StickyCode Documentation

Release beta

Michael McCallum

Nov 07, 2018

Contents:

Why “Sticky” well the sticky conventions let you build applications that don’t need glue, it just sticks together, its “Sticky Code”

Sticky Code takes the experience of convention over configuration frameworks and enables it for plain old java

The framework is intended to allow developers to be deliberate all the time and to give them easy obvious decisions to encourage clean code at all times.

Apache 2 Licence

CHAPTER 1

Getting Started

StickyCode provides an abstraction for build an Dependency Injection context. Currently Guice and Spring are supported.

Let StickyCode take care of the glue transparently, so you can focus on writing components.

An example main class:

```
public static void main(String[] args) {
  StickyBootstrap b = StickyBootstrap.crank() (1)
    .scan("com.example"); (2)

  b.start(); (3)

  Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() { (4)

    @Override
    public void run() {
      b.shutdown(); (5)
    }
  }));
}
```

1. Initiate a bootstrap context
2. scan the net.stickycode and com.example packages
3. Start the underlying dependency injection context
4. Add a shutdown thread that will shutdown the context on VM shutdown
5. Invoke shutdown of the bootstrap when the VM shuts down

1.1 Guice

Add the dependencies for maven:

```
<!-- This is for configuration resolution -->
<dependency>
  <groupId>net.stickycode.configuration</groupId>
  <artifactId>sticky-configuration</artifactId>
  <version>[2.5]</version>
</dependency>

<!-- This is for the guice4 binding -->
<dependency>
  <groupId>net.stickycode.configured</groupId>
  <artifactId>sticky-configured-guice4</artifactId>
  <version>[1.6]</version>
</dependency>
```

Note that fixed ranges are used, this causes Maven to fail if you introduce conflicts rather than behaving non-deterministically

1.2 Spring 4

Add the dependencies for maven:

```
<!-- This is for the spring4 binding -->
<dependency>
  <groupId>net.stickycode.configured</groupId>
  <artifactId>sticky-configured-spring4</artifactId>
  <version>[1.3]</version>
</dependency>

<!-- This is for configuration resolution -->
<dependency>
  <groupId>net.stickycode.configuration</groupId>
  <artifactId>sticky-configuration</artifactId>
  <version>[2.5]</version>
</dependency>
```

Note that fixed ranges are used, this causes Maven to fail if you introduce conflicts rather than behaving non-deterministically

CHAPTER 2

Composition

2.1 Dependency Compositions

Using third party libraries is essential to java development. Library and tool support make java what it is.

StickyCompositions use the standard OO concept of composition to provide manageable lifecycles to third party libraries.

The lifecycle of compositions is explained by CompositionVersioning, with some example compositions described here:

LoggingComposition SpringComposition UnitTestingComposition

CHAPTER 3

Stereotypes

In Component based programming the Stereotype annotation tells the Dependency Injection framework that a class is a component and an instance should be created for injection (in the **Context**)

3.1 StickyComponent

The base component to create singletons in the **Context**

This stereotype is a meta annotation and can be used to creation other marker stereotypes to make the code cleaner.

The current marker component stereotypes are Repository, Gateway, Mapper, Service

This will result in a bean called **component** in the DI Context:

```
@StickyCompoent
public class Component {
}
```

To inject the value then you use an Injection marker:

```
@Inject
Component component;
```

3.1.1 StickyRepository

Used for compoenents that represent repositories of data:

```
@StickyRepository
public void IssueRepository {

  Issue findAll() {
    ...
```

(continues on next page)

(continued from previous page)

```
  }
}
```

3.1.2 StickyGateway

Used for components that interface with external systems they act as a gateway

3.1.3 StickyMapper

Used for components that transform data from one form to another

3.1.4 StickyService

Used to mark components that provide a set of business logic

3.2 StickyPlugin

Another meta annotation for components that are intended to be used for multple values to be injected.

Guice requires special semantics for multiple injection i.e. into Lists and Sets:

```
@StickyPlugin
public class A implements Contract {
}

@StickyPlugin
public class B implements Contract {
}

@StickyComponent
public Class Bean {
  @Inject
  Set<Contract> values;
}
```

3.2.1 StickyExtension

For components that are intended as additions to the system

3.2.2 StickyStrategy

For components where you select one of the implementations that are available

3.3 StickyFramework

The StickyCode framework its self is based on components, for some DI systems these components but be initialised in a first pass.

All system components and annotatied as Framework components for this purpose

CHAPTER 4

Configured

4.1 Introduction

Configuration is something that is often an afterthought, its varies by developer to developer, project to project.

The purpose of the `@Configured` annotation is to allow the developer to simply declare a field value as being source outside their control.

This means the developer only needs to decide that they need a value and defer the resolution of it. It makes it an easy decision and encourages deliberate programming.

There are three kinds of configuration:

Application

Configuration that is applied to components when assembled into an application. This configuration is constant across deployments and environments.

Developers can choose these values and they cannot be overridden by environmental configuration

For example a database component defines the database schema, that is common across environments but specific to an application.

Environment

Configuration that is different across environment deployments, but may be common across applications.

This configuration should not be defined by the developer as there is no reasonable values they can choose.

An example of this would be the size of the thread pool

Default

Configuration that can be defined at development time, setting sensible values

The value can be overridden at run time to allow for that fact that developers are not omniscient

An example might be a timeout which can have a very sensible default at development time but due to environmental conditions or changes in third parties that make it invalid later on.

4.2 Lifecycle

The lifecyle of configuration for a system fits into wiring like this

1. Construct
2. PostConstruct
3. Wiring Injection
4. Configuration resolution
5. BeforeConfiguration
6. For each configured component
   1. PreConfigured
   2. Configuration Injection
   3. PostConfigured
   1. AfterConfiguration

Construct

The DI container creates the component

PostConstruct

The DI container invokes methods annotated with `@PostConstruct`

Wiring Injection

The DI container injects all fields marked with `@Inject`

I recommend using javax.inject and not framework annotations to avoid unnecessary dependencies

Lots of frameworks include Spring do configuration at Injection time, but it violates the separation of concerns and fail fast paradigms.

BeforeConfiguration

Hook for action before the configuration system is invoked, perhaps to affect the running of the configuration system

Configuration Resolution

All configuration type information is derived and fails fast for anything that cannot be coerced

All sources are merged and configuration values are resolved

PreConfigured

Methods marked with `@PreConfigured` are invoked to prepare for configured fields, it was added for symmetry, not sure if its that useful

Configuration Injection

Fields marked with `@Configured` are injected by the relevant Coercion

PostConfigured

Methods marked with `@PostConfigured` are invoked to act on the inject configuration

For example you might set a URI and then create a client for it

AfterConfiguration

Once all method hooks are invoked and configured fields are injected

4.3 Usage

4.3.1 @Configured

The configured annotation is the key for the developer. It allows for a simple declaration that a field should be set externally.

In contrast to other frameworks the develoer is not allowed to choose a name for the configuration other than the component and field name.

The reason being that the developer has already reasoned about those names, there is not need to add an indirection.

In my experience you end up with large variation and confusion when the decisions about naming are declared at development time.:

```
@StickyComponent
public class Bean { (1)

  @Configured("Description of the configured field") (2)(3)
  Period cycle; (4)(5)(6)

  @Configured("The timeout for sending email")
  Duration timeoutInSeconds = Duration.ofSeconds(10); (7)(8)

}
```

1. The name of the component is **bean**
2. The configuration annotation marked the field as injected by configuration
3. The description of the configuration, shown when there are failures in configuration or in exports describing the system
4. The type of the configured field, a Coercion is used to convert the Configuration value into this type, in this case a java.util.time.Period,
5. Because this field does not have a value providing configuration at run time is mandatory.
6. The name of the field **cycle**, its combined to define the configuration lookup **bean.cycle** in properties format
7. The name of the timeout includes seconds, this convention is very useful for configurators to know the scale of the value.
8. The default value is defined using plain old java which makes unit test super easy and saves on indirection. This field does not need to be configured at runtime as it has a default

4.3.2 @PostConfigured

Method hook for the developer to execute code as part of the configuration lifecycle:

```
@StickyComponent
public class Bean {

  @Configured("Description of the configured field")
  URL url;
```

(continues on next page)

(continued from previous page)

```
  Client client;

  @PostConfigured (1)
  public void createClient() { (2)
    client = new Client(url); (3)
  }

}
```

1. Post configured annotation declares the hook called **createClient**
2. The result is not used so void is fine, the hook can have any access modifier. Standard practice to to scope it for ease of unit testing.
3. The url is used to create the client object. The url value can be used freely without error checking because

- the framework makes sure that there is a value
- the url is typed so must be a value URL or whatever the type should be
- sophisticated coercions can do extra checked to ensure the validaty as a cross cutting concern saving the developers cognitive load to just the business value they are adding

4.3.3 @AfterConfiguration

Method hook for the developer to execute code after everything is configured:

```
@StickyComponent
public class Bean {

  @Inject
  ConfigurationSytem system;

  @AfterConfiguration
  public void showConfiguration() {
    log.info("configuration {}", system.exportTree());
  }

}
```

4.3.4 @BeforeConfiguration

Method hook for the developer to execute code as part of the configuration lifecycle:

```
@StickyComponent
public class Bean {

  @BeforeConfiguration
  public void example() {
    // contrived example to show off before configuration
  }

}
```

4.3.5 @PreConfigured

Method hook for the developer to execute code as part of the configuration lifecycle:

```
@StickyComponent
public class Bean {

  @PreConfigured (1)
  public void resetSomething() { (2)
    // contrived example to show off pre configuration
  }

}
```

4.3.6 Configuration

The configuration system collects all the defined sources of configuration together, the default configuration sources are

- Application - loads from **META-INF/sticky/application.properties** in the form **bean.field**
- File - loads a property file based on the system property **configuration.path**
- System - load system properties in the form **bean.field**
- Environment - loads environment properties in the form **BEAN_FIELD**
- Default - loads a properties file from **META-INF/sticky/defaults.properties**

The resolution builds the list of values for each field, with different sources having precedence.:

```
Application > File > System > Environment > Default
```

Resolution can be nested for example given some configuration sources

application.properties:

```
application-name=MyApp
```

System properties:

```
environment-colour=blue
```

File:

```
aBean.field=${some.other.value}
some.other.value=found it

anotherBean.veryNested=${colour.${environment-colour}}
```

defaults.properties:

```
colour.blue=azul
colour.yellow=amarillo
```

This will result in aBean.field having value **found it** and anotherBean.veryNested having value **azul**

4.4 Coercions

4.4.1 Overview

The concept is simple a transformation from a string value to a concrete type.

The coercion framework has tooling to derive the type information as a **CoercionTarget**

4.4.2 Standard Coercions

The standard coercions cover most of the standard java libraries

These examples all assume that the configured field is in a class called `Bean`

String

Obviously strings are one to one:

```
@Configured
String value;

bean.value=Some String
```

StringConstructor

Anything with a single string constructor is obviously coercable:

```
@Configured
URL url;

bean.url=https://www.example.com/path
```

Enum

Enums are mapped based on the name of the enum.

I consider Enums to be classes and so camel case them e.g.:

```
public enum Status {
  Good,
  Bad
}

@Configured
Status value;

bean.value=Good
```

Pattern

For regular expressions, super handy with default values in case you missed some cases and need a quick update in production

This example derives the names of the standard coercions:

```
@Configured
Pattern pattern = Pattern.compile(".*");

bean.pattern=*/\(.*\)Coercion.java
```

ValueOfMethod

Anything with a `valueOf(String)` or `of(String)` factory method can be coerced:

```
@Configured
Integer count = 20;

bean.count=10
```

ParseMethod

Anything with a `parse(String)` factory method can be coerced, this includes lots of the time classes:

```
@Configured
Duration length = Duration.ofSeconds(10);

bean.length=PT20s
```

DateTimeFormatter

For building time formatters

InetSocketAddress

Creates inet sockets and validates that they are available

Class

Loads a class from a fully qualifed name:

```
type=com.example.ConcreteBean
```

CharacterSet

Externalise a character set useful for file operations integration with disparate systems

4.4.3 Collection Coercions

Collection coercions are nested in that the elements use other coercions to derive the components.

Map:

```
some.map=a=b,c=d,e=f (1)
alternateSeparator=[;]a=b,c;c=d;e=f (2)

@Configured
Map map;
```

Collection:

```
some.collection=a,c,c,d,e
another=[;]a;b;c;d;e (3)
aThird=a,c,e,b,f
```

Array

anArray=1,2,3,4,5

1. Maps are a list of key=value
2. alternate separators can be defined if you need to use the standard comma separator in the value

3. alternate separators work for all collections
4. obviously strings and numbers and anything coercable is supported

4.4.4 Adding Coercions

Coercions are based around the Coercion interface:

```
public interface Coercion<T> {

  boolean isApplicableTo(CoercionTarget target);

  T coerce(CoercionTarget type, String value);

  boolean hasDefaultValue();

  T getDefaultValue(CoercionTarget target);

  boolean isInverted();

}
```

Most Coercions will extend `AbstractNoDefaultCoercion` that only leaves `isApplicableTo` and `coerce` for implementation

isApplicableTo

> This allows the coercion to decide if the target is something it can coerce
>
> Its a method not just a type as more complicate coercions required more information than just the type e.g. collections have nested potentially parameterized types

coerce

> The core of the coercion it takes a string value and returns a real value
>
> It should always fail for invalid values
>
> The value will never be null but can be blank

hasDefaultValue

> Some coercions can have a system default value
>
> CharacterSet is a good example that has the system defined character set

getDefaultValue

> Derive a default value from the target in the absense of a value

isInverted

> In some cases a coercion will return a fully fledged bean from the Dependency Injection container in which case there is no need for injection to happen to the configured value

4.4.5 Example coercion

This is the Pattern Coercion:

```
package net.stickycode.coercion;

import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

import net.stickycode.stereotype.plugin.StickyExtension;

@StickyExtension (1)
public class PatternCoercion
    extends AbstractNoDefaultCoercion<Pattern> { (2)

  @Override
  public Pattern coerce(CoercionTarget type, String value) (3)
      throws PatternCouldNotBeCoercedException {
    try {
      return Pattern.compile(value);
    }
    catch (PatternSyntaxException e) {
      throw new PatternCouldNotBeCoercedException(e, value); (4)
    }
  }

  @Override
  public boolean isApplicableTo(CoercionTarget type) {
    return type.getType().isAssignableFrom(Pattern.class); (5)
  }

}
```

1. The framework is made of components and a coercion is just a component and uses DI, the `@StickyExtension` stereotype allows framework components to be initialised first
2. Extending the abstract no default coercion to keep implementation simple and consistent
3. The standard method returns a new Pattern
4. I always use explicit exceptions as they are more meaningful that generic expections with odd messages
5. The coercion is only valid for Pattern types

CHAPTER 5

Mockwire

Mockwire is a tool to remove wiring boiler plate wiring from tests, it provides a convention to wire tests using the same dependency injection system you use in production, i.e. Test it like you plan to run it.

It also simplifies the use of mocking for isolating code under test. Think empirical analysis and controlled variables.

5.1 Introduction

Mockwire is a tool that takes care of the boiler plate of wiring up your tests, that means that all you see in your tests is

1. a simple manifest of what to test
2. mocking of controlled variables
3. assertions stating the assumptions to validate environment

Mockwire uses the test class itself to define which beans are mocks and which real by the use of some annotations **@UnderTest**, **@Controlled**, **@Uncontrolled**

Implementations of Mockwire build a registry of these beans and mocks and wire them by type. You can use Spring 2, 3, and 5 and guice 3 and 4.

5.2 Why use mockwire

Many projects rely heavily on Dependency Injection to manage application assembly, because its a powerful tool. However many setups suffer from too much boiler plate when defining tests to verify the assembly.

Ideally we would always test our code just as it gets wired in production, if we write boiler plate code in our tests and use dependency injection in production environments we aren't really testing our code as it is run, which is somewhat pointless.

Mockwire just leverages the standard DI containers to build that assembly just as if the code under test was wire up properly. I call the assemblies isolated because its a well defined subset of the final application.

5.3 Testing Configuration

Too often testing configuration is neglected or just not possible. MockwireConfigured makes it very easy to test different configuration

5.4 Including Mockwire in you project

Add the appropriate MockwireDependency to your project to get started

To use Mockwire with Junit

I primarily use Junit because I use Infinitest and Junit support in eclipse is a tad better than Testng. If you don't use junit See UsingMockwireWithoutJunit

Mockwire provides a junit4 runner: MockwireRunner to easily run your test all wired up.

@RunWith(MockwireRunner.class) public class UnitTest {

5.4.1 Test Manifest

@UnderTest, @Controlled and @Uncontrolled are used to define the types that end up in the wired test environment.

5.5 @UnderTest

Creates a concrete instance of the marked bean in the DI context and inject it

5.6 @Controlled

Create a Mock of the marked type in the DI context and inject it

5.7 @Uncontrolled

Create a concrete instance of the marked bean and *DO NOT* inject it

5.7.1 Running Mockwire

5.8 @MockwireRunner

Annotated

5.9 @MockwireContainment

Mockwire scans the test class and identitifies the code you wish to test "@UnderTest",

e.g. SomeConcreteClass in being tested so we need to bless a real instance of it. How the class is turned into a real instance is left up to the DI tool in use. @UnderTest SomeConcreteClass codeToTest;

And controlled dependencies of the code to test,

e.g. SomeConcreteClass requires a SomeInterface, this requirement is defined by @Inject @Controlled SomeInterface thatSomeConcreteClassNeeds; How the class actually gets mocked is left up to the Mocking implementation, I would highly recommend Mockito, but will supply bindings to other mocking libraries on request.

For assertions and mocking See UnitTestingComposition which defines the libraries used and what they do.

Examples MockwireHelloWorld MockwireDecentExample MockwireExampleProject Containment Once you have confirmed that a fully isolated chunk of code is tested you want to integrated it. Thats where @MockwireContainment comes in. In the simplest case it defines the containment of a test as not just the manifest from the Test class but also the result of scanning the package that the Test class exists in for components.

See Component Oriented Programming

Any class you want to be defined in scanned context gets marked with a @StickyComponent (or some other Component annotation).

Given some interfaces and implementations all in the same package:

```
package net.stickycode.example.containment;

public interface Service {
  boolean doIt();
}

public interface Other {
  boolean it();
}

@StickyComponent
public class AlwaysTrueOther {
  public boolean it() {
    return true;
  }
}

@StickyComponent
public class ServiceImpl {

  @Inject
  private Other anOther;

  public boolean doIt() {
    anOther.it();
  }
}
```

For containment we define an integration test and Inject the code we wish to test:

```
@RunWith(MockwireRunner.class) @MockwireContainment public class␣
↪ServiceImplIntegrationTest {

@Inject Service impl;

@Test public void coherency() { assertThat(impl.doIt()).isTrue(); } }
```

Specifying the package(s) to scan:

```
package net.stickycode.example.containment.main;
public interface Service { boolean doIt(); }

package net.stickycode.example.containment.other;
public interface Other { boolean it(); }

package net.stickycode.example.containment.other;

@StickyComponent
public class AlwaysTrueOther { public boolean it() { return true; } }

package net.stickycode.example.containment.main;

@StickyComponent
public class ServiceImpl {

  @Inject private Other anOther;

public boolean doIt() { anOther.it(); }
}
```

The package(s) can be defined as a parameter to the MockwireContainment annotation:

```
package net.stickycode.example.containment.main;

@RunWith(MockwireRunner.class)
@MockwireContainment("/net/stickycode/example/containment/main")
public class ServiceImplIntegrationTest {

  @UnderTest
  ServiceImpl impl;

  @Test
  public void coherency() { assertThat(impl.doIt()).isTrue(); }
}
```

More than one package can be scanned with:

```
@MockwireContainment("/net/stickycode/example/containment/main", "/some/other/package)
```

If you prefer the package syntax:

```
@MockwireContainment("net.stickycode.example.containment.main")
```

MockwireDependency

Example

5.9.1 Example

5.9.1.1 Example 1

CHAPTER 6

Scheduled

6.1 @Scheduled

6.2 Schedulers

6.3 Background processing

CHAPTER 7

Plugins

Bounds Maven Plugin

A maven plugin to update the lower bounds of ranges to reduce metadata downloads

Happy Maven Plugin

Shifty Maven Plugin

A maven plugin to download and maybe unpack a set of artifacts

Wait Maven Plugin

A maven plugin to halt the build process and wait for the user

7.1 Bounds Maven Plugin

A maven plugin to update the lower bounds of ranges to reduce metadata downloads

7.1.1 Example delta

When you run bounds:update for a project that contains this:

```
<plugin>
 <groupId>net.stickycode.composite</groupId>
 <artifactId>sticky-composite-logging-api</artifactId>
 <version>[2.3,3)</version>
</plugin>
```

and the latest release of sticky-composite-logging-api is 2.4, then you will end up with:

```
<plugin>
  <groupId>net.stickycode.composite</groupId>
  <artifactId>sticky-composite-logging-api</artifactId>
```

(continues on next page)

(continued from previous page)

```
  <version>[2.4,3)</version>
 </plugin>
```

7.1.2 Usage

The plugin is in maven central so it should 'Just Work'.

Run the plugin from your Apache Maven project directory:

```
mvn net.stickycode.plugins:bounds-maven-plugin:2.2:update
```

And your version ranges will have there lower bound updated to the latest released artifact version.

If you want to include any SNAPSHOT references when calculating the lower bound, set the'includeSnapshots' property:

```
-DincludeSnapshots
```

when calling *mvn*.

7.1.3 Update bounds during release

To update the bounds during release you can do this:

```
<pluginManagement>
 <plugins>

  <plugin>
   <groupId>net.stickycode.plugins</groupId>
   <artifactId>bounds-maven-plugin</artifactId>
   <version>3.3</version>
  </plugin>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-release-plugin</artifactId>
   <version>2.2.2</version>
   <configuration>
     <preparationGoals>bounds:update enforcer:enforce clean verify</preparationGoals>
   </configuration>
  </plugin>
 </plugins>
</pluginManagement>
```

7.1.4 Line endings

You can specify the line separator used like so:

```
<plugin>
 <groupId>net.stickycode.plugins</groupId>
 <artifactId>bounds-maven-plugin</artifactId>
 <version>3.3</version>
 <configuration>
```

(continues on next page)

(continued from previous page)

```
  <lineSeparator>Unix</lineSeparator>
 </configuration>
</plugin>
```

7.1.5 Extract Current Version

To get the current version of a library from a range use bounds:current-version, this will set the property *stickyCoercion.version* to the right 2.x version:

```
<plugin>
  <plugin>
    <groupId>net.stickycode.plugins</groupId>
    <artifactId>bounds-maven-plugin</artifactId>
    <version>3.3</version>
    <executions>
      <execution>
        <goals>
          <goal>current-version</goal>
        </goals>
        <configuration>
          <stickyCoercion.version>net.stickycode:sticky-coercion:[2,3]</
↪stickyCoercion.version>
        </configuration>
      </execution>
    </execution>
  </plugin>
</plugin>
```

7.1.6 Releases

Release 3.3

- dependencies with classifiers were being ignored incorrectly

Release 3.2

- support for setting a property to the highest version in a range

Release 2.6

- added support for dependencyManagement - although I would suggest you never ever us it
- added support for version defined as properties - although again I would suggest you don't do that
- allow the line separator on rewrite to be configured (Mac, Unix Windows), useful when you define the line ending in your SCM and need re-generated poms to match

7.2 Happy Maven Plugin

A maven plugin to collect application metadata and validate during delivery acceptance testing

When you have an environment made up of a number of applications having a quick derived test to validate the correct versions of applications are deployed is important

7.2.1 Usage

7.2.1.1 Collecting the metadata

During the build process this will record the application version and context path for later validation:

```
<plugin>
  <groupId>net.stickycode.plugins</groupId>
  <artifactId>happy-maven-plugin</artifactId>
  <version>1.7</version>
  <executions>

    <execution>
      <id>collect-application-version</id>

      <goals>
        <goal>collect</goal> (1)
      </goals>

      <phase>compile</phase>

      <configuration>
        <contextPath>${context.path}/</contextPath> (2)
      </configuration>
    </execution>

  </executions>
</plugin>
```

1. Invoke the collect goal
2. The path that will be validated

The record is kept in the jar file of the application - see https://github.com/StickySource/happy-maven-plugin/issues/1

7.2.1.2 Using the medadata

This will collect all the version in the resolved classpath for the project and check the version url for them:

```
<plugin>
  <groupId>net.stickycode.plugins</groupId>
  <artifactId>happy-maven-plugin</artifactId>
  <version>1.7</version>

  <executions>

    <execution>
      <id>validate-application-version</id>

      <goals>
        <goal>validate</goal>
      </goals>

      <phase>integration-test</phase>

      <configuration>
        <targetDomain>https://demo.exmaple.com</targetDomain>
```

(continues on next page)

(continued from previous page)

```
        <retryDurationSeconds>120</retryDurationSeconds>
        <retryPeriodSeconds>3</retryPeriodSeconds>
      </configuration>
    </execution>

  </executions>
</plugin>
```

For example * If for example we had an application *demo* * with a context path *users* at version 1.7, * then a GET request would be made to *https://demo.example.com/users/version* * expecting demo-1.7 to be returned * the request will be retried every 3 seconds * the validation will fail if a successful call has not happened after 120 seconds

7.3 Shifty Maven Plugin

A maven plugin to download and maybe unpack a set of artifacts

Similar to the dependency plugin but

- efficiently leveraging metadata to be fast
- using version ranges for fluidity
- using parallel execution where possible

7.3.1 Usage

Fetching a dependency, this will download sticky-coercion-2.7.jar to target/shifty/:

```
<plugin>
   <groupId>net.stickycode.plugins</groupId>
   <artifactId>shifty-maven-plugin</artifactId>
   <version>1.2</version>
   <executions>
     <execution>
       <id>test</id>
       <phase>validate</phase>
       <goals>
         <goal>fetch</goal>
       </goals>
       <configuration>
         <artifacts>
           <artifact>net.stickycode:sticky-coercion:[2.1,3)</artifact>
         </artifacts>
       </configuration>
     </execution>
   </executions>
 </plugin>
```

Unpack a zip/jar into target/shifty:

```
<plugin>
   <groupId>net.stickycode.plugins</groupId>
   <artifactId>shifty-maven-plugin</artifactId>
   <version>1.2</version>
```

(continues on next page)

(continued from previous page)

```
   <executions>
     <execution>
       <id>test</id>
       <phase>validate</phase>
       <goals>
         <goal>fetch</goal>
       </goals>
       <configuration>
         <unpack>true</unpack>
         <artifacts>
           <artifact>net.stickycode:sticky-coercion:[2.1,3):sources:jar</artifact>
         </artifacts>
       </configuration>
     </execution>
   </executions>
 </plugin>
```

Unpack a zip/jar into target/other:

```
<plugin>
   <groupId>net.stickycode.plugins</groupId>
   <artifactId>shifty-maven-plugin</artifactId>
   <version>1.2</version>
   <executions>
     <execution>
       <id>test</id>
       <phase>validate</phase>
       <goals>
         <goal>fetch</goal>
       </goals>
       <configuration>
         <outputDirectory>${project.build.directory}/other</outputDirectory>
         <unpack>true</unpack>
         <artifacts>
           <artifact>net.stickycode:sticky-coercion:[2.1,3):sources:jar</artifact>
         </artifacts>
       </configuration>
     </execution>
   </executions>
 </plugin>
```

7.3.2 How it works

Its pretty straight forward, for gav net.stickycode:sticky-coercion:[2.1,3):

- [2.1,3) is first resolved, in this case to 2.7 via metadata
- and the property sticky-coercion.version is set to 2.7
- the artifact sticky-coercion-2.7 is resolved and downloaded if needed
- the artifact is copied into the output directory

Note that a Maven property is set in case you need to reference the artifact after its resolved.

7.4 Wait Maven Plugin

A maven plugin to halt the build process and wait for the user

This is useful to pause after integration test are run before they are verified to interact with the system manually

Add the plugin execution like so:

```
<plugin>
  <groupId>net.stickycode.plugins</groupId>
  <artifactId>wait-maven-plugin</artifactId>
  <version>1.5</version>
  <executions>
    <execution>
      <phase>post-integration-test</phase>
      <goals>
        <goal>wait</goal>
      </goals>
      <configuration>
        <promptMessage>${project.name} is now running on http://localhost:${port}$
↪{context}/,
        I'll wait while you do stuff, when you are finished let me know by hitting␣
↪enter or ctrl-c
        </promptMessage>
      </configuration>
    </execution>
  </executions>
</plugin>
```

And then when you are running a test running:

```
mvn clean verify -Dwait=true
```

If you prefer to use a profile:

```
<profiles>
  <profile>
    <id>webapp-interactive</id>
    <build>
      <properties>
        <wait>true</wait>
      </properties>
    </build>
  </profile>
</profiles>
```

CHAPTER 8

Indices and tables

- search