# Hermes Parser

EclipseLink Hermes is a JPQL parser. As of EclipseLink 2.4, Hermes has become the default JPQL parser. Hermes provides the following functionality:

* Parse JPQL queries that are
  + grammatically valid,
  + grammatically and/or semantically invalid or incomplete,
  + fragments;
* Semantic and grammatical validation;
* Calculate
  + query result type,
  + input parameter type;
* Refactoring support
  + Through an editable model (StateObject),
  + Basic through a delta;
* Content assist support.

**Table of Content**

[JPQL Parser](#_Toc329867695)

[JPQL Grammar](#_Toc329867696)

[JPQL Fragment](#_Toc329867697)

[Visiting a parsed JPQL query](#_Toc329867698)

[Third-party Provider](#_Toc329867699)

[Extending the JPQL grammar](#_Toc329867700)

[SPI](#_Toc329867701)

[Grammatical and Semantic Validation](#_Toc329867702)

[Content Assist](#_Toc329867703)

[Query Result Type and Input Parameter Type](#_Toc329867704)

[JPQL Query Helper](#_Toc329867705)

[State Model](#_Toc329867706)

[String Generation](#_Toc329867707)

[Manual Creation](#_Toc329867708)

[Extending the state model](#_Toc329867709)

[Visiting the State Model](#_Toc329867710)

[Third-party Provider](#_Toc329867711)

[Refactoring](#_Toc329867712)

## JPQL Parser

This section describes how to create a parsed tree representation of a JPQL query. The parser does not require any JPA metadata information when parsing a query.

Two modes are available when parsing a JPQL query:

* **Tolerant:** It means it will attempt – without failing – to parse any kind of strings, i.e. complete and grammatically valid and incomplete, malformed, invalid queries (which is grammatically invalid).
* **Non-tolerant**: It means it can only parse grammatically valid JPQL queries. This mode is used to increase performance.

The interface defining the parsed tree representation is

org.eclipse.persistence.jpa.jpql.parser.Expression

The root of the parsed tree hierarchy is

org.eclipse.persistence.jpa.jpql.parser.JPQLExpression

### JPQL Grammar

The JPQL grammar determines the structure of a JPQL query. The BNF of the grammar has been translated into Java. A query BNF then uses a factory to create an Expression object.

The interface defining the JPQL grammar is

org.eclipse.persistence.jpa.jpql.parser.JPQLGrammar

The interface defining the expression factory is

org.eclipse.persistence.jpa.jpql.parser.ExpressionFactory

The JPQL query BNF is defined by (it represents a single BNF of the grammar):

org.eclipse.persistence.jpa.jpql.parser.JPQLQueryBNF

By default, Hermes provides:

|  |  |
| --- | --- |
| DefaultJPQLGrammar | It is based on the latest version of the JPA functional specification, which is 2.0. |
| JPQLGrammar1\_0 | It is based on version 1.0 of the JPA functional specification. |
| JPQLGrammar2\_0 | It is based on version 2.0 of the JPA functional specification. |
| JPQLGrammar2\_1 | It is based on version 2.1 of the JPA functional specification. |
| DefaultEclipseLinkJPQLGrammar | It is based on the latest JPA functional specification supported by Hermes, which is JPA 2.0 and on the latest EclipseLink release, which is 2.4. |
| EclipseLinkJPQLGrammar1 | It is based on the JPA 1.0 functional specification and on EclipseLink 1.x. |
| EclipseLinkJPQLGrammar2\_0 | It is based on the JPA 2.0 functional specification and on EclipseLink 2.0 only. |
| EclipseLinkJPQLGrammar2\_1 | It is based on the JPA 2.0 functional specification and on EclipseLink 2.1 only. |
| EclipseLinkJPQLGrammar2\_2 | It is based on the JPA 2.0 functional specification and on EclipseLink 2.2 only. |
| EclipseLinkJPQLGrammar2\_3 | It is based on the JPA 2.0 functional specification and on EclipseLink 2.3 only. |
| EclipseLinkJPQLGrammar2\_4 | It is based on the JPA 2.0 functional specification and on EclipseLink 2.4 only. |

The following examples show how to parse a JPQL query:

// Using tolerant mode when parsing a valid JPQL query

JPQLExpression jpqlExpression = new JPQLExpression(

"SELECT e FROM Employee e",

DefaultJPQLGrammar.instance(),

**true**

);

// Using non-tolerant mode when parsing a valid/complete or

// invalid/incomplete JPQL query

// Note: The invalid segments are underlined

JPQLExpression jpqlExpression = new JPQLExpression(

"SELECT e FROM Employee WHERE GROUP BY e.",

DefaultJPQLGrammar.instance(),

**false**

);

// Create a string representation of the parsed query

String parsedJPQLQuery = jpqlExpression.toParsedText();

### JPQL Fragment

In version 2.0, it is now possible to parse JPQL fragments. The unique identifier of the query BNF is used to determine how that fragment will be parsed.

For example, if a conditional expression needs to be parsed without having to write a complete JPQL query, then the ID of the conditional expression BNF is passed to JPQLExpression.

JPQLExpression jpqlExpression = new JPQLExpression(

"e.name <> 'JPQL' AND TYPE(e) = Employee",

DefaultJPQLGrammar.instance(),

ConditionalExpressionBNF.ID

);

JPQLExpression.getQueryStatement() returns the conditional expression. If the JPQL fragment could not be parsed, then getQueryStatement() returns null and getUnknownEndingStatement() returns the unknown JPQL fragment.

### Visiting a parsed JPQL query

The parsed tree can be traversed by using a visitor; it allows adding additional operations and functionality without integrating it into the parsed tree. Validation and content assist provider have been coded this way.

The interface is:

org.eclipse.persistence.jpa.jpql.ExpressionVisitor

Various subclasses have been defined:

|  |  |
| --- | --- |
| AbstractExpressionVisitor | Simply implemented all the methods. |
| AnonymousExpressionVisitor | Every method is redirected to a generic method:  visit(Expression). |
| AbstractTraverseChildrenVisitor | Traverses the entire tree from the visited expression to its deepest children. |
| AbstractTraverseParentVisitor | Traverses back up the hierarchy from the visited expression. |

The following example shows how to collect all the identification variables that have been used throughout the query (except from the declaration clause).

String jpqlQuery = "SELECT e.name, e.age " +

"FROM Employee e " +

"WHERE EXISTS (SELECT p " +

" FROM Phone p " +

" WHERE p.employee = e AND p.type = 'Cell')";

JPQLExpression jpqlExpression = new JPQLExpression(

jpqlQuery,

DefaultJPQLGrammar.instance(),

false

);

final Set<String> variables = new HashSet<String>();

// AbstractTraverseChildrenVisitor is used to the entire tree can be visited

ExpressionVisitor visitor = new AbstractTraverseChildrenVisitor() {

/\*\*

\* {@inheritDoc}

\*/

@Override

public void visit(FromClause expression) {

// Prevent traversing the clause

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public void visit(IdentificationVariable expression) {

variables.add(expression.getText());

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public void visit(SimpleFromClause expression) {

// Prevent traversing the clause

}

};

jpqlExpression.accept(visitor);

### Third-party Provider

The parser uses the Visitor pattern to traverse the Expression hierarchy. This adds some constraints when creating new classes defined by third-party providers. The ExpressionVisitor cannot be extended to support those new classes, in order to circumvent this problem, third-party providers can use AbstractExpression.acceptUnknownVisitor(). It will programmatically call the visit method defined on the third-party visitor.

public class MyExpression extends AbstracExpression {

/\*\*

\* {@inheritDoc}

\*/

@Override

public void accept(ExpressionVisitor visitor) {

acceptUnknownVisitor(visitor);

}

}

public class MyExpressionVisitor extends AbstracExpressionVisitor {

public void visitor(MyExpressionVisitor visitor) {

}

}

The method’s access level can be public, protected or private. Package-private should not be used if used within an OSGi environment because it will not be accessible. Functionality provided by Hermes can be extended and the API was coded in such a way that subclasses of an ExpressionVisitor will be required to provide behavior for any third-party classes to be visited.

### Extending the JPQL grammar

It is possible to extend and to modify the JPQL grammar defined in the Java Persistence functional specification. Extending the grammar allows to modify the way JPQL queries are parsed and it also allows adding additional functionality, i.e. additional JPQL identifiers.

To do so, one can extend the abstract implementation:

org.eclipse.persistence.jpa.jpql.parser.AbstractJPQLGrammar

The following methods can be overridden to provide the extensibility support:

buildJPQLGrammar()

The base JPQL grammar that will be extended.

initializeBNFs()

Used to add additional query BNFs and to extend the BNF defined in the base JPQL grammar.

initializeExpressionFactories()

Used to register additional ExpressionFactory.

initializeIdentifiers()

Used to define the identifier role and version of the new JPQL identifiers.

Let say we want to add a new JPQL identifier to the JPQL grammar defined in the JPA 2.0 functional specification. The identifier name is **FOO**, it is a function and the BNF is

foo\_expression ::= **FOO**(scalar\_expression, simple\_arithmetic\_expression)

and the identifier can be used in the **SELECT** clause as well as in the **WHERE**/**HAVING** clauses:

import org.eclipse.persistence.jpa.jpql.parser.AbstractJPQLGrammar;

import org.eclipse.persistence.jpa.jpql.parser.JPQLGrammar2\_0;

public final class CustomJPQLGrammar extends AbstractJPQLGrammar {

/\*\*

\* The singleton instance of this JPQL grammar.

\*/

private static final CustomJPQLGrammar INSTANCE = new CustomJPQLGrammar();

/\*\*

\* Creates a new <code>CustomJPQLGrammar</code>.

\*/

private CustomJPQLGrammar() {

super();

}

/\*\*

\* Returns the singleton instance of this class.

\*

\* @return The {@link CustomJPQLGrammar}

\*/

public static CustomJPQLGrammar instance() {

return INSTANCE;

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected JPQLGrammar buildJPQLGrammar() {

return new JPQLGrammar2\_0();

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public IJPAVersion getJPAVersion() {

return IJPAVersion.VERSION\_2\_0;

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected void initializeBNFs() {

registerBNF(new FooExpressionBNF());

addChildBNF(SelectExpressionBNF.ID, FooExpressionBNF.ID);

addChildBNF(ConditionalExpressionBNF.ID, FooExpressionBNF.ID);

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected void initializeExpressionFactories() {

registerFactory(new FooExpressionFactory());

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected void initializeIdentifiers() {

registerIdentifierRole ("FOO", IdentifierRole.FUNCTION);

registerIdentifierVersion("FOO", IJPAVersion.VERSION\_2\_0);

}

}

The query BNF class would look like this:

import org.eclipse.persistence.jpa.jpql.parser.JPQLQueryBNF;

public final class FooExpressionBNF extends JPQLQueryBNF {

/\*\*

\* The unique identifier for this {@link FooExpressionBNF}.

\*/

public static final String ID = "foo\_expression";

/\*\*

\* Creates a new <code>FooExpressionBNF</code>.

\*/

public FooExpressionBNF() {

super(ID);

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected void initialize() {

super.initialize();

registerExpressionFactory(FuncExpressionFactory.ID);

}

}

The factory class would look like this:

import org.eclipse.persistence.jpa.jpql.WordParser;

import org.eclipse.persistence.jpa.jpql.parser.AbstractExpression;

import org.eclipse.persistence.jpa.jpql.parser.ExpressionFactory;

import org.eclipse.persistence.jpa.jpql.parser.JPQLQueryBNF;

public final class FooExpressionFactory extends ExpressionFactory {

/\*\*

\* The unique identifier of this {@link FuncExpressionFactory}.

\*/

public static final String ID = Expression.FUNC;

/\*\*

\* Creates a new <code>FooExpressionFactory</code>.

\*/

public FooExpressionFactory() {

super(ID, "FOO");

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected AbstractExpression buildExpression(AbstractExpression parent,

WordParser wordParser,

String word,

JPQLQueryBNF queryBNF,

AbstractExpression expression,

boolean tolerant) {

expression = new FooExpression(parent);

expression.parse(wordParser, tolerant);

return expression;

}

}

The actual Expression class representing the **FOO** function has two encapsulated expressions; the parser already provides such support.

import org.eclipse.persistence.jpa.jpql.WordParser;

public final class FooExpression extends AbstractDoubleEncapsulatedExpression {

/\*\*

\* Creates a new <code>FooExpression</code>.

\*

\* @param parent The parent of this expression

\*/

public FooExpression(AbstractExpression parent) {

super(parent);

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public void accept(ExpressionVisitor visitor) {

// A sub-interface of ExpressionVisitor will have to be defined

// Unfortunately, an instance is required

if (visitor instanceof SomeExtendedExpressionVisitor) {

((SomeExtendedExpressionVisitor) visitor).visit(this);

}

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public JPQLQueryBNF getQueryBNF() {

return getQueryBNF(FoorExpressionBNF.ID);

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public JPQLQueryBNF parameterExpressionBNF(int index) {

if (index == 0) {

return getQueryBNF(ScalarExpressionBNF.ID);

}

return getQueryBNF(SimpleArithmeticExpressionBNF.ID);

}

/\*\*

\* {@inheritDoc}

\*/

@Override

protected String parseIdentifier(WordParser wordParser) {

return "FOO";

}

}

## SPI

The Hermes SPI provides a simple way to access the domain model from a JPA provider. The interfaces to implement are defined in the package

org.eclipse.persistence.jpa.jpql.spi

Hermes 2.0 added an implementation of the SPI that accesses Java classes. Those classes are defined in the package:

org.eclipse.persistence.jpa.jpql.spi.java

The following example shows how to create the Java version of the SPI and how to retrieve the JPQL query defined on the Employee class.

import java.io.Serializable;

import javax.persistence.Entity;

import javax.persistence.NamedQuery;

@Entity

@NamedQuery(name="Employee.findAll", query="SELECT e FROM Employee e")

public class Employee implements Serializable {

@Id

public long id;

}

// Create the provider

IManagedTypeProvider provider = new JavaManagedTypeProvider();

// Add the entity

IEntity entity = provider.addEntity(Employee.class);

// Retrieve the JPQL query

IQuery query = entity.getNamedQuery("Employee.findAll");

## Grammatical and Semantic Validation

JPQL queries can be validated grammatically and semantically. Grammatical validation is based on the JPQL grammar. Semantic validation is validating the content of the query.

The grammatical validators are

org.eclipse.persistence.jpa.jpql.DefaultGrammarValidator

org.eclipse.persistence.jpa.jpql.EclipseLinkGrammarValidator

The semantic validators are

org.eclipse.persistence.jpa.jpql.DefaultSemanticValidator

org.eclipse.persistence.jpa.jpql.EclipseLinkSemanticValidator

The validators use a JPQLQueryContext to query information about the JPQL query and to cache that information so it can be retrieved many times.

The code can be used two different ways. Version 1 shows how to validate the JPQL query without having in mind to cache JPQLQueryContext and the validators for future use and version 2 is used with caching in mind.

Version 1:

// A JPQL query with a grammatical and semantic errors

// SELECT FROM Employee e WHERE e.name = 2

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

// Create the context that will be used by both validators

JPQLQueryContext context = new DefaultJPQLQueryContext(DefaultJPQLGrammar.instance());

context.setQuery(externalQuery);

// Create the list that will contain the problems

List<JPQLQueryProblem> problems = new ArrayList<JPQLQueryProblem>();

// Create the validator

DefaultGrammarValidator grammarValidator = new DefaultGrammarValidator(context);

// The validator will use our List to store the problems

grammarValidator.setProblems(problems);

// Now validate the JPQL query by visiting it

context.getJPQLExpression().accept(grammarValidator);

// The list contains the grammatical problem

// Create the validator

DefaultSemanticValidator semanticValidator = new DefaultSemanticValidator(context);

// The validator will use our List to store the problems

semanticValidator.setProblems(problems);

// Now validate the JPQL query by visiting it

context.getJPQLExpression().accept(semanticValidator);

// The list contains the semantic problem

Version 2:

// A JPQL query with a grammatical and semantic errors

String jpqlQuery = "SELECT FROM Employee e WHERE e.name = 2";

JPQLExpression jpqlQuery = new JPQLExpression(

DefaultJPQLGrammar.instance(),

jpqlQuery,

true

);

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

// Create the context that will be used by both validators

JPQLQueryContext context = new DefaultJPQLQueryContext(DefaultJPQLGrammar.instance());

// The following line is only required when the JPQLExpression is already parsed

context.setJPQLExpression(jpqlQuery);

// This line has to be invoked before the previous one,

// the JPQL query will not be parsed internally

context.setQuery(externalQuery);

// Create the list that will contain the problems

List<JPQLQueryProblem> problems = new ArrayList<JPQLQueryProblem>();

// Create the validator

DefaultGrammarValidator grammarValidator = new DefaultGrammarValidator(context);

try {

// The validator will use our List to store the problems

grammarValidator.setProblems(problems);

// Now validate the JPQL query by visiting it

jpqlQuery.accept(grammarValidator);

}

finally {

// Dispose is only required when the validator is cached for future use

grammarValidator.dispose();

}

// The list contain the grammatical problem

// Create the validator

DefaultGrammarValidator semanticValidator = new DefaultSemanticValidator(context);

try {

// The validator will use our List to store the problems

semanticValidator.setProblems(problems);

// Now validate the JPQL query by visiting it

jpqlQuery.accept(semanticValidator);

}

finally {

// Dispose is only required when the validator is cached for future use

semanticValidator.dispose();

}

// The list contains the semantic problem

// Dispose is only required when the context is cached for future use

context.dispose();

Version 3: Using the helper

// A JPQL query with a grammatical and semantic errors

// SELECT FROM Employee e WHERE e.name = 2

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

DefaultJPQLQueryHelper helper = new DefaultJPQLQueryHelper(

DefaultJPQLGrammar.instance()

);

helper.setQuery(externalQuery);

// Validate both the grammar and the semantic

List<JPQLQueryProblem> problems = helper.validate();

// Dispose is only required when the context is cached for future use

context.dispose();

## Content Assist

EclipseLink Hermes provides content assist support. Based on the position of the cursor within a JPQL query, it can determine the valid choices.

The content assist providers are

org.eclipse.persistence.jpa.jpql.DefaultContentAssistValidator

org.eclipse.persistence.jpa.jpql.EclipseLinkContentAssistValidator

These validators also use a JPQLQueryContext to query information about the JPQL query and to cache that information so it can be retrieved many times.

The code can also be used two different ways. Version 1 shows how to retrieve the possible choices without having in mind to cache JPQLQueryContext and the validators for future use and version 2 is used with caching in mind.

Version 1:

// SELECT e FROM Employee e WHERE e.|

int position = 33;

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

// Create the context that will be used by both validators

JPQLQueryContext context = new DefaultJPQLQueryContext(DefaultJPQLGrammar.instance());

context.setQuery(externalQuery);

// Create the visitor

DefaultContentAssistVisitor visitor =

new DefaultContentAssistVisitor(context);

// Retrieve the proposals

ContentAssistProposals proposals = visitor.buildProposals(position);

Version 2:

// SELECT e FROM Employee e WHERE e.|

int position = 33;

JPQLExpression jpqlQuery = new JPQLExpression(

DefaultJPQLGrammar.instance(),

jpqlQuery,

true

);

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

// Create the context that will be used by both validators

JPQLQueryContext context = new DefaultJPQLQueryContext(DefaultJPQLGrammar.instance());

// The following line is only required when the JPQLExpression is already parsed

context.setJPQLExpression(jpqlQuery);

// This line has to be invoked before the previous one,

// the JPQL query will not be parsed internally

context.setQuery(externalQuery);

// Create the visitor

DefaultContentAssistVisitor visitor =

new DefaultContentAssistVisitor(context);

try {

// Retrieve the proposals

ContentAssistProposals proposals = visitor.buildProposals(position);

}

finally {

// Dispose is only required when the validator is cached for future use

visitor.dispose();

}

// Dispose is only required when the context is cached for future use

context.dispose();

Version 3: Using the helper

// SELECT e FROM Employee e WHERE e.|

int position = 33;

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

DefaultJPQLQueryHelper helper = new DefaultJPQLQueryHelper(

DefaultJPQLGrammar.instance()

);

helper.setQuery(externalQuery);

// Retrieve the proposals

ContentAssistProposals proposals = helper.buildContentAssistProposals(position);

// Dispose is only required when the context is cached for future use

context.dispose();

## Query Result Type and Input Parameter Type

The query result type can be calculated for a read query (**SELECT** query).

// SELECT e FROM Employee e

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

JPQLQueryContext context = new DefaultJPQLQueryContext(DefaultJPQLGrammar.instance());

context.setQuery(externalQuery);

IType resultType = context.getType(context.getJPQLExpression());

It is also possible to calculate the closest type of any input parameter present in a JPQL query.

// SELECT e FROM Employee e WHERE e.name = ?1

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

DefaultJPQLQueryHelper helper = new DefaultJPQLQueryHelper(

DefaultJPQLGrammar.instance()

);

helper.setQuery(externalQuery);

IType resultType = helper.getParameterType("?1");

// resultType should be java.lang.String

If the type cannot be inferred, then the IType of java.lang.Object is returned.

#### JPQL Query Helper

Hermes provides, as shown in examples, a helper that provides easier access to most of the functionality.

The helpers are

org.eclipse.persistence.jpa.jpql.DefaultJPQLQueryHelper

org.eclipse.persistence.jpa.jpql.EclipseLinkJPQLQueryHelper

Any third-party can extend the helper by extending

org.eclipse.persistence.jpa.jpql.AbstractJPQLQueryHelper

## State Model

The parsed tree representation of a JPQL query is a read-only tree, once the objects are created, they cannot be modified. Either to create one from scratch or to edit an existing one, a different model needs to be used.

The interface defining the state object model is

org.eclipse.persistence.jpa.jpql.model.query.IStateObject

The root of the state model hierarchy is

org.eclipse.persistence.jpa.jpql.model.query.JPQLQueryStateObject

The root of the state object model requires a builder. It is used to create a complete representation of an existing JPQL query but it is also used when the state object is been edited by using one of the many parse methods.

The interface defining the builder is

org.eclipse.persistence.jpa.jpql.model.IJPQLQueryBuilder

By default, Hermes provides:

|  |  |
| --- | --- |
| DefaultJPQLQueryBuilder | It is based on the latest JPA functional specification supported by Hermes, which is JPA 2.0. |
| JPQLQueryBuilder1\_0 | It is based on version 1.0 of the JPA functional specification. |
| JPQLQueryBuilder2\_0 | It is based on version 2.0 of the JPA functional specification. |
| JPQLQueryBuilder2\_1 | It is based on version 2.1 of the JPA functional specification. |
| DefaultEclipseLinkJPQLQueryBuilder | It is based on the latest JPA functional specification supported by Hermes, which is JPA 2.0 and on the latest EclipseLink release, which is 2.4. |
| EclipseLinkJPQLQueryBuilder | It is based on the JPA 1.0 functional specification and on the version of EclipseLink defined by the JPQLGrammar given to this builder. |

The following example shows how a state model is created from an existing JPQL query:

String jpqlQuery = "SELECT e FROM Employee e WHERE e.name = 'JPQL'";

IManagedTypeProvider provider = ...;

DefaultJPQLQueryBuilder builder = new DefaultJPQLQueryBuilder();

JPQLQueryStateObject stateObject = builder.buildStateObject(

provider,

jpqlQuery,

true

);

### String Generation

The state object model does not provide support for generating a “valid” string representation; a formatter is used instead to convert the state object model into a string.

The interface is

org.eclipse.persistence.jpa.jpql.model.IJPQLQueryFormatter

By default, Hermes provides:

|  |  |
| --- | --- |
| DefaultJPQLQueryFormatter | It is based on the latest JPA functional specification supported by Hermes, which is JPA 2.0. |
| EclipseLinkJPQLQueryFormatter | It is based on the latest JPA functional specification supported by Hermes, which is JPA 2.0 and on the latest EclipseLink release, which is 2.4. |

The following example shows how to create a string representation of a state object using the default implementation of IJPQLQueryFormatter:

IStateObject stateObject = ...;

IJPQLQueryFormatter formatter = new DefaultJPQLQueryFormatter();

String jpqlQuery = builder.toString(stateObject);

### Manual Creation

Many of the concrete IStateObject classes have setter methods; add/remove methods and a parse method. The parse method allows parsing a JPQL fragment and automatically converts it into a state object.

// SELECT e FROM Employee e

JPQLGrammar jpqlGrammar = ...;

IManagedTypeProvider provider = ...;

IJPQLQueryBuilder jpqlQueryBuilder = ...;

JPQLQueryStateObject jpqlStateObject = new JPQLQueryStateObject(

jpqlQueryBuilder,

provider,

jpqlGrammar

);

SelectStatementStateObject select = jpqlStateObject.addSelectStatement();

select.addRangeDeclaration("Employee", "e");

select.addSelectItem("e");

The conditional expression of the WHERE, HAVING and WHEN clauses can be created manually or it can also be created using an IConditionalStateObjectBuilder.

// SELECT e

// FROM Employee e

// WHERE e.department.name = 'NA42' AND

// e.address.state IN ('NY', 'CA')

JPQLGrammar jpqlGrammar = ...;

IManagedTypeProvider provider = ...;

IJPQLQueryBuilder jpqlQueryBuilder = ...;

JPQLQueryStateObject jpqlStateObject = new JPQLQueryStateObject(

jpqlQueryBuilder,

provider,

jpqlGrammar

);

SelectStatementStateObject select = jpqlStateObject.addSelectStatement();

select.addRangeDeclaration("Employee", "e");

select.addSelectItem("e");

WhereClauseStateObject whereClause = select.addWhereClause();

There are four different ways to create the conditional expression.

Version 1:

// Manually create the conditional expression, which can be tedious

// e.department.name = 'NA42'

ComparisonExpressionStateObject comparison =

new ComparisonExpressionStateObject(whereClause, Expression.EQUAL);

comparison.setLeft(

new StateFieldPathExpressionStateObject(comparison, "e.department.name")

);

comparison.setRight(new StringLiteralStateObject(comparison, "'NA42'"));

// e.address.state IN ('NY', 'CA')

InExpressionStateObject in = new InExpressionStateObject(whereClause);

in.setStateFieldPath(new StateFieldPathExpressionStateObject(in, "e.address.state"));

in.addItem(new StringLiteralStateObject(in, "'NY'"));

in.addItem(new StringLiteralStateObject(in, "'CA'"));

// x AND y

whereClause.setConditionalStateObject(new AndExpressionStateObject(comparison, in));

Version 2:

// Add the WHERE clause and retrieve its IConditionalStateObjectBuilder

IConditionalStateObjectBuilder builder = whereClause.getBuilder();

// and create the conditional expression

builder

.path("e.department.name").equal("'NA42'")

.and(

builder.path("e.address.state").in("'NY'", "'CA'")

)

.commit();

Version 3:

// Use the parse method instead of the builder

whereClause.parse("e.department.name = 'NA42' AND e.address.state IN ('NY', 'CA')");

Version 4:

// Break the parsing into multiple fragments

whereClause.parse("e.department.name = 'NA42'");

whereClause.andParse("e.address.state IN ('NY', 'CA')");

### Extending the state model

It is also possible to extend the state object model. To do so, the extended behavior is coded into a new state object and it decorates the state object to extend. The methods that add the support for decorating a state object are:

public interface IStateObject {

/\*\*

\* Decorates this {@link IStateObject} with the given decorator. It means the

\* behavior of this {@link IStateObject} is modified by the given one. By default,

\* this {@link IStateObject} becomes the parent of the given one.

\*

\* @param decorator The {@link IStateObject} decorating this one

\*/

void decorate(IStateObject decorator);

/\*\*

\* Returns the {@link IStateObject} decorating this one if one has been set, which

\* means the behavior of this {@link IStateObject} is modified by the decorator.

\*

\* @return The {@link IStateObject} decorating this one

\*/

IStateObject getDecorator();

/\*\*

\* Determines whether this {@link IStateObject} is being decorated by another

\* {@link IStateObject}, which means the behavior is modified by the given one.

\*

\* @param <code>true</code> if this {@link IStateObject} is being decorated;

\* <code>false</code>

\*/

boolean isDecorated();

}

The given example shows how the EclipseLink’s **TREAT** function is created so it modifies the join association path expression.

// SELECT e FROM Employee e WHERE e.name = 'JPQL'

package org.eclipse.persistence.jpa.jpql.model;

public class EclipseLinkStateObjectBuilder extends BasicStateObjectBuilder

implements EclipseLinkExpressionVisitor {

/\*\*

\* {@inheritDoc}

\*/

@Override

protected IBuilder<JoinStateObject,

AbstractIdentificationVariableDeclarationStateObject> buildJoinBuilder() {

return new EclipseLinkJoinBuilder();

}

/\*\*

\* This builder adds support for the <code><b>TREAT</b></code> expression used in a

\* <code><b>JOIN</b></code> expression.

\*/

protected class EclipseLinkJoinBuilder extends JoinBuilder

implements EclipseLinkExpressionVisitor {

...

/\*\*

\* {@inheritDoc}

\*/

@Override

public void visit(TreatExpression expression) {

String entityTypeName = literal(expression.getEntityType(),

LiteralType.ENTITY\_TYPE);

// Create the TREAT state object

TreatExpressionStateObject treatStateObject = new TreatExpressionStateObject(

stateObject,

expression.hasAs(),

entityTypeName

);

// Now decorate the join association path

stateObject.getJoinAssociationPathStateObject().decorate(treatStateObject);

...

}

}

}

### Visiting the State Model

The state object model also supports being visited.

The interface is:

org.eclipse.persistence.jpa.jpql.model.StateObjectVisitor

EclipseLink also defines an interface that contains its extension:

org.eclipse.persistence.jpa.jpql.model.EclipseLinkStateObjectVisitor

Various subclasses have been defined:

|  |  |
| --- | --- |
| AbstractStateObjectVisitor/  AbstractEclipseLinkStateObjectVisitor | Simply implemented all the methods. |
| AnonymousStateObjectVisitor/  AnonymousEclipseLinkStateObjectVisitor | Every method is redirected to a generic method:  visit(StateObject). |
| AbstractTraverseChildrenVisitor/  AbstractEclipseLinkTraverseChildrenVisitor | Traverses the entire tree from the visited state object to its deepest children. |
| AbstractTraverseParentVisitor/  AbstractEclipseLinkTraverseParentVisitor | Traverses back up the hierarchy from the visited state object. |

### Third-party Provider

The parser uses the Visitor pattern to traverse the StateObject hierarchy. This adds some constraints when creating new classes defined by third-party providers. The StateObjectVisitor cannot be extended to support those new classes, in order to circumvent this problem, third-party providers can use AbstractStateObject.acceptUnknownVisitor(). It will programmatically call the visit method defined on the third-party visitor.

public class MyStateObject extends AbstracStateObject {

/\*\*

\* {@inheritDoc}

\*/

@Override

public void accept(StateObjectVisitor visitor) {

acceptUnknownVisitor(visitor);

}

}

public class MyStateObjectVisitor extends AbstracStateObjectVisitor {

public void visitor(MyStateObjectVisitor visitor) {

}

}

The method’s access level can be public, protected or private. Package-private should not be used if used within an OSGi environment because it will not be accessible. Functionality provided by Hermes can be extended and the API was coded in such a way that subclasses of an ExpressionVisitor will be required to provide behavior for any third-party classes to be visited.

## Refactoring

Version 2 added refactoring support. Two methods of refactoring are supported:

* By using an editable state object
* By gathering the changes in a delta

Version 1:

// Manual refactoring

// SELECT e FROM Employee e

// Rename Employee to TempEmployee

JPQLGrammar jpqlGrammar = ...;

IManagedTypeProvider provider = ...;

IJPQLQueryBuilder jpqlQueryBuilder = ...;

JPQLQueryStateObject jpqlStateObject = new JPQLQueryStateObject(

jpqlQueryBuilder,

provider,

jpqlGrammar

);

// Create the visitor that will change the state object

StateObjectVisitor visitor = new AbstractTraverseChildrenVisitor() {

public void visit(AbstractSchemaNameStateObject stateObject) {

if (stateObject.getText().equals("Employee")) {

stateObject.setText("TempEmployee");

}

}

}

jpqlStateObject.accept(visitor);

// Retrieve the string representation of the result

IJPQLQueryFormatter formatter = new DefaultJPQLQueryFormatter();

String result = builder.toString(stateObject);

Version 2:

// Refactoring using RefactoringTool

String jpqlQuery = "SELECT e FROM Employee e";

IManagedTypeProvider provider = ...;

IJPQLQueryBuilder jpqlQueryBuilder = ...;

RefactoringTool refactoringTool = new RefactoringTool(

provider,

jpqlQueryBuilder,

jpqlQuery

);

refactoringTool.renameEntityName("Employee", "TempEmployee");

// Retrieve the string representation of the result

String result = refactoringTool.toActualText();

Version 3:

// Refactoring using BasicRefactoringTool

String jpqlQuery = "SELECT e FROM Employee e";

IManagedTypeProvider provider = ...;

JPQLGrammar jpqlGrammar = ...;

BasicRefactoringTool refactoringTool = new BasicRefactoringTool(

jpqlQuery,

jpqlGrammar,

provider

);

refactoringTool.renameEntityName("Employee", "TempEmployee");

// Retrieve the delta

RefactoringDelta delta = refactoringTool.getDelta();

// The delta is stored in a TextEdit

TextEdit textEdit = delta.textEdits().next();

// Retrieve the string representation of the result

String result = delta.applyChanges();

Version 3:

// Refactoring using the helper

// Retrieve the JPQL query from the SPI

IQuery externalQuery = ...;

DefaultJPQLQueryHelper helper = new DefaultJPQLQueryHelper(

DefaultJPQLGrammar.instance()

);

helper.setQuery(externalQuery);

// Create the RefactoringTool

RefactoringTool refactoringTool = helper.buildRefactoringTool();

refactoringTool.renameEntityName("Employee", "TempEmployee");

// Create the BasicRefactoringTool, this method will be available in EclipseLink 2.5

BasicRefactoringTool basicRefactoringTool = helper.buildBasicRefactoringTool();

basicRefactoringTool.renameEntityName("Employee", "TempEmployee");