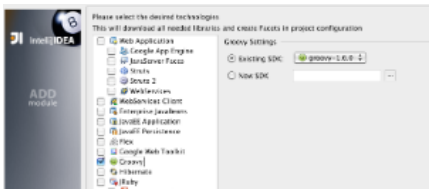
Scripting IDE for DSL awareness

GroovyDSL scripting framework

GroovyDSL is a framework with a domain-specific language designed to define the behaviour of end-user DSLs as script files which are executed by the IDE on the fly, bringing new reference resolution and code completion logic into the scope of a project. GroovyDSL relies on the Program Structure Interface (PSI), a set of internal classes and interfaces of IntelliJ IDEA, which allow all programming languages to be described in a uniform way. Due to Groovy's meta-programming capabilities, the developer of DSL descriptions in GroovyDSL need not be aware of how PSI works. All interoperation with PSI is covered by calls to methods and properties of GroovyDSL scripts.

Writing a simple GroovyDSL script

Let's write a simple GroovyDSL script to add a piece of new behavior to an existing class. If you create a new Java project in IntelliJ IDEA, make sure, that you attach Groovy facet to it. For existing projects Groovy facet may be adjusted via Project Structure settings. Also don't forget to attach Java SDK to your project.

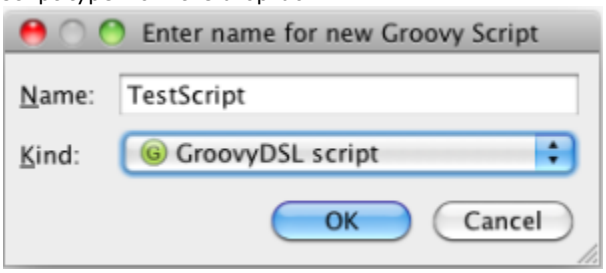


Suppose, we have the following script "DayCounter.groovy", which adds `daysFromNow' property to the Integer class.

```
Integer.metaClass.getDaysFromNow = { ->
  Calendar today = Calendar.instance
  today.add(Calendar.DAY_OF_MONTH, delegate)
  today.time
}

println(5.daysFromNow)
```

As one could notice, an invocation of the appropriate property is highlighted as an unresolved call. To fix it, we write a simple GrooyDSL script. To create a GroovyDSL script, invoke the `new Groovy Script' action in project view and choose GroovyDSL script type from the drop-down.



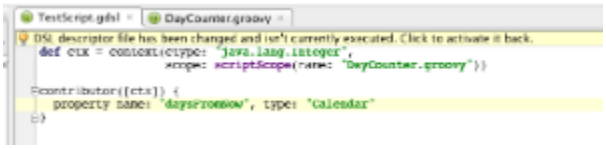
In the body of GroovyDSL script write the following code

```
def ctx = context(ctype: "java.lang.Integer",
                  scope: scriptScope(name: "DayCounter.groovy"))

contributor(ctx) {
  property name: "daysFromNow", type: "Calendar"
}
```

First two lines describe a context, to which we are going to add a new behaviour. It may be read as "augment class java.lang.Integer" in script with the name "DayCounter.groovy".
Next two lines describe a contributor, which is in charge for the new behaviour. It takes a context `ctx' and in a passed closure adds new property with name "daysFromNow" and type "Calendar" to the target class, namely "java.lang.Integer" which is known from the passed context.

After finishing editing a GroovyDSL script one can notice a yellow band on top of an editor screen, informing, that modified GroovyDSL script should be reloaded. After clicking on it an actual script will be processed and activated to make the IDE aware about new behaviour.



Switching back to the DayCounter script we'll see that `daysFromNow' is highlighted now as a class property, and, moreover, it's available for completion. This works pretty like Dynamic Properties feature, but GroovyDSL scripting gives much larger freedom in defining context. For example, by script we just have written, `daysFromNow' property will be available only on top of DayCounter scripts, not in others, whereas being defined with dynamic properties it would be available in all Groovy files in project. Of course, there are various scopes to define contexts of different granularity.

More examples of GroovyDSL scripts

GroovyDSL internal language relies on the IntelliJ IDEA open API, namely the set of Program Structure Interfaces (PSI). All GroovyDSL functions and properties are nothing but wrappers around existing PSI interfaces. To get an adequate code assistance with code completion it's strongly recommended to attach openapi.jar from $IDEA_DIR/lib folder as a library to an actual project. GroovyDSL scripts will be working fine even without it being attached, but in this case some references in GroovyDSL scripts will be marked as possibly unresolved or untyped. In light of said above, all standard PSI methods are available in GroovyDSL as well as synthetic ones, described below.

Next example shows, how to describe generic methods in terms of GroovyDSL for class Position.

```
class Position {
  def Left
  def Right
  def Middle
  def methodMissing(String name, args) {
    // implementation of dynamic method synthesis
  }
}

Position pos = new Position()
```

The GroovyDSL script below describes injection of dynamic methods `findAll*Positions' for all fields of Position class.

```
def ctx2 = context(ctype: "Position")
contributor (ctx2) {
  // Add generic methods
  classType?.fields?.each {
    method name: "findAll${it.name}Positions",
          type: 'java.util.Collection'
  }
}
```

It's important to remark, that classType is a predefined property of a closure, passed as a parameter to the contributor() meth od and it's available only inside of it, whereas fields is an invocation of the getFields() method of a PsiClass class. Here we are also using Groovy's higher-order functions to iterate by method `each' through the collection of fetched fields.

One can use GroovyDSL to describe delegation-based semantics of Groovy programs. The following code gives a definition of the class Runner, whose method run() takes first parameters, namely, any object obj and a closure cl. It redefines a default delegate of a given closure and assigns obj to its delegate property. When passing a closure to the method run() as a parameter, according to Groovy's rules of reference resolution it is allowed to invoke unqualified method of its first parameter inside of a body of closure

```
class MyDelegate {
  def saySomething(String str) {
    println str
  }
}

class Runner {
  def run(obj, Closure cl) {
    cl.delegate = obj
    cl()
  }
}

def runner = new Runner()
runner.run(new MyDelegate()) {
  saySomething("hello!")
}
```

This delegation-dependent logic may be captured by the following GroovyDSL code:

```
def ctx = context(scope: closureScope())

contributor(ctx, {
  def call = enclosingCall("run")
  if (call) {
    def method = call.bind()
    def clazz = method?.containingClass
    if ("Runner".equals(clazz?.qualName)) {
      delegatesTo(call.arguments[0])
    }
  }
})
```

Several predefined GroovyDSL methods are used in this code fragment. enclosingCall(methodName) call returns a method invocation expression of a given context with a name, matching methodName or null otherwise. delegatesTo method takes an expression and adds all member of its type to the augmented class reference. In the given example this is enclosing closure, since the context's scope is defined as closureScope().

Contexts and contributors

In this section we give an overview of two main GroovyDSL concepts: contexts and contributors
In GroovyDSL context is an entity which is supposed to give an answer to the question ``Where?''. In other words, defining context one describes a place, where some behavior will be available without any reference to the kind of this behavior. The context may be considered as a groovy program with a ``hole'', which stays for a method or property invocation. Contexts are first-order citizens in GroovyDSL, so they may be stored in local script variables and passed as parameters.

```
def ctx = context ctype: <ctype>,
            filetypes: [<file_ext>*],
            scope: <scope>
```

Context definition in GroovyDSL is nothing but an invocation of a predefined method context with optional arguments, each of which may be omitted. Without any arguments given invocation of context method returns the definition of an empty context, which has a meaning ``everywhere''. Below we describe possible values of optional parameters (in angle brackets).

Reference class type <ctype> A string, representing a fully qualified name of a class type to be augmented with the new behavior. This change affects both qualified and unqualified calls to methods and properties of a given class. In the case of an unqualified invocation, <ctype> is thought as a type of this reference. If this argument is omitted, java.lang.Object class type is taken as a type to be augmented.
Supported file types <filetypes>* A possible empty list of comma-separated file extensions passed as strings or GString instances with an opening dot or without it. Since some dynamic behavior may be scoped only in specific groovy-like files with a specific file extension, for instance, Gradle scripts, providing an explicit extension list gives a way to describe such cases.
Script type <scriptType> A string denoting the script type ID, e.g. 'gant', 'gradle', 'default'. Differs from <fileTypes> in that it doesn't depend on the extension. For example, the Gant scripts in Grails usually have groovy extension. Since IDEA X
Path pattern <pathRegexp> A regexp that should match the path to the Groovy file where your script contributes to. The path always uses forward / slashes. Since IDEA X
Context scope <scope> A scope of a current context. If no scope passed to a context, it is assumed to capture all possible places, where other conditions are applicable. It may be of three different kinds:

1. scriptScope(name : <name>) The scope of a behavior defined in this context is available in a script body as well as in its inner closures, for instance, ones which are passed as method arguments. scriptScope() method takes one optional parameter <name> which should be a string, GString or a regular expression. In a presence of this argument appropriate script names will be matched against it. By default the value of this parameter is an all-matching regular expression.

2. classScope(name : <name>) Restricts the scope of an actual behavior to the class with some particular properties, such as <name> (rules for class naming filter are same as for ScriptScope).

3. closureScope(isArg: Boolean) This scope describes Groovy's closure blocks. There is an important case of closure, passed as a parameter to a method, replacing its delegate, which allows to invoke callee's unqualified methods in a body of closure. The typical example of such a behavior is identity() method of groovy.lang.GroovyObject class. This case may be captured by passing an optional named parameter isArg to the closure

Contributors are entities which consume contexts as initialization parameters and answer to the question ``What?''. Contributors provide new properties and methods according to given contexts. We provide a set of utility methods to augment existing types with new behavior, which may be used in a contributor application as regular Groovy methods. New contributor may be defined by an invocation of a method contributor(), taking two parameters:
1. A list of contexts to be used to contribute a new behavior
2. Zero-argument closure, adding new behavior to the reference expression ``in the focus'' of a context

The code fragment below shows the definition of a contributor, taking a list of three contexts, defined earlier, namely ctx1 and ctx2 and one closure, which adds unconditionally the method foo with a parameter of type java.lang.String and return type int to the target expression of a context.

```
contributor [ctx1, ctx2], {
 method name: "foo",
       params: [s: "java.lang.String"],
       type: "int"
}
```

Extending GroovyDSL.

The core GroovyDSL functionality is concentrated in a body of closures, passed to the contributor() method as a second parameter. All invocations of functions and properties inside of it are calls to wrappers around internal IntelliJ IDEA's PSI. There are two kinds of such wrappers. The first kind is unqualified ones, such as findClass(). The second kind is methods, added to the original PSI classes via the mix-in categories mechanism. For example, methods property is added externally by a GroovyDSL environment to the PsiClass class.

Both of these sets of functions may be extended by new ones using a mechanism of plugins for IntelliJ IDEA. To provide new unqualified methods one should provide a class, implementing the GdslMembersProvider interface, and register it by a plugin descriptor. All methods of such a newly added components will be used as possible delegates for unqualified method calls. All methods of GdslMembersProvider implementation must take an instance of GdslMembersHolderConsumer as a last parameter to get an information about a context by it.

```
public class GroovyDslDefaultMembers implements GdslMembersProvider {

  @Nullable
  public PsiClass findClass(String fqn, GdslMembersHolderConsumer consumer) {
    final JavaPsiFacade facade = JavaPsiFacade.getInstance(consumer.getProject());
    final PsiClass clazz = facade.findClass(fqn, GlobalSearchScope.allScope(consumer.getProject()));
    return clazz;
  }

...

}
```

New methods and properties may be added to PSI by implementing the PsiEnhancerCategory interface. Its registered implementations will be used as categories and ``wrapped around'' a body of a contributor's closure in a moment of its execution.

```
public class PsiClassCategory implements PsiEnhancerCategory {

  @Nullable
  public static String getQualName(PsiClass clazz) {
    return clazz.getQualifiedName();
  }
// Other category methods
...
}
```

Describing GroovyDSL internal language in its own terms

Since GroovyDSL language is freely extensible in terms of IDEA PSI, here we list all GroovyDSL methods, available to invoke on the top-level of scripts or in the body of a closure, passes to contributor as a parameter. The code below is an actual script,

describing the semantics of GroovyDSL build-up atop of IDEA PSI.

```
def gdslScriptContext = context(scope: scriptScope(), filetypes: ["gdsl"])

contributor([gdslScriptContext]) {
  method name: "context", params: [args: [:]], type: "java.lang.Object"
  method name: "contributor", params: [contexts: "java.util.List", body: {}], type: void

  // scopes
  property name: "closureScope", type: {}
  property name: "scriptScope", type: {}
}

def contributorBody = context(scope: closureScope(isArg: true))

contributor([contributorBody]) {
  if (enclosingCall("contributor")) {
    method name: "method", type: "void", params: [args: [
            parameter(name:'name', type:String.name, doc:'Method name'),
            parameter(name:'params', type:Map.name, doc:'A map representing method parameters'),
            parameter(name:'namedParams', type:Object.name, doc:'''A list representing method named
parameters.<br>
Its elements should be calls to <code>parameter</code> method.'''),
            parameter(name:'type', type:Object.name, doc:'Return type name of the method'),
            parameter(name:'doc', type:String.name, doc:'Method documentation text'),
    ]], doc:'Describe a DSL method'
    method name: "property", type: "void", params: [args: [
            parameter(name:'name', type:String.name, doc:'Property name'),
            parameter(name:'type', type:Object.name, doc:'Property type name'),
            parameter(name:'doc', type:String.name, doc:'Property documentation text'),
    ]], doc:'Describe a DSL property'
    method name: "parameter", type: "Parameter", params: [args: [
            parameter(name:'name', type:String.name, doc:'Parameter name'),
            parameter(name:'type', type:Object.name, doc:'Parameter type name'),
            parameter(name:'doc', type:String.name, doc:'Parameter documentation text'),
    ]], doc:'Describe a method named parameter'

    method name: "add", type: "void", params: [member: "com.intellij.psi.PsiMember"]
    method name: "findClass", type: "com.intellij.psi.PsiClass", params: [name: "java.lang.String"]
    method name: "delegatesTo", type: "void", params: [elem: "com.intellij.psi.PsiElement"]

    method name: "enclosingCall",
          type: "com.intellij.psi.PsiElement",
          params: [methodName: "java.lang.String"]

    method name: "enclosingMethod", type: "com.intellij.psi.PsiMethod"
    method name: "enclosingMember", type: "com.intellij.psi.PsiMember"
    method name: "enclosingClass", type: "com.intellij.psi.PsiClass"

    property name: "place", type: "com.intellij.psi.PsiElement"
    property name: "classType", type: "com.intellij.psi.PsiClass"
  }
}

def psiClassContext = context(scope: closureScope(isArg: true), ctype: "com.intellij.psi.PsiClass")
contributor([psiClassContext]) {
  method name: "getMethods", type: "java.util.Collection"
  method name: "getQualName", type: "java.lang.String"
```

```
}

def psiMemberContext = context(scope: closureScope(isArg: true), ctype:
"com.intellij.psi.PsiMember")
contributor([psiMemberContext]) {
  method name: "hasAnnotation", params: [name: "java.lang.String"], type: "boolean"
  method name: "hasAnnotation", type: "boolean"
  method name: "getAnnotation", params: [name: "java.lang.String"], type:
"com.intellij.psi.PsiAnnotation"
  method name: "getAnnotations", params: [name: "java.lang.String"], type:
"java.util.Collection<com.intellij.psi.PsiAnnotation>"
}

def psiFieldContext = context(scope: closureScope(isArg: true), ctype: "com.intellij.psi.PsiField")
contributor([psiFieldContext]) {
  method name: "getClassType", type: "com.intellij.psi.PsiClass"
}

def psiMethodContext = context(scope: closureScope(isArg: true), ctype: "com.intellij.psi.PsiMethod")
contributor([psiMethodContext]) {
  method name: "getParamStringVector", type: "java.util.Map"
}

def psiElementContext = context(scope: closureScope(isArg: true), ctype:
"com.intellij.psi.PsiElement")
contributor([psiElementContext]) {
  method name: "bind", type: "com.intellij.psi.PsiElement"
  method name: "eval", type: "java.lang.Object"
  method name: "asList", type: "java.util.collection<com.intellij.psi.PsiElement>"
  method name: "getQualifier", type: "com.intellij.psi.PsiElement"
}

def expressionContext = context(scope: closureScope(isArg: true),
                                ctype:
"org.jetbrains.plugins.groovy.lang.psi.api.statements.expressions.GrExpression")
contributor([expressionContext]) {
  method name: "getArguments", type: "java.util.Collection"
```

```
  method name: "getClassType", type: "com.intellij.psi.PsiClass"
}
```

Describing custom annotations

GroovyDSL was used to describe compile-time AST transformations, introduced in Groovy 1.6. All scrip files listed below are bundled into last distributions of IntelliJ IDEA 9.

Delegate transformation: delegateTransform.gdsl
Category and Mixin transformations: categoryTransform.gdsl
Newify transformation: newifyTransform.gdsl
Singleton transformation: singletonTransform.gdsl
Bindable and Vetoable transformation: bindableTransform.gdsl, vetoableTransform.gdsl

Versioning

In the new versions of IDEA, GDSL may add some new primitives to describe your DSLs. If you want your single script to work with various IDEA versions (>=9.0.3), you may use `supportVersion`:

```
if (supportsVersion("10.0")) {
  contributor(ctxIntroducedIn10:methodIntroducedIn10()) { ... }
}
```