Groovy DSLs in 2016

by Cédric Champeau (@CedricChampeau)

Who am I

```
speaker {
    name 'Cédric Champeau'
    company 'Gradle Inc'
    oss 'Apache Groovy committer',
    successes (['Static type checker',
                'Static compilation',
                'Traits',
                'Markup template engine',
                'DSLs'])
        failures Stream.of(bugs),
        twitter '@CedricChampeau',
        github 'melix',
        extraDescription '''Groovy in Action 2 co-author
Misc OSS contribs (Gradle plugins, deck2pdf, jlangdetect, ...)'''
}
```

Domain Specific Languages

- Focused
- Readable
- Practical
- (usually) embeddable
- Examples: SQL, HTML, XSLT, Ant, ...

Disclaimer

This is an opiniated talk about how a DSL designed with Apache Groovy should look like.

Apache Groovy for DSLs

- Concise, clean syntax
- Supports scripting
- Supports metaprogramming
- Embeddable
- Mature tooling: Eclipse, IntelliJ, Netbeans...

Some old Groovy DSLs

Groovy SQL

```
sql.execute "insert into PROJECT (id, name, url) values ($map.id, $map.name, $map
```

Grails dynamic finders

```
def persons = Person.findByLastName('Stark')
assert persons.findAll { it.alive }.isEmpty()
```

Spring Boot

```
@EnableAutoConfiguration
@ComponentScan("foo.bar")
class Application {
    static void main(String[] args){
        new SpringApplication(Application).run(args)
    }
}
```

- Not strictly speaking a DSL
- But can be converted into one

Gradle task execution

```
task(hello) << {
    println "hello"
}
```

vs

```
task(hello) {
    println "hello"
}
```

Some thoughts

- removing semicolons is not designing a DSL
- removing parenthesis is not designing a DSL
- user experience is important
- consistency is important
- Try to be idiomatic

Modern Apache Groovy DSLs

Spock

```
setup:
def map = new HashMap()

when:
map.put(null, "elem")

then:
notThrown(NullPointerException)
```

Grails 3 where queries

```
assert Person.findAll {
    lastName == 'Stark' && alive
}.isEmpty()
```

Gradle new model

```
model {
    components {
        shared(CustomLibrary) {
            javaVersions 6, 7
        }
        main(JvmLibrarySpec) {
            targetPlatform 'java6'
            targetPlatform 'java7'
            sources {
                java {
                    dependencies {
                        library 'shared'
                    }
                }
            }
        }
    }
}
```

Ratpack

```
ratpack {
    handlers {
        get {
            render "Hello World!"
        }
        get(":name") {
            render "Hello $pathTokens.name!"
        }
    }
}
```

Jenkins Job DSL

```
job {
    using 'TMPL-test'
    name 'PROJ-integ-tests'
    scm {
        git(gitUrl)
    }
    triggers {
        cron('15 1,13 * * *')
    }
    steps {
        maven('-e clean integTest')
    }
}
```

MarkupTemplateEngine

```
modelTypes = {
   List<String> persons
}

html {
    body {
        ul {
            persons.each { p ->
                li p.name
            }
        }
    }
}
```

Implementing modern DSLs

The tools

- Closures with support annotations (`@DelegatesTo`, ...)
- Compilation customizers
- AST transformations
- Type checking extensions
- Groovy Shell / Groovy Console

Closures

- Still at the core of most DSLs
- `delegate` is very important:

```
['Paris', 'Washington', 'Berlin'].collect { it.length() == 5 }
```

- do we really need `it`?

Setting the delegate

```
class  HelperExtension {
    public static <T,U> List<U> myCollect(List<T> items, Closure<U> action) {
        def clone = action.clone()
        clone.resolveStrategy = Closure.DELEGATE_FIRST
        def result = []
        items.each {
            clone.delegate = it
            result << clone()
        }
        result
    }
}

HelperExtension.myCollect(['Paris', 'Washington', 'Berlin']) {
    length() == 5
}
```

Convert it to an extension module

- `META-INF`
  - `services`
    - `org.codehaus.groovy.runtime.ExtensionModule`

```
moduleName=My extension module
moduleVersion=1.0
extensionClasses=path.to.HelperExtension
```

Convert it to an extension module

- Consume it as if it was a regular Groovy method

```
['Paris', 'Washington', 'Berlin'].myCollect {
    length() == 5
}
```

Declare the delegate type

- Best IDE support
- Only way to have static type checking

```
public static <T,U> List<U> myCollect(
    List<T> items,
    @DelegatesTo(FirstParam.FirstGenericType)
    Closure<U> action) {
...
}
```

Removing ceremony

- Is your DSL self-contained?
- If so
  - Try to remove explicit imports
  - Avoid usage of the `new` keyword
  - Avoid usage of annotations
  - Embrace SAM types

SAM what?

This is ugly:

```
handle(new Handler() {
    @Override
    void handle(String message) {
       println message
    }
})
```

SAM what?

This is cool:

```
handle {
   println message
}
```

SAM type coercion works for both interfaces and abstract classes.

Compilation customizers

```
class WebServer {
   static void serve(@DelegatesTo(ServerSpec) Closure cl) {
      // ...
   }
}
```

Compilation customizers

```
def importCustomizer = new ImportCustomizer()
importCustomizer.addStaticStars 'com.acme.WebServer'

def configuration = new CompilerConfiguration()
configuration.addCompilationCustomizers(importCustomizer)

def shell = new GroovyShell(configuration)
shell.evaluate '''
serve {
   port 80
   get('/foo') { ... }
}
'''
```

Compilation customizers

- `ImportCustomizer`: automatically add imports to your scripts
- `ASTTransformationCustomizer`: automatically apply AST transformations to your scripts
- `SecureASTCustomizer`: restrict the grammar of the language
- `SourceAwareCustomizer`: apply customizers based on the source file
- See docs for customizers

Avoiding imperative style

```
class WebServer {
   static void serve(@DelegatesTo(ServerSpec) Closure cl) {
       def spec = new ServerSpec()
       cl.delegate = spec
       cl.resolveStrategy = 'DELEGATE_FIRST'
       cl()
       def runner = new Runner()
       runner.execute(spec)
   }
}
```

Avoiding imperative style

```
class ServerSpec {
   int port
   void port(int port) { this.port = port }
   void get(String path, @DelegatesTo(HandlerSpec) Closure spec) { ... }
}
```

Avoiding imperative style

- Use the `ServerSpec` style above
- The closure should configure the model
- Execution can be deferred

User-friendly immutable builders

- Example from Gradle

```
java {
    dependencies {
        library 'foo'
        project 'bar' library 'main'
    }
}
```

- `SourceSet` has a `DependencySpecContainer`
- `DependencySpecContainer` defines `project` and `library` methods
- as well as a `getDependencies` method returning an immutable view

User-friendly immutable builders

```
public interface DependencySpecContainer {
    DependencySpecBuilder project(String path);

    DependencySpecBuilder library(String name);

    Collection<DependencySpec> getDependencies();
}
```

Immutable builders

- A dependency spec is by default immutable

```
public interface DependencySpec {

    @Nullable
    String getProjectPath();

    @Nullable
    String getLibraryName();
}
```

Immutable builders

- The builder specializes the spec interface

```
public interface DependencySpecBuilder extends DependencySpec {

    DependencySpecBuilder project(String path);


    DependencySpecBuilder library(String name);

    DependencySpec build();
}
```

Immutable builders

- Concrete implementation provides the builder

```
public class DefaultDependencySpec implements DependencySpec {
    private final String projectPath;
    private final String libraryName;

    // ...

    public static class Builder implements DependencySpecBuilder {
        private String projectPath;
        private String libraryName;

        @Override
        public DependencySpecBuilder project(String path) {
            projectPath = path;
            return this;
        }

        @Override
        public DependencySpecBuilder library(String name) {
            libraryName = name;
```

Immutable builders

- And the container stores a list of builders

```
public class DefaultDependencySpecContainer implements DependencySpecContainer {

    private final List<DefaultDependencySpec.Builder> builders = new LinkedList<

    @Override
    public DependencySpecBuilder project(final String path) {
        return doCreate(new Action<DefaultDependencySpec.Builder>() {
            @Override
            public void execute(DefaultDependencySpec.Builder builder) {
                builder.project(path);
            }
        });
    }

    @Override
    public DependencySpecBuilder library(final String name) {
        return doCreate(new Action<DefaultDependencySpec.Builder>() {
            @Override
            public void execute(DefaultDependencySpec.Builder builder) {
```

Type checking extensions

Goals

- Provide early feedback to the user
- Type safety
- Help the compiler understand your DSL

Type checking extensions API

- Event-based API
- React to events such as undefined variable or method not found
- Developer instructs the type checker what to do

```
methodNotFound { receiver, name, argList, argTypes, call ->
    if (receiver==classNodeFor(String)
            && name=='longueur'
            && argList.size()==0) {
        handled = true
        return newMethod('longueur', classNodeFor(String))
    }
}
```

MarkupTemplateEngine example

- Given the following template

```
pages.each { page ->
    p("Page title: $page.title")
    p(page.text)
}
```

- How do you make sure that `pages` is a valid model type?
- How do you notify the user that `page` doesn't have a `text` property?
- How to make it fast?

Solution

- Declare the model types

```
modelTypes = {
    List<Page> pages
}

pages.each { page ->
    p("Page title: $page.title")
    p(page.text)
}
```

- Implement a type checking extension

MarkupTemplateEngine extension

- Recognizes unresolved method calls
  - converts them into direct methodMissing calls
- Recognizes unresolved variables
  - checks if they are defined in the binding
  - if yes, instructs the type checker what the type is

MarkupTemplateEngine extension

- Applies `@CompileStatic` transparently
- Performs post-type checking transformations
  - Don’t do this at home!

(Optional) @ClosureParams

- For type checking/static compilation

```
['a','b','c'].eachWithIndex { str, idx ->
  ...
}
```

(Optional) @ClosureParams

```
public static <T> Collection<T> eachWithIndex(
   Collection<T> self,
   @ClosureParams(value=FromString.class, options="T,Integer")
   Closure closure) {
   ...
}
```

Check out the documentation for more details.

What we learnt

- Leverage the lean syntax of Groovy
- Scoping improves readability
- Use the delegate
- Use `@DelegatesTo` and `@ClosureParams` for IDE/type checker support
- Use imperative style as last resort
- Help yourself (builders, immutable datastructures, ...)

Questions

We're hiring!

http://gradle.org/gradle-jobs/

Gradle

Build Happiness.

Thank you!

- Slides and code : https://github.com/melix/s2gx-groovy-dsls
- Groovy documentation : http://groovy-lang.org/documentation.html
- Follow me: @CedricChampeau