come in the form of bundles which are dynamically deployed in the *OSGi*

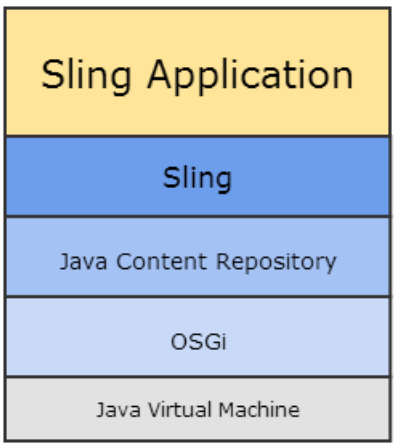


Figure 3.2. Architecture of a Sling application.

framework (similar to a container), and whose life cycle is managed dynamically: they can at any time be installed, started, updated, stopped, or uninstalled without requiring a reboot of the container. The OSGi framework is implemented as a series of layers as shown in Figure 3.3.

Components are isolated at runtime from other components and communicate through well-defined *services*. Services are simply Java interfaces, and service implementations are simply components implementing the interfaces and registered as such in the framework's *service registry*. When a component looks up a service, it simply asks for implementations that are registered under a specific interface or class. Additionally, each service provides a set of standard and custom properties during registration, which can be used to filter out unsuitable services during look-up.

The service registry helps components detect the addition and removal of services and adapt accordingly. This dynamic nature of the OSGi execution environment is one of the major benefits of the OSGi technology because OSGi applications do not rely on its bundles to be started in a specific order during initialization.

The fundamental aim of OSGi is to provide modularity for Java applications. Modularity, in the OSGi sense, is about reusing code, assuming less about the "outside world", and not sharing. Modularity in OSGi is achieved via the concept of bundles. Bundles are standard JARs that contain OSGi components and services and have additional entries in the manifest. In a regular Java application, when a JAR is added to the class path, by default all its classes become visible to every other class in the application. With OSGi, bundles specifically list the Java packages they would like to *export*, and only the exported packages become visible to other bundles, which in turn declare the packages they would like to *import*. Both exports and imports may additionally indicate package versions. The framework is then responsible for wiring the bundles to each other such that all dependencies are satisfied. This mechanism not only allows bundles to hide implementation details, but also resolves the dependency hell (also known as *JAR hell* in the Java world) because it allows for multiple versions of the same bundle to be deployed side-by-side.

When starting, bundles register their services with the service registry, after which other

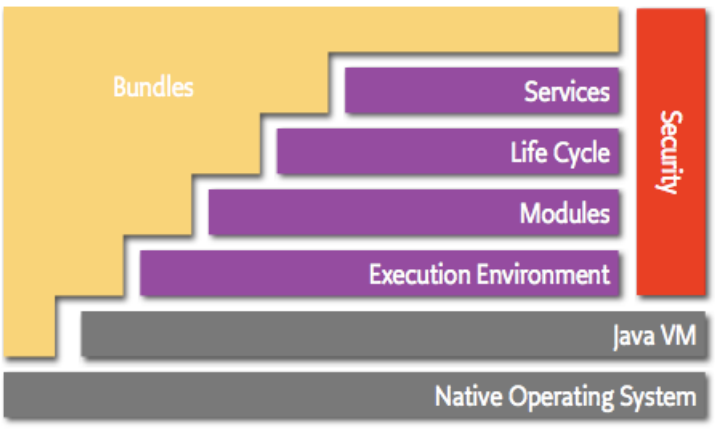


Figure 3.3. OSGi layering. (Image courtesy of the OSGi website.)

components waiting for these services may make use of them. Similarly, when a bundle is stopped, it unregisters its services, and components that were using them are expected to adjust gracefully. Although this puts additional pressure on programmers, this maps very well to the dynamics of the real world, especially in the context of distributed applications. Moreover, this adaptive model allows bundles to be updated without the need to restart the OSGi framework, which was one of the primary goals of OSGi.

OSGi differentiates between two types of configurations. One type, known as *framework properties*, applies to the framework itself. These are global key-value pairs (similar to system properties) that the components may freely use. Additionally, components themselves may expose *configuration properties*. These are also key-value pairs, but are unique to the components that define them. Different components may thus define configuration properties with the same key, whereas the framework properties must all have unique keys.

To define some standard services, the OSGi Alliance also publishes the *Compendium* specification [13]. It includes definitions for common services such as the *Log Service* for logging, system services such as the *Configuration Admin* for component configuration and *Event Admin* for inter-bundle communication, and ubiquitous services such as the *HTTP Service* for sending and receiving HTTP requests.

Besides Sling, other notable examples of applications built on OSGi are the Eclipse IDE and the GlassFish application server.

Sling uses the Apache Felix [14] implementation of the OSGi framework.

Java Content Repository

Java Content Repository (JCR) is a specification adopted through the Java Community Process that defines an abstract model and an API for data management and storage [15] [16]. The JCR model is similar to an object-oriented database and is tailored to storing content in a hierarchical fashion. It addresses the needs of content-driven applications in storing documents and binary objects together with associated metadata.

In addition to storage, JCR provides support for other features such as querying, access control, versioning, locking, transactions, observation, and import from and export to XML. As such, JCR combines features of both databases (querying, locking, transactions) and file systems (hierarchy, access control). JCR implementations are not required to support all features: the JCR specification differentiates between basic features (§§ 4–9) that are required, and additional features (§§ 10–23) that are optional.

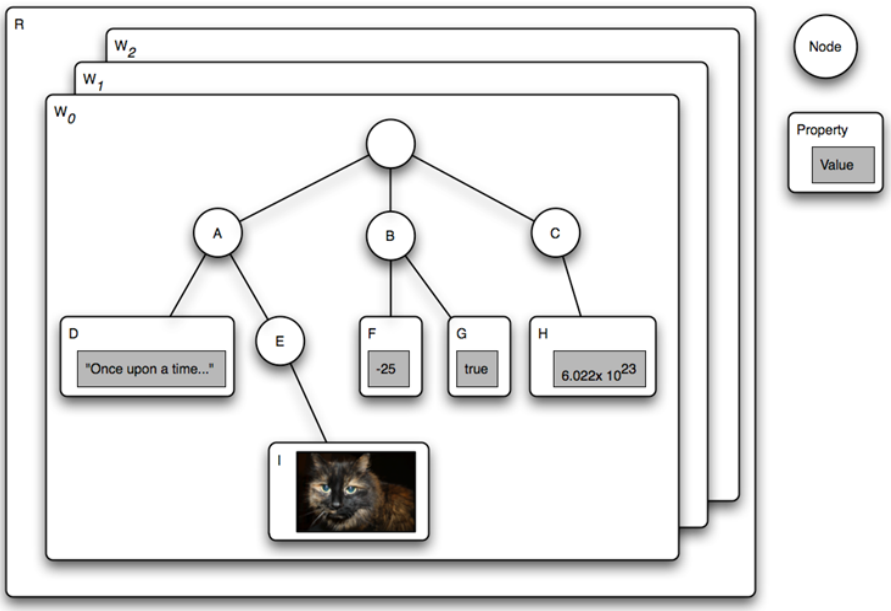


Figure 3.4. An example JCR repository. Depicted here is a repository $\boldsymbol{R}$ with workspaces $\boldsymbol{W_0}$, $\boldsymbol{W_1}$, $\boldsymbol{W_2}$ and the contents of $\boldsymbol{W_0}$. (Image courtesy of the JCR 2.0 specification.)

The JCR repository (Figure 3.4) is composed of one or more *workspaces* each of which is a hierarchical tree of *nodes* where each node can have its associated properties. Data is stored in the values of the properties, which can be as simple as integers or strings, or binary data of arbitrary length. Properties can additionally be multi-valued. Nodes can have one or more types that define the child nodes and properties the node must have. Paths in JCR are represented as sequences of node names from the root node to the target item, typically separated with forward slashes ("/") reminiscent to the Unix file system paths.

While not enforced, workspaces are intended to be used as branches of the same content (similar to branches in Git) because JCR provides support for cloning nodes across workspaces and merging them back. Cloning a node essentially makes a copy of it in a different workspace, marking the two nodes as *corresponding*, after which each node can be modified independently of the other. A node can later be merged with its corresponding node or have its data updated from it. Workspace management, which is required for this branching and merging functionality, is an optional feature of the JCR specification, and thus implementations are not required to support it.

Sling does not only store content in the content repository, but most system data as well.

By default, Sling uses the Apache Jackrabbit [17] implementation of JCR, which is also its reference implementation.

Request Processing

In traditional Java web applications, a servlet to process an incoming HTTP request is determined from the request URL, and the servlet in turn loads some data from the data tier. Sling, on the other hand, places the data first, and uses the request URL to resolve the data

first before choosing which servlet or script will process it. Sling refers to this data as a *resource*.

Request processing in Sling follows three steps:

1. Resource resolution
2. Servlet or script resolution
3. Processing and response

Resource Resolution

During the resource resolution step, the resource requested by the user is resolved, usually to some existing content in the content repository. While resources typically reside in the content repository, this does not have to be the case. The JCR resource resolver, which is the default resource resolver, looks the resources up in the content repository, but Sling also supports custom resource resolvers that could provide *virtual resources*, with built-in support for bundle-based and file system-based resources. All that is necessary is for these custom resource resolvers to register the (virtual) content repository paths that they are responsible for. Sling uses longest prefix match on these paths to find the appropriate resource resolver, and falls back to the next resolver in case the current one is unable to provide the resource. The JCR resource resolver, being responsible for the root path, is therefore always used as a last resort.

Servlet or Script Resolution

After the resource is resolved, Sling uses the resource type to determine the servlet or script that will be called to handle the request.

An interesting fact about Sling is that scripts are actually resources and, like resources, can reside in the content repository or be provided by custom resource resolvers. If residing in the content repository, scripts are customarily located under `/apps` or `/libs` paths, which are configured as the default script search paths in Sling.

Scripts are simply extensions of servlets. Scripting languages supported by Sling are provided as engines able to interpret the languages. A script and its corresponding engine are packed into a servlet that interprets the script and handles requests accordingly. Therefore, strictly speaking, it is always servlets that handle requests.

Servlets are OSGi services which must also register with the framework's service registry. Upon registration, a servlet is expected to provide properties that specify the resource types, extensions, and HTTP methods the servlet is responsible for. Sling uses these properties to pick a suitable servlet or script to handle a request, and provides default servlets that handle requests for which no suitable servlet or script can be found.

Appendix A covers Sling servlet and script resolution in more detail.

Request Handling and Response

When the servlet or script is resolved, the request is ready to be handled. Sling calls the resolved servlet or script to handle the request and hands it the resolved resource. The servlet

or script then processes the user-supplied input parameters, if any, and generates the response using the resource and possibly including other resources.

HTTP Sessions

As Sling API is an extension of the Servlet API, Sling implicitly provides support for HTTP sessions.

Because by definition HTTP is a stateless protocol, the web server normally treats each HTTP request in isolation—without any memory of the previous interactions with the same user. An HTTP session is an attempt to make HTTP stateful whereby the server generates a session ID that it sends to the user in a cookie, and is then able to maintain state by associating it with this ID. This is typically how web servers "remember" that a user is logged in (so that authenticating credentials do not need to be sent in every user request), what the contents of the user's shopping cart are, etc.

HTTP sessions introduce problems in case the web application is hosted on a cluster of servers: either each of them must have access to the same state information, or HTTP requests from the same user must always be forwarded to the same web server.

Because Sling provides a unified storage mechanism—the content repository—it discourages the use of raw HTTP sessions of the Servlet API. Instead, servlets and scripts in Sling should save state information in the content repository.

Deployment and Runtime

A Sling application can be launched either by itself from a standalone JAR, or be deployed as a web application in an application server. Internally, Sling does not differentiate between the two launch options.

Upon launch, Sling is configured with the port number it should use for its HTTP server (if launched from the standalone JAR) and a *home directory*. The home directory is a directory on the host file system that Sling, OSGi, and JCR use for storing configuration, logs, and other bookkeeping records. For example, file system-based content repository back-ends typically persist data in the Sling home directory.

Selected Services

Sling is composed of a series of OSGi bundles that constitute the various modules making up its infrastructure. An out-of-the-box Sling instance consists of over 100 bundles, though not all of them may be required for a given application and may thus be removed in production.

The bundles contain a number of components and services that define Sling. These components and services provide access to the JCR repository and allow features such as logging, user authentication, support for various formats, scripting, task scheduling, testing, and so on. Some of the core Sling bundles are:

- *Web Console.* Apache Felix, the OSGi framework implementation that Sling uses, provides a web console for managing the framework (Figure 3.5). The web console provides an overview of the framework and the ability to configure components, manage bundles, and view system information. Individual components may also add their own pages to the web console. The web console can be found at the `/system/console` path in Sling.
- *Content Loader.* Bundles may provide content (also called "initial content") that they wish to load into the content repository upon being installed. Initial content may be individual content files or descriptor files in XML or JSON specifying entire content sub-trees. The Content Loader component is responsible for loading this content into the content repository. Because Sling scripts are also content, initial content may also be used to load scripts.
- *JCR Installer.* The JCR Installer component is capable of installing bundles and component configurations found in the content repository: it does so by monitoring specific paths. By default, the JCR Installer is configured to monitor folders named `install` located up to four levels deep under the previously mentioned `/apps` and `/libs` paths (which contain scripts). Bundles or configurations are installed in Sling upon being added to these locations and are uninstalled upon being removed.
- *Discovery.* The Discovery Service provides an overview of the Sling cluster topology for multi-node deployments of Sling that share a content repository. This includes the HTTP endpoints at which each Sling instance can be reached. A topology may consist of multiple clusters each of which may in turn consist of multiple Sling instances. Each cluster of instances elects a leader that may be used in case some work needs to be done on only one instance (e.g. restructuring of the repository). The default

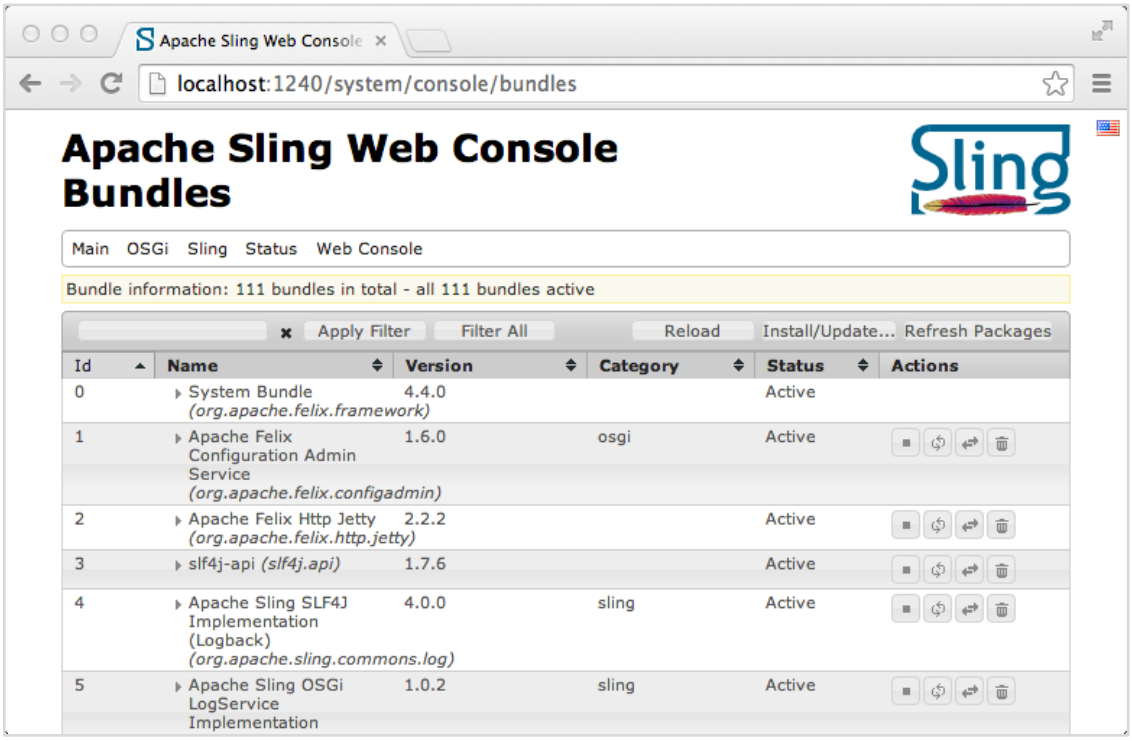


Figure 3.5. Sling web console.

implementation of the service uses the content repository for inter-node communication.

- *Health Checks.* Sling provides the notion of *health checks* for checking whether some aspect of the Sling instance is as expected, typically used by components. A health check can be viewed as a test that defines an expected result; if the actual result is not as expected, the health check is considered as failed, which may indicate a problem with the instance. Some use cases for health checks include verifying the integrity of the content repository, ensuring that a specific bundle is active, and checking if Sling is configured properly. A health check service is a service capable of executing health checks. For example, one available service implementation allows running JUnit tests as health checks.

Clustering

In a production environment, a Sling application may be running on one Sling instance or on a cluster of instances. Clustering is normally done to increase the load capacity of the application. When a Sling application runs on several Sling instances, it does not matter which of the instances processes a specific HTTP request. Thus, the instances are typically placed behind an HTTP front-end, usually a load balancer, which maintains a list of active instances and distributes incoming requests among them.

The main aspect of configuring a cluster of Sling instances is enabling them to share the same content repository, which may or may not be possible. The question of clustering thus boils down to whether the used JCR implementation supports it.

Jackrabbit, the default JCR implementation used by Sling, does support clustering when the storage back-end can be shared between instances (e.g. a database). However, because Jackrabbit was originally designed long before clustering became commonplace in production, it is limited in terms of scalability. Adobe Experience Manager uses a custom proprietary extension of Jackrabbit called *Content Repository Extreme* (CRX) that, in addition to Jackrabbit clustering, can share an otherwise local content repository back-end via replication: each AEM instance still uses only its own local content repository, but any time an instance writes to it, the write is propagated by CRX to all instances in the cluster. CRX thus improves the read scalability, but write scalability is still limited.

A relatively recent development in the Jackrabbit project is the evolution of *Jackrabbit Oak* (also known simply as *Oak*) [18], which is an alternative implementation of the JCR API distinct from Jackrabbit that focuses on scalability and performance. The goal of Oak is to provide more out-of-the-box functionality than NoSQL databases (as specified by JCR), but comparable performance. Because Oak is relatively young and saw its first release after the alpha phase only in May 2014, it currently supports a limited number of storage back-ends. At the time of writing, these include local tar files and MongoDB databases, of which only the latter can be clustered.

Continuous Delivery

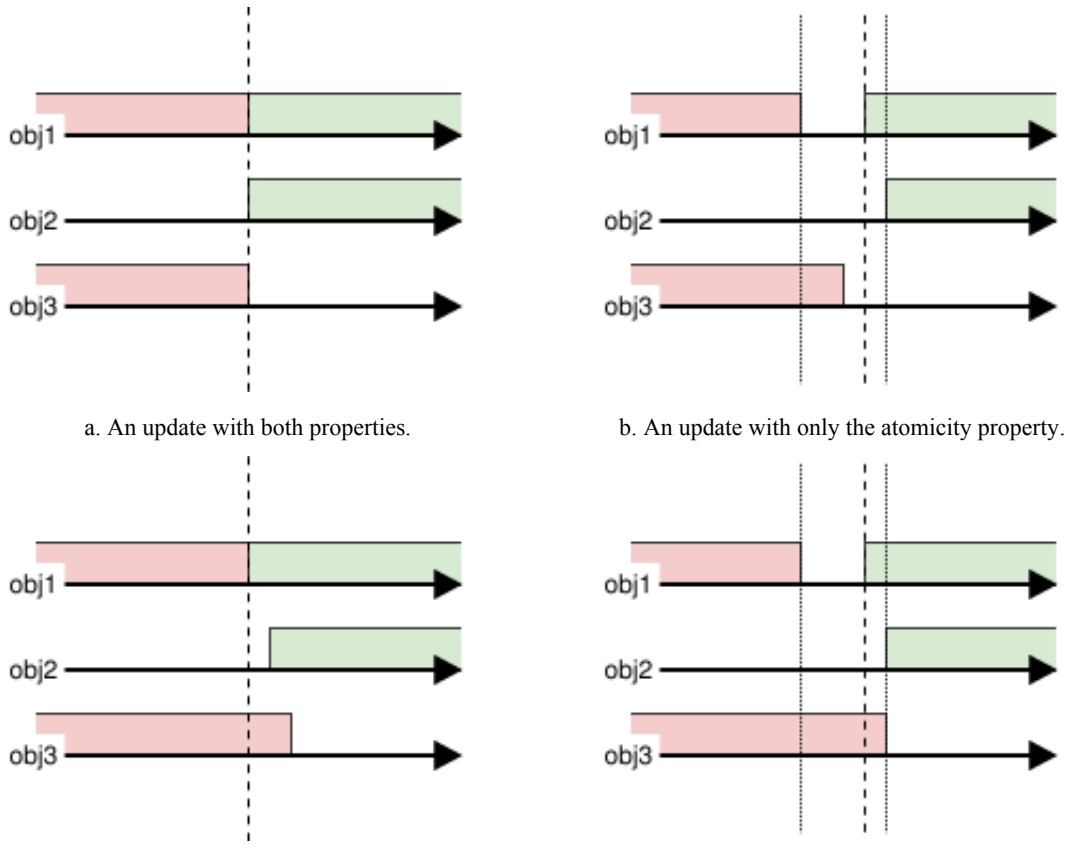
Continuous delivery of Sling applications is complicated by the fact that Sling does not offer a mechanism to perform online updates out of the box. An online update of an application can be formally defined as an update that has two properties:

- *Atomicity:* From the point of view of the user, the update is applied to all perceivable elements at the same time, i.e. there is no point in time during which the user will notice some updated elements and some old elements at the same time. Atomicity is necessary so that users do not see the application in an inconsistent state.
- *Zero downtime:* There is no point in time in which the application or any part of it is unavailable due to the update, i.e. during which a user request will remain unanswered or cause an error to be displayed.

Figure 3.6 demonstrates the different combinations of the above properties in different update outcomes as perceived by the end user along a timeline.

Trivially, a typical offline update—where the system is shut down, updated, and then started back up—satisfies the atomicity property but not the zero downtime property.

OSGi helps Sling improve slightly on this because it does allow updating individual bundles atomically without shutting down the framework, which partially satisfies the zero downtime



a. An update with both properties.

b. An update with only the atomicity property.

c. An update with only the zero downtime property.

d. An update with neither property.

Figure 3.6. Effects of atomicity and zero downtime properties on an update. Red (left) and green (right) rectangles represent old and new versions of objects respectively. Here, `obj1` is updated, `obj2` is added, and `obj3` is removed during the update. Vertical dashed line tests atomicity. Vertical dotted lines indicate downtime.

property. However, zero downtime of the bundle being updated is not guaranteed because the framework specifically allows for components and services to disappear. For Sling, this means that servlets and scripts provided by the bundle may become unavailable during an update of the bundle, though other parts of the application would remain available. Additionally, OSGi also does not allow updating *a set* of bundles atomically, and updating bundles one-by-one would not be atomic. Using the OSGi mechanisms is therefore not an option.

One plausible way to update a Sling application would be to utilize the *systems approach*: to make individual Sling instances immutable and instead of updating them individually, bring up new instances and switch them for the old. This mechanism would shift the complexity of an online update to the systems level and also simplify reasoning about the instances. Such an approach is similar to an offline update in that the two sets of instances are isolated, and therefore inherits the atomicity property. The zero downtime property can be satisfied if the switch between the instances can be done without downtime.

Therefore, there are three possible ways to update a Sling application running on a cluster of instances, as summarized in Table 3.1.

Table 3.1. Summary of update options in Sling.

	Atomicity	Zero Downtime
Offline update	+	−
OSGi-based update	−	±
Systems approach update	+	+*

(*) if it is possible to switch instances without downtime

Chapter 4 Improving Continuous Delivery

The last update option mentioned in Table 3.1 looks promising because it satisfies both properties of online updates, with the assumption that it is possible to reconfigure the HTTP front-end without downtime. This approach also closely resembles the Imago system described by Dumitraş and Narasimhan as summarized in the "Related Literature" chapter, thus affirming its applicability.

A valid concern that may arise is why not instead adapt OSGi to provide an ability to update a set of bundles atomically? Since there is already a mechanism for updating one bundle atomically, it should be possible to extend it to a set of bundles. While it might certainly be possible to satisfy the atomicity property this way, satisfying the zero downtime property fully might be a lot harder. As a reminder, OSGi does not guarantee zero downtime of the bundle being updated. Such a guarantee may only be possible with dynamic code replacement (which OSGi does not currently do), which, as has been mentioned, may not be worth the effort to implement. But a deeper issue is that trying to satisfy the zero downtime property all the way in OSGi would go against one of its main principles, that is to explicitly allow components and services to disappear. There is not much that can be done then short of implementing an extension of the OSGi framework specifically for Sling. The systems approach might just be a much better alternative.

The basic premise emerging from the systems approach is that Sling instances are *immutable*. Once started, they are never reconfigured, and each instance is associated with a specific version of the application during its entire lifetime. The instances are created when a new version is available and destroyed once it is outdated. Immutability is good. It simplifies things in concurrent programming because immutable objects are by definition thread-safe and simpler to reason about. If carried over to distributed programming, immutable nodes offer comparable benefits: they are more predictable and simpler to understand.

Proposed Solution

The systems approach by itself already provides a vague specification of the update mechanism: start new Sling instances, update them, and reconfigure the HTTP front-end to use them instead of the old instances. This specification needs to be made more concrete.

Problem 1: Online Update Mechanism

The vague specification actually produces quite a clear mechanism for the first stated problem of this thesis, defining a way to update Sling applications atomically and without

downtime. One thing that needs to be defined more concretely is how to migrate the content repository data during the update.

A suitable candidate for this is a content repository with branching and merging capabilities (i.e. the optional workspace management feature of JCR). The content repository would be shared among all Sling instances. When a new version of the application is available, a new branch in the content repository would be created for it. When the new instances start, they would update it as needed. When they are ready, the branch on which the older instances were working would be merged into the new branch (to import any changes that happened since the creation of the branch) and the front-end would be switched to point to the new instances.

This leads to a mechanism that satisfies the first problem. Assuming that the current version of the Sling application is v, when version $v + 1$ becomes available, branch b_{v+1} is created in the shared content repository and a set of Sling instances S_{v+1} running version $v + 1$ and using the branch is started. When the instances are initialized, branch b_v is atomically merged into branch b_{v+1} and the HTTP front-end is reconfigured to use instances S_{v+1} instead of S_v. Instances S_v and branch b_v in the content repository can at this point be discarded.

Problem 2: Automating the Online Update Mechanism

For the second problem, automating the above approach, the necessary things are:

- a way to define a particular version of the application,
- a way to detect that this new version is available,
- a way to start instances of this version,
- a way to detect that the instances are properly initialized and ready,
- a way to reconfigure the HTTP front-end, and
- a way to stop the old instances.

A version, or *config*, of the Sling application should be defined in a single file, which can be called the *application definition file*. A single file is easy to distribute because the instances may potentially be brought up on distinct machines. This file could be either a package actually containing the application code, or a text file listing the OSGi properties, configurations, and bundles that could be read by a parser. This single file could be put somewhere where updates to it can be detected, which could be a version control system.

The remaining tasks are all closely related. The version control system could be monitored by a separate entity that, upon an update to the application definition file, would download it and start Sling instances from it. This entity could also be responsible for detecting when the instances are ready, reconfiguring the HTTP front-end, and stopping the old instances. This entity would thus *orchestrate* the entire online update process, and thus could be called the *Orchestrator*. The instances that it starts could then appropriately be called *Minions*.

Minions could announce their readiness to a cluster coordination service that the Orchestrator would also be listening to, indicating which application version they are running. Before making the announcements, the Minions should ensure that they have initialized correctly.

Tentative Solution Proposal

The application definition file, the version control system, the Orchestrator, the cluster coordination service, and self-checking Minions together form a mechanism that satisfies the second stated problem of this thesis, automation. Combined with the branching and merging technique for migrating the content repository data, a solution that satisfies both stated problems is obtained. This results in the deployment diagram shown in Figure 4.1 (where C_1 and C_2 represent two different configs).

With this solution, the whole update scenario would work as follows:

1. The administrator of the application defines a new config and pushes the new application definition file to a version control system monitored by the Orchestrator.
2. The Orchestrator detects the push, creates a new branch in the content repository, and starts n Minion instances running the new config.
3. The Minion instances determine if they initialized properly, and if so, announce their readiness to a cluster coordination service also monitored by the Orchestrator.
4. The Orchestrator waits until all the Minions are ready, merges the new content repository branch with the old branch, and reconfigures the HTTP front-end to switch to the new Minions.

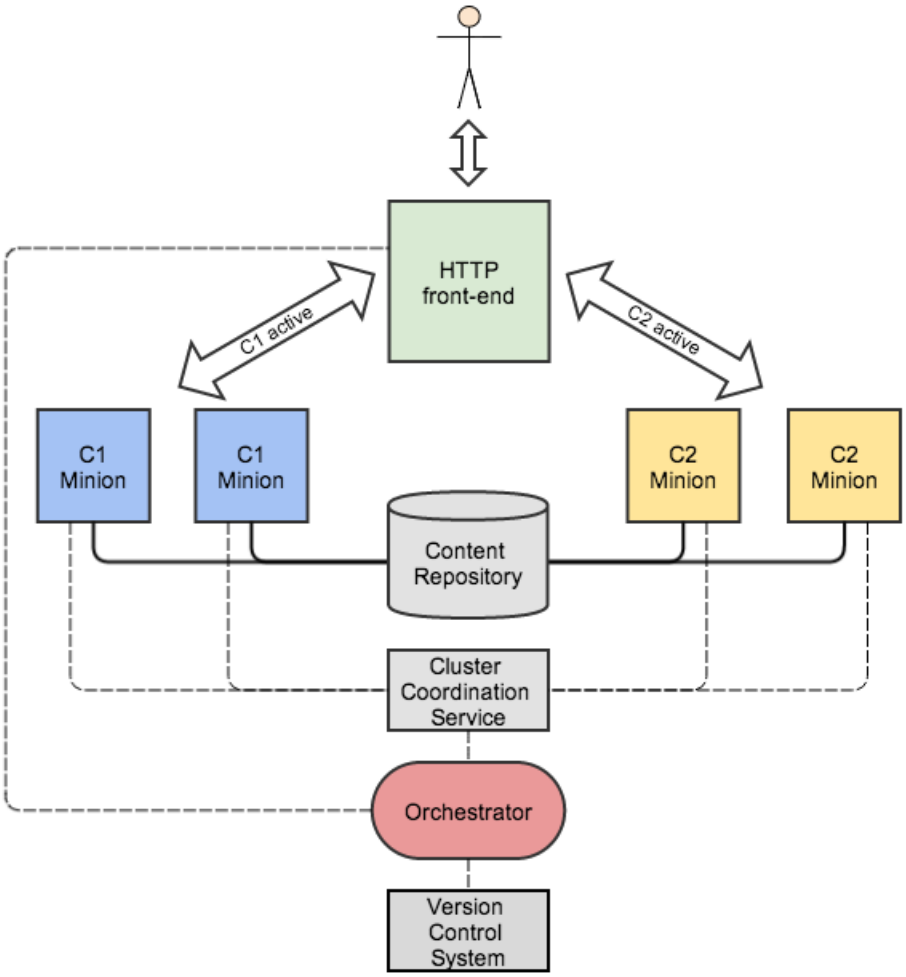


Figure 4.1. Deployment diagram of the proposed solution.

5. The HTTP front-end gracefully switches from old Minions to the new ones.
6. The Orchestrator terminates the old Minions.

Each update would thus take the amount of time it would take to start the updated application from scratch. This time is usually on the order of a few minutes.

In addition to being easy to understand, this approach has a few other benefits as a side effect:

- It makes dynamic scaling possible because additional Minions can be brought up at any time.
- New Minions can be extensively tested before being switched to, thus preventing broken versions of the application from becoming public.
- It enables soft launches where the two sets of Minions coexist, with only a portion of all requests being forwarded to the new ones.

It can be seen that this solution is very similar to the Imago system of Dumitraş and Narasimhan. The new Minions resemble the "parallel universe" in Imago, and the switch of the HTTP front-end resembles the substitution of universes at the ingress point. The main distinction of this solution from Imago is that the storage back-end, the content repository, is not entirely copied during updates, only its content is branched. This introduces the interoperability requirement because it requires the different Sling configs to use the same JCR implementation and the same storage back-end. Imago, on the other hand, avoids the need for interoperability altogether. However, this trade-off in favor of a small amount of interoperability greatly simplifies the proposed solution.

Implementation and Proposal Refinements

With the tentative solution proposal available, implementation could begin. Iterative and incremental software development approach was used to implement this solution, ending each iteration, called a *volume*, with a functioning prototype that implements a part of the functionality. There were a total of five prototypes, with the fifth implementing the complete proposed solution.

Table 4.1 shows the functionality contained in each volume. Complete code and instructions for each can be found in a public GitHub repository [19]. Each volume is tagged as a release, so the code of each volume can be obtained individually.

Table 4.1. Functionality of each prototype.

Component	Functionality	Volume				
		1	2	3	4	5
Building Blocks	HTTP front-end	✔	✔	✔	✔	✔
	Cluster coordination service	✔	✔	✔	✔	✔
	Shared content repository		✔	✔	✔	✔
	Application definition file			✔	✔	✔
	Version control system				✔	✔
Minions	Self-announcement	✔	✔	✔	✔	✔
	Self-checking		✔	✔	✔	✔
The Orchestrator	Minion detection	✔	✔	✔	✔	✔
	Front-end reconfiguration	✔	✔	✔	✔	✔
	Version control system monitoring				✔	✔
	Minion control					✔

Each component's implementation will now be covered in detail, describing how the individual pieces of functionality were implemented.

Building Blocks

Before diving into the implementation details of Minions and the Orchestrator, it is necessary to define how other components of the deployment diagram, which can be thought of as the building blocks, look like.

HTTP Front-End

Apache HTTP Server [20] with the `mod_proxy_balancer` module was used for the HTTP front-end. The module is capable of providing the front-end functionality required while the server has a graceful restart option as required by the proposed solution.

Apache HTTP Server, simply known as Apache, is an open-source general-purpose HTTP server available for all major operating systems. As of June 2014, it is the most popular web server in use with 36.5% market share [21]. Apache is a modular server, with only the basic functionality included in the core. Additional features are installed via a variety of modules, which can provide support for features such as various server-side programming languages, authentication mechanisms, security mechanisms, forward and reverse proxy, load balancing, and so on.

One of the popular modules for Apache is `mod_proxy`, which provides (forward) proxy and gateway (reverse proxy) capabilities for various protocols, optionally together with load balancing. The `mod_proxy_http` module provides proxy support for the HTTP protocol, and the `mod_proxy_balancer` module provides load balancing. These modules together

can provide the HTTP front-end functionality required for forwarding HTTP requests to Sling instances.

Most importantly, Apache supports atomic reconfiguration. The server supports a graceful restart, which enables it to restart without interrupting requests being processed. This aspect of the server was tested to detect any possible downtime, and the latest version, 2.4, was found to indeed restart with no downtime.

Cluster Coordination Service

For the cluster coordination service, the open-source Apache ZooKeeper project [22] [23] was chosen because it also meets the requirements.

ZooKeeper is a cluster coordination service that provides distributed configuration information, naming registry, synchronization, presence, and group services. In a nutshell, it provides services that in one form or another require atomic broadcast, which is tricky to implement from scratch. ZooKeeper uses a novel atomic broadcast protocol called Zab, which shares some characteristics with the well-known Paxos protocol but is more efficient for ZooKeeper purposes [24].

ZooKeeper itself is a system service that runs on the machines making up the ZooKeeper cluster. These machines are called *ZooKeeper servers*, each of which can accept a connection from a *ZooKeeper client* to provide the coordination service. The ZooKeeper project provides client libraries in Java and C, which can be used to connect to the service from within applications.

Shared Content Repository

The Apache Oak implementation of JCR, introduced before in the "Clustering" section, was chosen to provide the shared content repository for Minions.

There was a dilemma when deciding whether to stay with Sling's default Jackrabbit implementation of JCR or to use another implementation. The choice essentially lay in continuing with Jackrabbit or switching to Oak. Oak is by all means a better choice for clustering than Jackrabbit because of its specific focus on scalability. However, one drawback of Oak is that it does not support the optional workspace management feature of the JCR specification, meaning it does not provide branching and merging capabilities either. Although the feature may be implemented in the future, lack thereof meant Oak could not be used for the proposed continuous delivery mechanism.

Drawbacks of the Proposed Approach

This dilemma forced a revisit to the proposed branching and merging mechanism. The proposed solution calls for branching the content repository when a new version $v + 1$ of the application is available, and then merging branch b_v into it when the new Minions are ready to take over. As it stands, there are two potential problems that need to be addressed:

1. What happens if the merge fails?
2. What happens if branch b_v changes after the merge?

The first scenario may occur if, for example, both branches are changed significantly: branch b_v may be changed by ongoing HTTP requests while branch b_{v+1} may change due to the new version of the application calling for a restructuring. It may be okay to require a manual resolution of the conflict, but that would hinder automation. One way to avoid this scenario would be to prevent updates from ever restructuring the content repository.

The second scenario may happen because the S_v set of Minions continues to process HTTP requests all the way until the HTTP front-end is switched to use the S_{v+1} set. This introduces a race condition between the merge and the switch. One way to avoid this condition would be to disable write access to the b_v branch before starting the merge, or to block the corresponding HTTP requests. This is also a required step in the Imago system.

Refinement of the Proposed Approach

These problems led to the exploration of alternative ways to handle the content repository. One observation that can be done in the case of Sling is that the data stored there is of two types:

- *User content:* content visible to users, such as text and images, typically located under `/content`
- *System content:* scripts and data generated by components, typically located under `/apps` and `/libs`

User content is usually not modified across application versions because it is mostly content created by users. Components can generally operate on data created by older component versions. The only data that usually really differs between versions of a Sling application are scripts, which are customarily located under `/apps` and `/libs` paths. However, if scripts are not stored in the content repository and are instead provided as resources in OSGi bundles, there is nothing to worry about. The only component that would need reconfiguration is the JCR Installer, which should monitor config-specific paths instead of generic `/apps` and `/libs`.

This results in another idea of handling the Sling content repository across application versions: simply share the same repository across all currently running Minions and ensure that scripts are provided as bundle resources. With this approach, there is no need to branch and merge anymore, though the JCR Installer component must be reconfigured appropriately. Branching and merging were only necessary for user content, and if a content repository is

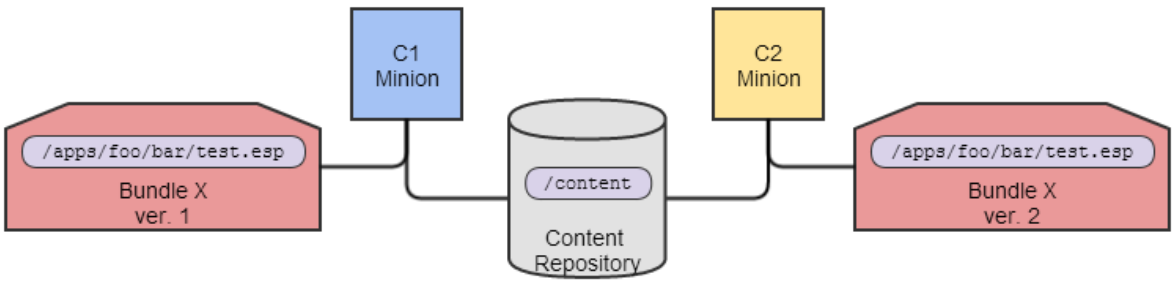


Figure 4.2. Refined approach to the shared content repository. The content repository, providing `/content`, is shared between all Sling configs; scripts are provided by bundles specific to each config.

fully shared, all Minions see the same user content. The only drawback with this approach is that it increases the amount of required interoperability by additionally requiring that new Minions do not change the content repository structure during start-up. However, the problems with branching and merging are avoided. This approach is illustrated in Figure 4.2.

The tentative solution proposal was thus amended with the refined approach to a shared content repository. Oak could now be chosen for the JCR implementation and a MongoDB database was used as the storage back-end.

Application Definition File

Sling configs were chosen to be defined in a single text file that lists the OSGi properties, configurations, and bundles, which can be processed by an application capable of starting the OSGi framework appropriately.

Out of the box, Sling provides two ways to customize an application: installing bundles and configurations via the JCR Installer, or building a custom JAR (or WAR) package.

JCR Installer

Customizing an application via the JCR Installer entails that a vanilla Sling instance is started and then customized at runtime via bundles and configurations explicitly uploaded to the content repository. The application definition file could then, for example, be a list of bundles and configurations that must be uploaded. One drawback of this approach is that the content repository would be populated with unnecessary data. Another, more major drawback, is that a vanilla Sling instance uses Jackrabbit, not Oak. Reconfiguring it to use Oak via the JCR Installer monitoring a Jackrabbit repository results in a chicken-and-egg problem: Sling disconnects from the Jackrabbit repository, which leads to the bundles and configurations installed from it to be uninstalled, which brings Sling back to the vanilla instance.

An alternative approach to using the JCR Installer would be to only install bundles through it, but handle configurations via the Sling web console, using a shell script. In general, using shell scripts is rarely a clean approach. Additionally, this approach does not result in a single application definition file; now there must be at least two: one for bundles and another for configurations.

Using the JCR Installer was therefore not an option.

Sling Launchpad

Sling makes use of a *Launchpad* to define an application with a custom set of OSGi bundles and configurations. The Sling Launchpad is a Java application that starts up the OSGi framework and loads bundles into it. Bundles are specified as Maven dependencies in an XML file known as a *bundle list* and component configurations are specified as text files with a `.cfg` extension. The Launchpad is built into a runnable JAR that can start the application, and a WAR package that can be deployed into an application server to start the application there instead.

The vanilla Sling package is also built with the Launchpad. It is therefore possible to customize that package such that a particular version of the application can be defined as a single JAR or WAR package.

One drawback of this approach is that the application package may be large in size. The vanilla Sling package is over 60 megabytes, and custom application packages may be much larger. For example, the vanilla AEM package is over 450 megabytes. Not all version control systems handle binary files of such sizes well. Additionally, binary files are not diff-friendly, and simply diffing two JAR files is not enough to tell what changes were introduced by the second.

Crankstart

Although its drawbacks are not as catastrophic as those of the JCR Installer, an approach better suited to continuous delivery than the Sling Launchpad was sought after. This resulted in an application called *Crankstart* [25]. It is an OSGi application launcher that reads text files, referred to as *crank files*, defining an OSGi application, launches an OSGi framework, and customizes it according to the instructions in the file. It supports commands for defining OSGi properties and configurations and for installing bundles. Although committed under the Sling project, Crankstart is not limited to Sling and can launch any OSGi application.

Crank files list OSGi bundles in the form of Maven dependencies. Crankstart retrieves these from available Maven repositories. This functionality is achieved via the use of OPS4J Pax URL project providing custom URL handlers [26], one of which can handle custom URLs specifying Maven artifacts.

Because crank files are simple text files, they are extremely lightweight and are well supported by version control systems, also being diff-friendly. In order to avoid having each Minion download each bundle from the Internet, Minion machines can be configured to use one shared Maven repository on the local network (such a repository may be needed for proprietary artifacts anyway). Due to these benefits offered by crank files, they were chosen as the Sling application definition files.

Detailed information about crank files can be found in Appendix B.

Version Control System

Git, an open-source version control system [27], was chosen as the version control system. This choice was arbitrary as there are no specific requirements on this building block.

Git is a distributed version control and source code management system that particularly focuses on distributed and non-linear workflows. Unlike most client–server version control systems, the working copy in Git is a full-fledged repository independent of the one residing on the server that maintains its own history. Git is structured around its branching and merging feature, and encourages the use of branches for common development tasks like fixing bugs, developing new features, or trying out ideas.

The version control system is the last building block of the proposed solution. The deployment diagram introduced before can now be refined as shown in Figure 4.3.

Minions

The Minion functionality was implemented as an OSGi component running on the Minion Sling instances. This functionality is relatively simple because there are only two tasks. Both of these are taken care of when the instances initialize, before they start processing user requests. Hence, Minions incur almost no overhead from the continuous delivery mechanism at runtime.

Self-Announcement

To be discovered by the Orchestrator, Minions announce themselves to the cluster coordination service, i.e. ZooKeeper, indicating the Sling config they are running and the HTTP endpoints (URLs) under which they are reachable. The Minion component uses the Discovery service to detect the endpoints of the Sling instance it is running in. It then creates an ephemeral node in ZooKeeper where it lists these endpoints and the Sling config. The benefit of using an ephemeral node is that it is automatically deleted once ZooKeeper loses contact with the client that created it. This means that the Minion must maintain its session

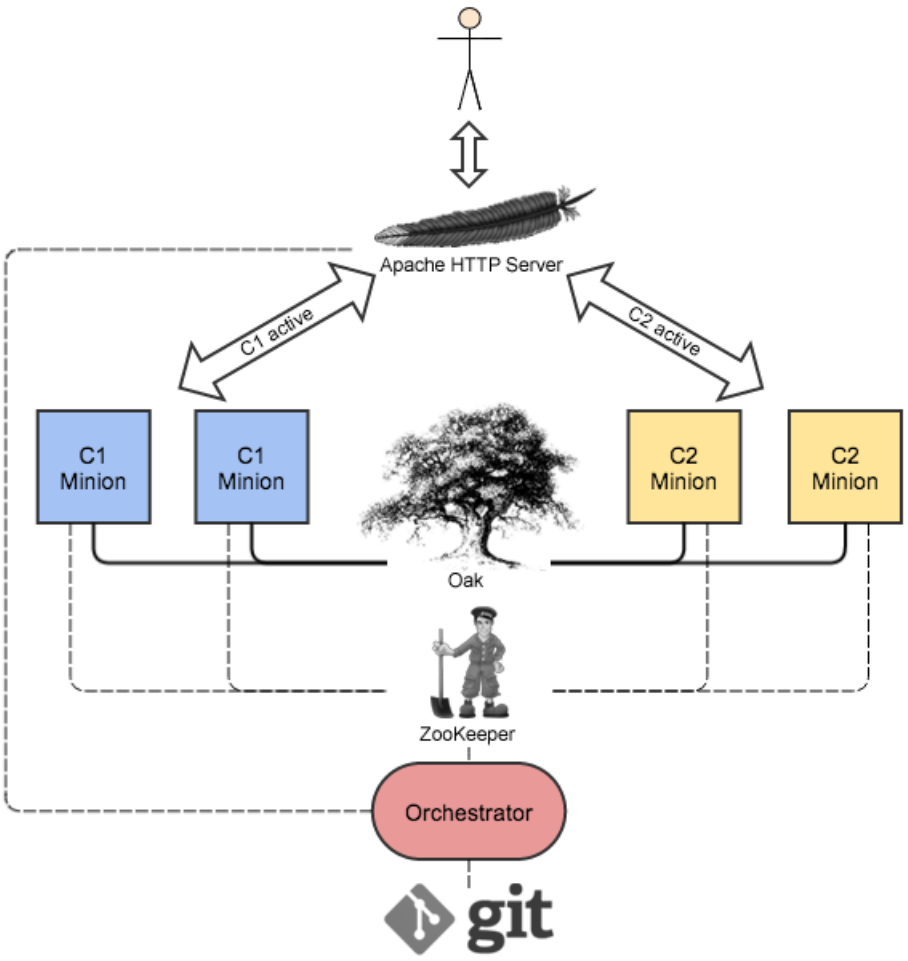


Figure 4.3. Deployment diagram with defined building blocks.

with the ZooKeeper server during its entire lifetime, but the overhead of this is negligible because the session only consists of periodic heartbeats.

Self-Checking

However, before Minions announce themselves, they need to check whether they are ready. Sling health checks are used for this purpose.

Minions have several things to check to ensure that they initialized properly:

- Whether all bundles are active
- Whether important components are active
- Whether the JCR Installer's search path has been reconfigured properly

Sling health checks are perfect for such purposes. This list is not meant to be exhaustive; more health checks may be necessary in real scenarios and can be added by individual applications. In the current implementation, the "important components" are considered to be the JCR Resource Resolver and the JCR Installer.

Each Minion runs health checks repeatedly (with exponential back-off) until they succeed, at which point it makes the announcement to ZooKeeper.

The Orchestrator

The Orchestrator is also a Sling instance. Although with the current functionality it does not need to be one (it is started with Crankstart, but simply being an OSGi application would have sufficed for that), future work on this project may require it. For example, if the content repository handling mechanism is modified, the Orchestrator may need to access the application's content repository, which would be easier if it is already a Sling instance.

The Orchestrator functionality was also implemented as an OSGi component, though it is more involved than that of the Minion component. The Orchestrator makes use of four sub-components each performing one of its four tasks.

The Orchestrator is responsible for starting and stopping the Minions for each version of the Sling application. Minions are started when a new application definition file becomes available, and stopped when the version of the application they run is outdated. The Orchestrator exposes a configurable parameter n, the number of Minions that must be brought up for each config.

At all times, the Orchestrator maintains two values: the active Sling config and the target Sling config. The active Sling config is the config that the front-end is currently configured for, i.e. the application version currently exposed to users. The target config is the config the Orchestrator *wants* to configure the front-end for. When a version v of the Sling application is running on the cluster, both active and target configs are v. When the Orchestrator detects that a definition file for config $v + 1$ is available, $v + 1$ becomes the target config. When n Minions running version $v + 1$ become available, $v + 1$ becomes the active config: the Orchestrator reconfigures the front-end to use the new Minions and stops the old Minions.

The Orchestrator provides a status page (Figure 4.4) where its progress can be monitored. It shows the active and target configs, endpoints of their corresponding Minions, and a log of the Orchestrator's actions.

Minion Detection

The Orchestrator monitors the ZooKeeper node where Minions register to know which Minions are currently active. It does so by registering a watch on the node where Minions place their ephemeral nodes. ZooKeeper thus notifies the Orchestrator when a Minion registers or unregisters. The Orchestrator reads each new node to find out the HTTP endpoints and Sling config of each newly registered Minion.

Front-end Reconfiguration

To reconfigure the HTTP front-end, the Orchestrator writes one HTTP endpoint of each Minion into a configuration file used by Apache and then gracefully restarts it.

The Orchestrator requires that Apache's main configuration file, typically named `httpd.conf`, includes the necessary directives to enable load balancing, and additionally refers to another file for the list of actual load balancing nodes. This other file is where the Orchestrator lists the new Minions' HTTP endpoints when it makes the switch to a new config. Appendix C gives more details on configuring Apache for the Orchestrator.

In the current implementation, the Orchestrator randomly picks one of the endpoints of each Minion (several endpoints exist when the Minion has multiple IP addresses) to write to the file. When the file is ready, the Orchestrator makes a system call to the Apache executable, giving it the `graceful` command. The command causes the server to restart gracefully, i.e. letting current requests finish processing. The system call functionality is achieved using the

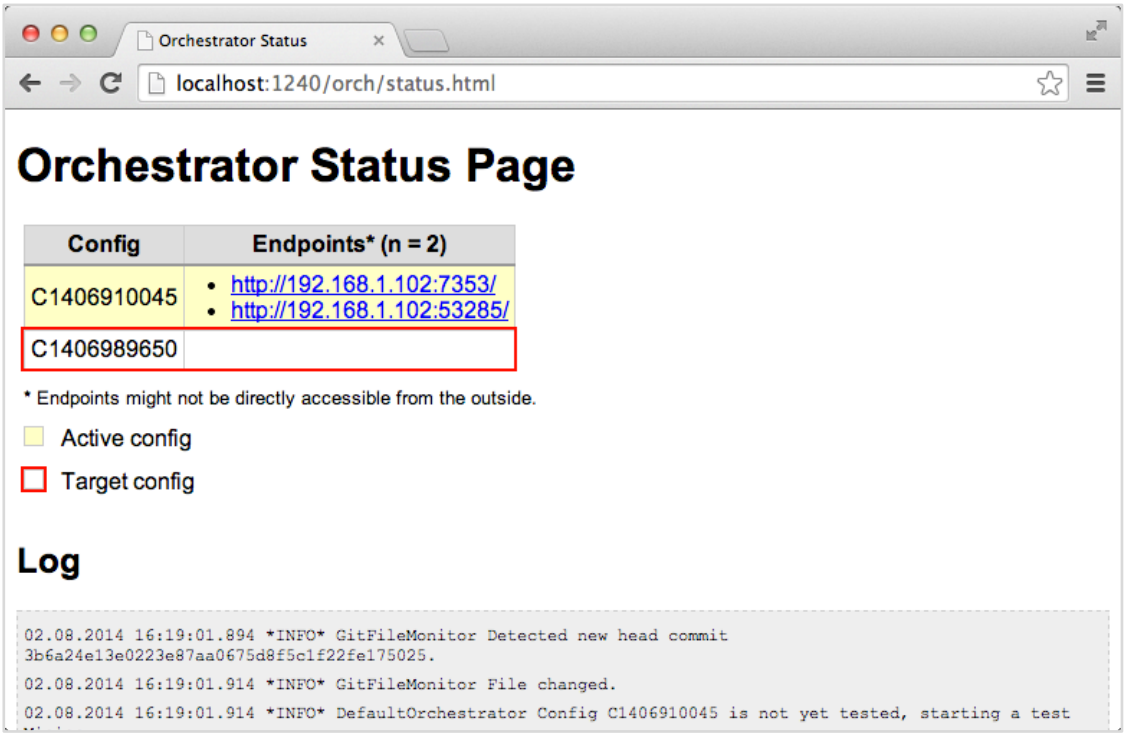


Figure 4.4. Orchestrator status page.

Apache Commons Exec [28] library.

Version Control System Monitoring

The Git repository, used to store the crank file defining each Sling config, is monitored by the Orchestrator for changes. The Orchestrator accesses it via the JGit library [29]. The Orchestrator is configured with the URL of the repository, which could be local or remote, and a path to the crank file within that repository. If the repository is remote, the Orchestrator clones it in the Sling home directory first. The monitoring functionality is achieved by polling the repository at a configurable interval.

When the Orchestrator detects a change to the crank file, it downloads it to the Sling home directory and attempts to start n Minions from it.

Minion Control

The Orchestrator uses Crankstart to start n Minions from each newly available crank file downloaded from the Git repository.

When a new config becomes available and the Orchestrator obtains the crank file for it, it first starts only one Minion. This is done to ensure that the new config is not broken: if the Minion successfully registers with ZooKeeper, the Orchestrator brings up the remaining $n - 1$ Minions. Minions are brought up using system calls to the Crankstart executable.

In the current implementation, Minions are brought up on the same machine as the Orchestrator.

Minion control functionality was the last functionality of the proposed continuous delivery mechanism that was implemented. With it in place, the implementation was complete.

Chapter 5 Evaluation

In order to evaluate the described implementation of continuous delivery in Sling, an experiment was designed for testing whether it does an online update, i.e. an update with both atomicity and zero downtime properties. The experiment consists of a simple Sling application that, besides standard Sling bundles, makes use of one additional bundle. The bundle defines a web page visible to the user of the application. This bundle is built in two versions such that the design of the page is different between the two.

In the experiment, the application is launched with version 1 of the experiment bundle. When the application is initialized, it becomes accessible via the HTTP front-end, and the page defined by the bundle becomes visible. The updated application definition file, with the version of the experiment bundle changed from 1 to 2, is then made available. After a while, the application is updated on the front-end. A special testing tool that reports all changes in content continuously monitors the bundle's page on the front-end. This tool helps establish whether both atomicity and zero downtime were achieved during the update.

Experiment Bundle

The purpose of the experiment bundle is two-fold: to help test atomicity and to verify that user content survives the application update.

The web page defined by the bundle is simply a user content node of the content repository rendered with a script also supplied by the bundle. In reality, the node would be created by a user of the application; for the purposes of the experiment, it is created by version 1 of the bundle via the initial content mechanism. Version 2 of the bundle does not create the node, so its availability after the update would indicate that user content was successfully preserved.

The script supplied by the bundle for rendering the content node helps test atomicity. It does so by displaying not only the content of the node, but also the versions of itself and an OSGi service also supplied by the bundle. These two dissimilar application components were chosen intentionally: OSGi services and scripts (or servlets) are things typically differing between application versions, and are also things impacting rendering of pages and thus visible to users. To be atomic, an online update should cause updates of both OSGi elements and request processing elements at the same time.

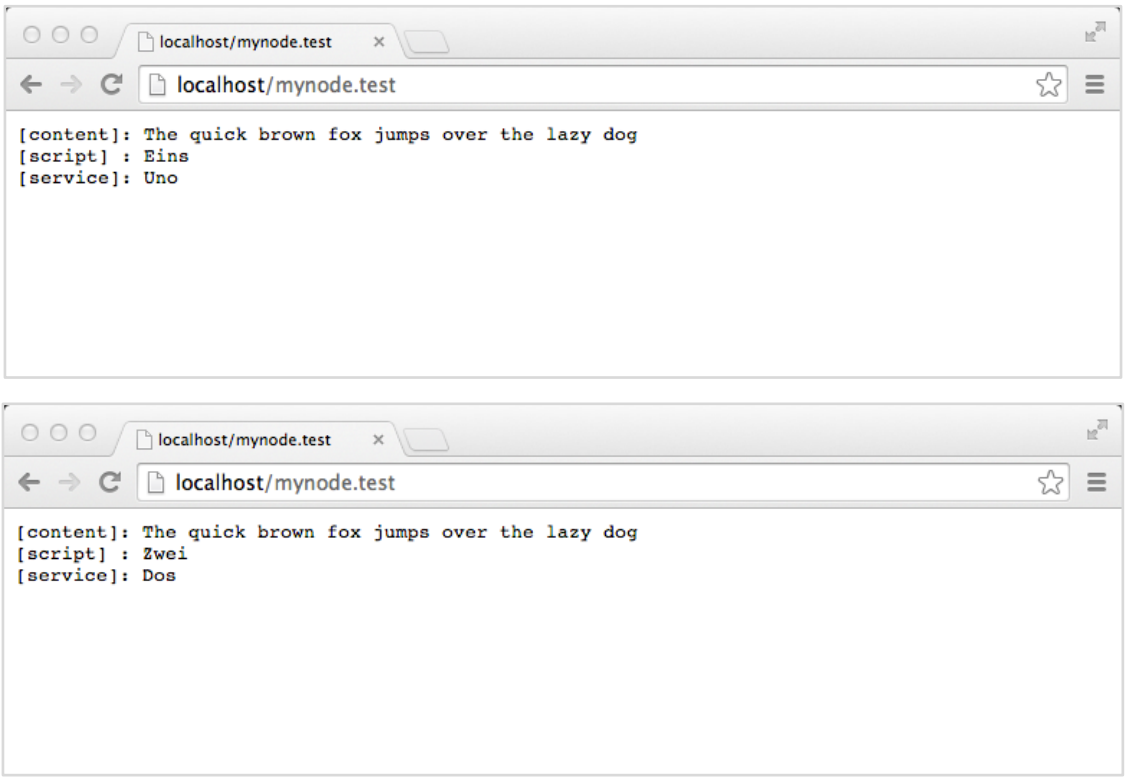


Figure 5.1. Web page defined by the experiment bundle. The web page has two variants, version 1 (top) and version 2 (bottom), depending on the bundle version. The first line is the content of the `mynode` node in the content repository, the second is the version of the script, and the third is the version of the OSGi service.

Figure 5.1 demonstrates the web page under both versions of the experiment bundle.

Testing Tool

A testing tool was developed to test whether atomicity and zero downtime are achieved in the experiment. The tool is a Java application that sends HTTP `GET` requests from multiple threads over and over to a given resource and logs changes in responses, including errors if the response status is not `200 OK`. Each thread thus simulates a user of the application.

In the experiment, the tool monitors the web page defined by the experiment bundle. The tool's logs throughout the simulated update are then analyzed. If the update satisfies both atomicity and zero downtime properties, each thread of the tool would indicate only one change in response and no errors. If any thread encounters more than one change, the update is not atomic; similarly, if any thread encounters a response status other than `200 OK`, the update is not zero-downtime.

Experiment Results

With Apache 2.4 used for the HTTP front-end, each thread of the testing tool does log only one change in the web page during the update, which happens when the front-end is switched to the second Sling config. This indicates that both properties of online updates are met and the update is indeed online. It also indicates that Apache 2.4 is capable of doing a graceful

restart without downtime, which is a requirement for any HTTP front-end used with the proposed solution. Version 2.4 is the latest version of Apache as of July 2014.

It is notable that when Apache 2.2 is used as the front-end instead, the update is atomic, but encounters a brief period of downtime approximately half a second long. This illustrates that the solution does depend on the HTTP front-end supporting zero downtime reconfigurations.

Chapter 6 Discussion

Although the proposed solution can be adapted to most Sling applications, because this project was exploratory in nature, there is room for improvement. There are currently a few limitations, some of which have significant impact on how Sling applications must be built. It also exposes some poor design choices for applications, leading to recommendations on how to avoid them. The solution is by no means complete and lays down the foundation for further work.

Limitations

The proposed solution has limitations that need to be taken into account or lifted before it can be used in production. Some of these are significant because many Sling applications tend to rely on them, but may be sidestepped if the guidelines outlined in the "Recommendations" section below are followed.

The following is a list of the most significant limitations:

- Bundle initial content is not supported
- Different Sling configs must be (to a certain degree) interoperable
- HTTP sessions are not supported
- The Orchestrator and the HTTP front-end are single points of failure
- Minion termination is not graceful

Initial Content

The proposed solution does not take into account bundles providing initial content (content that Sling must load into the content repository), particularly if this content is visible to users (including scripts). One of the fundamental principles of the proposed shared content repository mechanism is that the applications do not modify user content. If Minions running config $v + 1$ modify user content during start-up, it becomes visible to the Minions running config v before the update cycle is complete and the front-end is reconfigured, violating the atomicity principle of online updates.

A reasonable workaround for this limitation is to provide bundle-supplied content as bundle resources instead of as initial content. This limitation is part of a larger limitation of interoperability, discussed next. However, unlike the limitation of interoperability, this limitation can be resolved by a carefully thought-out branching and merging mechanism for the content repository.

Inter-Config Interoperability

Although largely based on the big flip approach to online updates in a cluster environment, the proposed solution does not eliminate the need for interoperability between different versions of the Sling application.

The configs must be interoperable in the way they use the content repository because it is shared between them. This means that they must use the same JCR implementation, the same or compatible versions of this implementation, and the same back-end storage mechanism. For the current implementation, which uses Oak with MongoDB as the back-end, this means that Minions must all use mutually compatible versions of Oak and MongoDB as the back-end.

Furthermore, all components storing data in the content repository must be backwards compatible. This not only means that they must be able to read each other's data, but also that components should not change any aspect of the content repository structure. The latter may be a significant drawback.

One possible way to lift this limitation would be to implement a mechanism for opportunistically copying the content repository to a new location, as proposed by Dumitraş and Narasimhan's paper, taking care to resolve the associated problems. This may, however, be a problem in practice because Sling applications may make use of content repositories several terabytes in size, and copying that amount of data may be impractical due to the amount of time it would likely require. Alternatively, it may be possible to update content repositories using a technique similar to the presented continuous delivery mechanism.

HTTP Sessions

An HTTP front-end can generally be configured to forward requests originating from the same address to the same load-balancing node. Apache also supports this feature. This makes it possible to use the HTTP session mechanism provided by Java servlets. With the proposed continuous delivery mechanism, however, state information maintained in such fashion would not survive application updates because the load-balancing nodes are swapped for new ones. The solution hence does not support servlets' HTTP sessions. However, as Sling discourages their use anyway, pushing the use of the content repository instead for state information, this limitation should not be a problem for most Sling applications.

Single Points of Failure

There are two single points of failure in the proposed solution: the Orchestrator and the HTTP front-end, although in the implemented solution they both reside on the same machine.

The proposed solution assumes that exactly one Orchestrator instance is available at all times. The failure of this instance, however, would not be catastrophic because it would only cause the continuous delivery mechanism to stop working. The application would still be available and the existing Minions would continue processing HTTP requests.

A failure of the HTTP front-end is more serious because it would cause the application to become unavailable. However, the HTTP front-end is not a novelty in the proposed solution,

and existing approaches to providing highly available load balancers would work. Because the DNS protocol supports serving multiple IP addresses for a host, one usual approach is to have several HTTP front-ends (in the case of this project, the Orchestrator would need to reconfigure them all) and to have DNS serve the IP addresses of all front-ends in a single record.

A possible way to lift the limitation on the Orchestrator being a single point of failure would be to have several Orchestrator instances properly collaborating with each other.

Minion Termination

The proposed solution is further limited in that it does not wait for Minions to finish their tasks before they are stopped. For a vast majority of HTTP requests that take only a fraction of a second to execute, this should not be a problem: since the old Minions are stopped only after the HTTP front-end is reconfigured, they would not be executing any requests at the time they are shut down.

This limitation rather applies in the case of HTTP requests that take a long time to execute (e.g. file download), as well as background tasks that Minions may be performing (e.g. generating thumbnails for a newly-uploaded set of images). In the first case, zero downtime property would be violated, whereas the effects of the second depend on how the (updated) application would handle interrupted tasks of previous instances.

One way to lift this limitation would be for the Orchestrator to explicitly check whether a particular Minion is ready to be stopped.

Recommendations

The proposed solution exposed several bad practices of building Sling applications, resulting in the following recommendations on how to achieve the same effects using proper Sling methods. Although all Sling applications should follow these recommendations, applications wishing to use the proposed continuous delivery mechanism absolutely *must* follow them.

Avoid Initial Content

Sling applications should avoid loading content via the initial content mechanism and instead provide necessary content as resources in OSGi bundles.

Content loading is a bad idea in a clustered environment anyway unless the nodes collaborate on who loads this content. The current implementation of the Content Loader service does not account for this and behaves unpredictably if several Sling instances try to load the same content simultaneously. Additionally, in the proposed solution where the same content repository is shared across all configs, the initial content mechanism breaks the isolation of these configs and may result in violation of atomicity during updates.

However, it is understandable that some applications do need to rely on being able to load and modify content: such content may include CSS files, images, and text for the pages. Such

content should be provided as resources in OSGi bundles. Using bundle resources instead of initial content avoids both of the content loading problems outlined above.

A significant drawback with using bundle resources over the content repository, however, is that JCR features cannot be used with them. This means that such resources cannot be searched for or queried, unlike the resources in JCR. However, JCR features can still be used with user content, the amount of which for most applications by far exceeds the amount of content provided by the application itself.

Avoid Scripts in Content Repository

Extrapolating from the previous recommendation, Sling applications should avoid storing scripts in the content repository and instead provide them as bundle resources.

Storing scripts in the content repository goes beyond its original purpose as a *data* store. Scripts are *code*, not data, and should therefore not reside in the data store. OSGi bundles are an ideal location for scripts because bundles provide code. With the proposed approach to continuous delivery, it is impossible to update scripts stored in the content repository without violating atomicity.

Avoid HTTP Sessions

Sling applications should avoid using HTTP sessions provided by servlets and instead use the content repository for state information.

HTTP sessions are bad for a clustered environment because the HTTP front-end must be configured to always forward requests from the same source to the same node. Because Sling already provides a unified storage mechanism—the content repository—all other storage mechanisms should be avoided.

Future Work

Because this project was exploratory in nature, both the presented solution and its implementation have missing pieces that open up a lot of possibilities for future work. These possibilities can be divided into two categories: those that apply only to the current implementation, and those that apply to the solution as a whole.

Implementation-Level Possibilities

The solution already provides for a lot more than is currently implemented. These additional features can be added to the current implementation without the need to rethink the solution itself.

- *Soft Launch.* In the current implementation, the Orchestrator transitions blindly to each newly available config, relying on Minions to do proper self-checking before announcing themselves. This mechanism could be extended to a soft launch, where the transition is done in stages: a new config moves slowly from being in closed beta to being rolled out to a limited percentage of users to finally becoming public.

- *Intelligent Endpoint Selection.* In the current implementation, in case a Minion has several HTTP endpoints (due to having several IP addresses), the Orchestrator randomly picks one of them to configure the HTTP front-end with. Some of these endpoints may not be reachable from the HTTP front-end, which, if picked, would make requests forwarded to the Minion fail. This endpoint selection mechanism should be improved to avoid such possibilities.
- *Minions beyond One Machine.* The Orchestrator in the current implementation is limited to starting Minions only on the same machine as itself. In real-life scenarios, Minions may need to be started on other machines. The current mechanism could be extended to do that, for example using configuration management tools like Puppet [30] and Chef [31].

Solution-Level Possibilities

Another set of possible extensions to the project require amending and refining the solution first before implementing them. All these extensions arise from the limitations of the current solution, previously covered in the "Limitations" section.

- *Graceful Minion Termination.* The current solution provides no mechanism for Minions to delay their termination, which may be necessary in case a Minion is performing a long-running task. Such a mechanism should be added so that long-running user requests and background tasks executing on the Minions being terminated are not interrupted.
- *Removing Inter-Config Interoperability.* The current solution relies heavily on the content repository being shared across all versions of the Sling application. This requires all content repository-interfacing code of the application to be backwards compatible, which may be undesirable. The solution could be refined to lift this interoperability requirement, which, as was explained, may not be a trivial task.
- *Highly Available Orchestrator.* The Orchestrator is a single point of failure in the current solution. It could be extended to provide high availability, for example via the introduction of several Orchestrators collaborating with each other.

Chapter 7 Conclusion

This project was an effort to come up with a continuous delivery mechanism for Apache Sling applications. The presented solution embodies a valid mechanism that satisfies the criteria for being a continuous delivery mechanism because

1. it allows updating applications in an online fashion, and
2. it is automatic.

The solution takes a systems approach to continuous delivery. Its main contribution is that it makes the Sling instances running the application immutable, designating an instance to run a particular version of the application during its entire lifetime. Updates are applied by starting new instances and disposing of the old ones, carefully coordinated to ensure continuous service to users. While designed specifically for Sling, the approach is general enough and can be adapted for many distributed applications, not only those serving web content.

Evaluation of the proposed solution has shown that it indeed causes updates to happen online: they are both atomic and do not cause downtime. In its present form, the solution has several limitations that form either the basis for future work or recommendations on what to avoid when designing and building Sling applications. Despite the limitations, the solution as it stands is already sophisticated enough to be suitable for production use for many existing Sling applications.

Bibliography

[1] "Apache Sling," The Apache Software Foundation, [Online]. Available: http://sling.apache.org/.

[2] T. Bloom, "Dynamic Module Replacement in a Distributed Programming System," Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1983.

[3] A. Nicoara, G. Alonso and T. Roscoe, "Controlled, Systematic, and Efficient Code Replacement for Running Java Programs," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Glasgow, Scotland, 2008.

[4] C. Giuffrida, A. Kuijsten and A. S. Tanenbaum, "Safe and Automatic Live Update for Operating Systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, Texas, USA, 2013.

[5] H. Yamada and K. Kono, "Traveling Forward in Time to Newer Operating Systems Using ShadowReboot," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Houston, Texas, USA, 2013.

[6] E. A. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing,* vol. 5, no. 4, pp. 46-55, 2001.

[7] E. T. Roush, "Cluster Rolling Upgrade Using Multiple Version Support," in *Proceedings of the 3rd IEEE International Conference on Cluster Computing*, Newport Beach, California, USA, 2001.

[8] T. Dumitraş and P. Narasimhan, "Why Do Upgrades Fail and What Can We Do About It?: Toward Dependable, Online Upgrades in Enterprise System," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Urbanna, Illinois, USA, 2009.

[9] T. Dumitraş, P. Narasimhan and E. Tilevich, "To Upgrade or Not to Upgrade: Impact of Online Upgrades Across Multiple Administrative Domains," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Reno/Tahoe, Nevada, USA, 2010.

[10] D. E. Lowell, Y. Saito and E. J. Samberg, "Devirtualizable Virtual Machines Enabling General, Single-node, Online Maintenance," in *Proceedings of the 11th International*

Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, 2004.

[11] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini and W. Zwaenepoel, "Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, Washington, USA, 2007.

[12] The OSGi Alliance, *OSGi Service Platform, Core Specification,* 4.3 ed., 2011.

[13] The OSGi Alliance, *OSGI Service Platform, Service Compendium,* 4.3 ed., 2012.

[14] "Apache Felix," The Apache Software Foundation, [Online]. Available: http://felix.apache.org/.

[15] Day Management AG, *JSR 170: Content Repository for Java™ Technology API,* 2005.

[16] Day Management AG, *JSR 283: Content Repository for Java™ Technology API 2.0,* 2009.

[17] "Apache Jackrabbit," The Apache Software Foundation, [Online]. Available: http://jackrabbit.apache.org/.

[18] "Jackrabbit Oak," The Apache Software Foundation, [Online]. Available: http://jackrabbit.apache.org/oak/.

[19] A. Stetsenko and B. Delacrétaz, "Sling DevOps Experiments," [Online]. Available: https://github.com/ArtyomStetsenko/sling-devops-experiments.

[20] "Apache HTTP Server Project," The Apache Software Foundation, [Online]. Available: https://httpd.apache.org/.

[21] Netcraft Ltd., "June 2014 Web Server Survey," 6 June 2014. [Online]. Available: http://news.netcraft.com/archives/2014/06/06/june-2014-web-server-survey.html.

[22] P. Hunt, M. Konar, F. P. Junqueira and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-Scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, Boston, Massachusetts, USA, 2010.

[23] "Apache ZooKeeper," The Apache Software Foundation, [Online]. Available: http://zookeeper.apache.org/.

[24] B. Reed and F. P. Junqueira, "A Simple Totally Ordered Broadcast Protocol," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, Yorktown Heights, New York, USA, 2008.

[25] "Apache Sling Crankstart," Apache Software Foundation, [Online]. Available: http://svn.apache.org/repos/asf/sling/trunk/contrib/crankstart/.

[26] "OPS4J Pax URL," [Online]. Available: https://ops4j1.jira.com/wiki/display/paxurl.

[27] "Git," [Online]. Available: http://git-scm.com/.

[28] "Apache Commons Exec," The Apache Software Foundation, [Online]. Available: http://commons.apache.org/proper/commons-exec/.

[29] "JGit," The Eclipse Foundation, [Online]. Available: http://www.eclipse.org/jgit/.

[30] "Puppet," Puppet Labs, [Online]. Available: http://puppetlabs.com/puppet.

[31] "Chef," Chef Software, Inc., [Online]. Available: http://www.getchef.com/.

Appendix A Servlet or Script Resolution

Servlet or script resolution is the second of the three-step request processing procedure followed by Sling. The servlet or script is resolved based on four things:

1. Type of the resolved resource
2. Request selectors (dot-separated strings in the request)
3. Request extension (string after the last dot in the request)
4. Request method

Servlets register the resource types, selectors, extensions, and HTTP methods they are responsible for. Scripts, which are resources themselves, expose these parameters in the full paths under which they can be resolved, either as virtual resources or in the content repository.

Figure A.1 shows Sling request processing in detail, including how servlet and script resolution takes place.

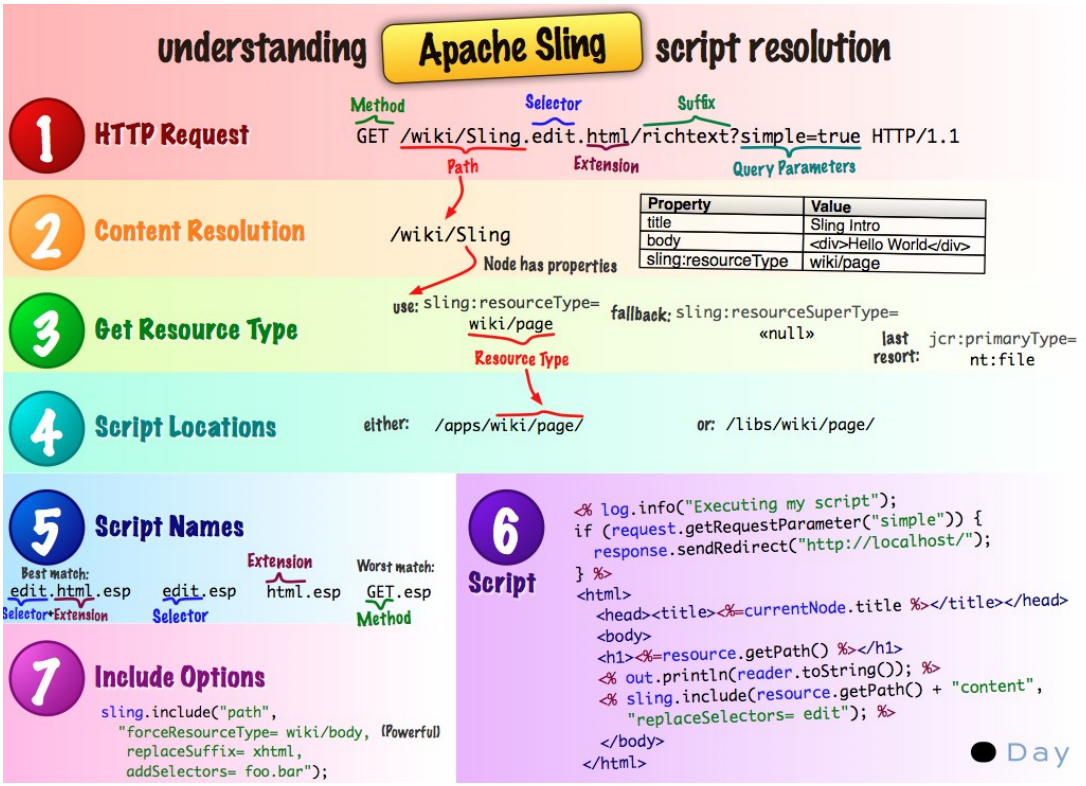


Figure A.1. Sling request processing in detail. (Image courtesy of the AEM 6.0 documentation.)

Appendix B Crank Files

Crank files consist of commands followed by arguments. Most commands are single-line, but multi-line commands are also supported. The main crank file commands are:

- `defaults`
- `osgi.property`
- `config`
- `bundle`

Crankstart supports variables that can be passed on the command line via the `-D` switch of the `java` command. These variables can then be accessed in crank files using the `${variable_name}` syntax. The `defaults` command can be used to set default values for variables that were not defined.

```
defaults sling_home sling-${config}-${port}-crankstart
defaults zk_conn_string localhost:2181
defaults mongo_uri mongodb://localhost:27017
defaults mongo_db oak
```

The `osgi.property` command is used to define OSGi framework properties.

```
osgi.property org.osgi.service.http.port ${port}
osgi.property sling.home ${sling_home}
osgi.property org.apache.sling.commons.log.level INFO
osgi.property org.apache.sling.commons.log.file logs/error.log
osgi.property sling.devops.config ${config}
```

The `config` command is a multi-line command used for specifying OSGi component configurations.

```
config org.apache.sling.installer.provider.jcr.impl.JcrInstaller
          sling.jcrinstall.search.path = /sling-cfg/${config}/apps:200
          sling.jcrinstall.search.path = /sling-cfg/${config}/libs:100
config org.apache.jackrabbit.oak.plugins.document.DocumentNodeStoreService
          mongouri = ${mongo_uri}
          db = ${mongo_db}
```

Finally, the `bundle` command is used to specify OSGi bundles that must be installed, as Maven dependencies.

```
bundle mvn:org.apache.felix/org.apache.felix.http.jetty/2.2.2
bundle mvn:org.slf4j/slf4j-api/1.7.6
bundle mvn:org.apache.zookeeper/zookeeper/3.3.6
```

Appendix C Apache Configuration

Apache HTTP Server must be configured appropriately to be usable as the HTTP front-end for the continuous delivery implementation presented in this thesis. Besides the modules enabled by default, the following additional modules must be activated:

- `mod_headers`
- `mod_proxy`
- `mod_proxy_balancer`
- `mod_proxy_http`
- `mod_slotmem_shm`
- `mod_lbmethod_byrequests`

Additionally, the load balancing functionality must be configured in the server's configuration file as shown below. This configuration exposes the balancer manager status page at the `/balancer-manager` URL of the server and designates the `mod_proxy_balancer.conf` file to list the load balancing nodes (to be written by the Orchestrator).

```
<Location /balancer-manager>
          SetHandler balancer-manager
</Location>

Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/"
env=BALANCER_ROUTE_CHANGED
<Proxy balancer://mycluster growth=100>
          Include mod_proxy_balancer.conf
          ProxySet stickysession=ROUTEID
</Proxy>

ProxyPass /server-status !
ProxyPass /balancer-manager !
ProxyPass / balancer://mycluster/

ProxyPreserveHost On
ProxyRequests Off
```