Query API

This guide shows you how to implement development rules with Query API.

The *com.optimyth.qaking.highlevelapi.dsl.Query* class represents a query in the syntax tree, in this case, the High-Level Tree (HLT).

Query provides a fluent interface to perform searches in the AST, specifying a sequence of operations (find, filter, navigate, visit...), which will be done starting from a given set of nodes. Each operation will be set passing primitive objects (NodePredicate, NodeVisitor, NodeSearch or Navigation) so then a call to one of the available *run()* methods makes to execute all the registered operations.

Once the nodes of interest are found, the *report()* operation will generate in the report one violation for each node reached.

Primitive objects

- *com.als.core.ast.NodeVisitor*: apply some logic to the node. Execution sequence on a tree or sub-tree.
- *com.als.core.ast.NodePredicate:* check if a node meets a series of clauses or conditions.
- *com.optimyth.qaking.highlevelapi.nodeset.NodeSearch*: find some node starting from a given one. For example, find the declaration of a variable from its use, the definition of a function from its call...
- *com.optimyth.qaking.highlevelapi.navigation.Navigation:* navigate through the AST from a given node, possibly applying some other primitive object.
- *com.optimyth.qaking.highlevelapi.dsl.QueryOperation*: definition of a new operation.

The available operations to perform the queries can be classified as follows:

Operation	Effect	Signature
custom operation	Registers a custom operation	operation(QueryOperation)
execute query	Executes the query, specifying initial node(s)	run(Rule, RuleContext) run(Rule, RuleContext, BaseNode...) run(Rule, RuleContext, NodeSet)
filter	Filter current context nodes	filter(NodePredicate) filter(Query)
find following a navigation	Find nodes traversed by navigation, matching predicate	find(NodePredicate, Navigation)
find sucessors	Find all successors matching predicate	find(NodePredicate match)
navigate	Traverses the given navigation from current (context) nodes	navigate(Navigation) navigate (NodeSearch) navigate(String xpath)
navigate & visit	Visit each node reachable via given navigation with visitor	visit(NodeVisitor, Navigation)
report	Emit a rule violation for each current context node or snapshot, using the ToViolation object to custimize violation to emit	report() report(String snapshot) report (ToViolation) report(String, ToViolation)
snapshot	Create a "snapshot" of current context nodeset, giving it a name	snapshot(String)
visit	Visits each node in current context	visit(NodeVisitor)

Most of these operations return the same Query instance so that **calls can be chained**. Besides, available actions and primitives are typically *thread-safe*, so you could use the Query object as a rule instance field and make the *query.run()* call from the *visit* method of the rule to process each source file under analysis.

As a simple example, imagine you want to report the getter methods that return null. Using the *Query* API, your rule could have an implementation that:

```
Predicate isGetter = ...;
NodePredicate returnsNull = ...;

Query q = Query.query()
               .find(methods(isGetter))
               .filter(returnsNull)
               .report();

... // execute query from high-level root node
q.run(rule, ctx, ctx.getHighLevelTree());
```

The rule will be with a **declarative format and coded in a few lines**, leaving the *thickest* part of the implementation in the declaration of the appropriate *primitives* for each case.

For the supported languages (Abap, C#, Cobol, C++, Java, Javascript, PHP), we can find *primitives*, both predicates, and navigations, already predefined, available to use directly in our rules. In each case, they will be found in the jar of the technology parser; for instance, we will have the classes com.optimyth.qaking.java.hla.JavaPredicates (where we can find the *isGetter* and *returnsNull* methods, used in the example above, already defined) and com.optimyth.qaking.java.hla.JavaNavigations.

Besides, we can always define those specific classes we need. To do that and specifically in the case of the predicates, it is possible to **use both the own class com.als.core.ast.NodePredicate**, already mentioned, **and the one defined in Google *Guava* library** (com.google.common.base.Predicate), which is included in the classpath for developing.

Let's see a complete example; in this case, a rule that detects, in Java classes, not used variables (local, class fields or parameters).

Java rule implementation example - Query API

```
import com.als.core.AbstractRule;
import com.als.core.RuleContext;
import com.als.core.ast.BaseNode;
import com.als.core.ast.NodePredicate;
import com.als.core.util.NodeToStr;
import com.optimyth.qaking.highlevelapi.dsl.Query;
import com.optimyth.qaking.highlevelapi.nodeset.ToViolation;
import com.optimyth.qaking.java.hla.ast.JavaVariable;

import staticcom.als.core.ast.NodePredicates.*;
import staticcom.optimyth.qaking.highlevelapi.dsl.Query.query;
import staticcom.optimyth.qaking.highlevelapi.nodeset.ToViolation.
extraMessage;
import staticcom.optimyth.qaking.java.hla.JavaPredicates.*;

public class UnusedVars extends AbstractRule {
  // For reporting the name of the unused var
  private static final ToViolation reportVarName = extraMessage(new
NodeToStr() {
    public String apply(BaseNode node) {
      return"unused " + ((JavaVariable) node).getName();
    }
  });

  // Match variable declarations of interest
  private static final NodePredicate varsPredicate = or(
    localVariablePred,
    // Unused private fields can only be used in other classes using
reflection
    // In Java, some class fields like serialVersionUID or
serialPersistentFields
    // are used by JVM, not by user code. How could you add such
exceptions?
    and(instanceVariablePred, isPrivatePred),
    parameterPred
  );

  // What is unused var?
  private static final Query unusedVars= query()
    .find( varsPredicate )
    // An unused var should not have initialization with side-effects
    // (because then, declaration cannot be removed)
    .filter(not(hasSideEffectInInitPred))
    .filter(not(hasUsages))
    .report( reportVarName);

  @Override protected void visit(BaseNode root, RuleContext ctx) {
    unusedVars.run(this, ctx, ctx.getHighLevelTree()); // rule is simply
query execution
  }

}
```

And another one: **the implementation of a rule for Cobol files that detects the use of the DISPLAY statement** in arithmetical operations.

```
Cobol rule implementation example - Query API
import com.als.cobol.CobolAstUtil;
import com.als.cobol.rules.AbstractCobolRule;
import com.als.core.RuleContext;
import com.als.core.ast.BaseNode;
import com.als.core.ast.NodePredicate;
import com.optimyth.qaking.cobol.ast.CobolNode;
import com.optimyth.qaking.cobol.hla.ast.DataEntry;
import com.optimyth.qaking.cobol.util.Declarations;
import com.optimyth.qaking.highlevelapi.dsl.Query;

import static com.als.cobol.UtilCobol.STATEMENT;
import static com.optimyth.qaking.cobol.hla.primitives.CobolPredicates.
ARITH_STATEMENTS;

public class NoDisplayDataInArithmeticOp extends AbstractCobolRule {

 // Match QualifiedDataName operand in arithmetic statement
 private final NodePredicate dataRefInArithStmt = new NodePredicate() {
  // To check that data item reference iside arithmetic statement
  private NodePredicate onArithStatement = new NodePredicate() {
   public boolean is(BaseNode node) {
    CobolNode containerStmt = ((CobolNode)node).ancestor(STATEMENT).child
(0);
    return ARITH_STATEMENTS.is(containerStmt);
   }
  };
  public boolean is(BaseNode node) {
   if(node.isTypeName("QualifiedDataName")) {
    CobolNode n = (CobolNode)node;
    return onArithStatement.is(n);
   }
   return false;
  }
 };

 // Match data item with DISPLAY or DISPLAY-1 types
 private final NodePredicate isDisplayType = new NodePredicate() {
  public boolean is(BaseNode node) {
   DataEntry de = Declarations.getDataEntry(node); // Find data item
declaration in DATA DIVISION
   if(de == null) return false; // no data item declaration in DATA
DIVISION
   String type = de.getType();
   return "DISPLAY".equalsIgnoreCase(type) || "DISPLAY-1".equalsIgnoreCase
(type);
  }
 };

 // "Match data reference in arithmetic expression where the data item
type is DISPLAY type"
 private final Query query = Query.query()
  .find(dataRefInArithStmt) // get data references in arithmetic statement
  .filter(isDisplayType) // ... but only DISPLAY / DISPLAY-1 types
  .report();

 @Override protected void visit(BaseNode root, RuleContext ctx) {
  query.run(this, ctx, CobolAstUtil.getProcedureDivision(root));
 }

}
```

If you have knowledge of XPath, the class *com.optimyth.qaking.highlevelapi.navigation.Region* is available, which will provide **Navigation** instances for each XPath axis.

Region name XPath axis Nodes traversed

- SELF self:: context node itself
- ROOT / go to the root node
- CHILDREN child:: immediate children
- PARENT parent:: parent node
- ANCESTORS ancestor-or-self:: ancestors, including self
- SUCCESSORS descendant:: subtree nodes from node, not including itself
- LEFTSIBLINGS preceding-sibling:: Siblings of node at left, not including itself
- RIGHTSIBLINGS following-sibling:: Siblings of node at right, not including itself
- PRECEDING preceding:: Nodes appearing before node (before in code text)
- FOLLOWING following:: Nodes appearing after node (before in code text)

As you can guess, the power and potential this API provides are great, so **we encourage you to examine and test** the available options in the development of your own rules.

To expand and consolidate the concepts that we have presented, please consult the documentation in the *development\doc* directory of Kiuwan Local Analyzer distribution.

Read more:

- Basic API
- XPath API