Guide

Release 1.0.0

Author: Uwe Finke

http://cubaja.googlecode.com

config

Simple example

Imagine a batch application which reads rows from a database and writes them into a file. The selection is restricted to rows which were inserted within a certain time period. Database connection parameters, file name and time period have to be configurable. We place this information in an XML file named 'config.xml':

```
<?xml version="1.0"?>
<config dateFrom="2010-01-01"
        dateTo="2010-01-31">
  <database driver="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/cubaja"
            user="cubaja"
            password="test"/>
  <output name="/path_to_file/output.txt"/>
</config>
```

A Java class corresponds to the structures of our XML; all attribute and element names match to setter methods by name. With an IDE, we only have to write the attribute types and names; the setter and getter methods are generated.

```
// imports ...
public class Config {
  private Date dateFrom;
  private Date dateTo;
  private DatabaseConfig database;
  private FileConfig output;
  public Config() {
  }
  public void setDateFrom(Date dateFrom) {
    this.dateFrom = dateFrom;
  }
  public void setDateTo(Date dateTo) {
    this.dateTo = dateTo;
  }
  public void setDatabase(DatabaseConfig database) {
    this.database = database;
  }
  public void setOutput(FileConfig output) {
    this.output = output;
  }
  public DatabaseConfig getDatabase() {
    return database;
  }
  // ... other getter methods ...
}
```

At runtime, we put the XML file's directory (which may be a common config directory) into the classpath. The application uses a Configurator to parse the XML:

```
import de.ufinke.cubaja.config.Configurator;

// ...

Configurator configurator = new Configurator();
Config config = configurator.configure(new Config());
```

Protection against mistakes

Sometimes it may happen, e.g. because of a typo, that an XML configuration attribute or element doesn't match a setter method in the configuration class. Then the Configurator throws a ConfigException.

The other way round, when there was no attribute or element in the config file which corresponds to an existing setter method, the Configurator doesn't complain about something missing. An attribute or element may be optional and a null or another default value may be used by the application. If a default value is not applicable, we should protect our application against invalid data by marking the concerning setter methods with the Mandatory annotation:

```
import de.ufinke.cubaja.config.Mandatory;

// ...

@Mandatory
public void setDatabase(DatabaseConfig database) {

  this.database = database;
}
```

Now, if someone forgets to code the database element, the Configurator will throw a ConfigException.

There is another annotation named 'Pattern' to check XML content. The pattern is a regular expression, or - with Date parameters - a date pattern for SimpleDateFormat. We may code an optional hint attribute which is included in an error message.

```
import de.ufinke.cubaja.config.Pattern;

// ...

@Pattern ("dd.MM.yyyy")
public void setDateFrom(Date dateFrom) {

  this.dateFrom = dateFrom;
}

@Pattern (value="d{10}" hint="10-stellig numerisch")
public void setAccount(long account) {

  this.account = account;
}
```

The setter method may perform its own checks and throw a ConfigException:

```
public void setNumber(int number) throws ConfigException {

  if (number > 42) {
    throw new ConfigException("number must be less than 42");
  }
  this.number = number;
}
```

Properties

XML may contain properties in the form of '${*propertyName*}'. Properties are replaced by values which come from various sources. In a Configurator instance, we may supply properties and define the search sequence.

By default, properties are searched in the following order:

- System properties.
- Properties defined in an optional file named 'config.properties'. If the ResourceLoader finds such a file, it is processed.
- Properties defined by elements named 'configProperty' in the XML source. They are valid from the point of their definition downward in the XML. Another definition with the same name replaces the original value.
- Environment variables.

```
<configProperty name="path" value="/home/path/foo"/>

<database driver="com.mysql.jdbc.Driver"
          url="${DB_URL_FROM_ENV}"
          user="${DB_USER_FROM_ENV}"
          password="${DB_PASSWORD_FROM_ENV}"/>

<output name="${path}/output.txt"/>
```

A special case are properties supplied by a NamedPropertyProvider. They are useful when the property value depends on program logic. We may define such a provider with the Configurator or in the XML.

```
<configPropertyProvider name="timestamp"
                        class="de.ufinke.cubaja.config.TimestampProvider"/>

<configProperty name="path" value="/home/path/foo"/>
<configProperty name="date" provider="timestamp">
  <parm pattern="yyyyMMdd"/>
</configProperty>

<output name="${path}/output_${date}.txt"/>
```

Includes

The special element 'configInclude' includes another resource which is loaded by the ResourceLoader. Attention: The root element of the included XML is discarded! Only the children of the root element are processed.

A resource called 'databases.xml' ...

```
<config>

  <db_foo url="..."/>
  <db_bar url="..."/>

</config>
```

... is included in a main configuration resource:

```
<config>

  <configInclude include="databases.xml"/>

</config>
```

sql

Connect to a database

The parameters which are needed to connect to a database are configurable data. Therefore the XML configuration should contain an element to define the connection. The corresponding class is DatabaseConfig. It is recommended to set the log attribute to true.

A Database object wraps the JDBC Connection. We pass the DatabaseConfig with the connection properties in the constructor.

```
Database db = new Database(config.getDatabase());
```

With a default DatabaseConfig autocommit is set to false. We have to commit or rollback the current transaction explicitly in our application.

```
db.commit();
```

When the Database instance is no longer needed, it should be closed.

```
db.close();
```

Write SQL

SQL may be written in a simple String (recommended only for very short statements) or assembled in an Sql object.

The Sql class offers several append methods to add lines or other SQL fragments. In contrast to concatenation of simple strings we don't need take care about space characters or imbedded comments.

```
Sql sql = new Sql().
  append("select   name").
  append("       , birthday").
  append("from     customer");
```

If we want to use the SQL - without quotes and brackets - in other tools, we place the statement in a file besides our Java source files. The Sql class is able to read this file as resource. To locate the resource, it has not only to know the file's name, but also the package where it resides.

When we have a resource named 'select.sql' in the same package as the class which contains the Sql object, we may code:

```
Sql sql = new Sql(getClass(), "select");
```

The purpose of the Sql class is to parse and format a statement. It can't be executed directly. For execution, we need the appropriate methods of the Database class:

```
Query query = db.createQuery(sql);
```

In most cases, we don't need an explicit Sql object, because the Database class offers a shortcut:

```
Query query = db.createQuery(getClass(), "select");
```

Variables

The SQL may contain variables in the form of ':*variableName*'. For JDBC, the variables are replaced by '?'. Within Query and Update, which are both subclasses of PreparedSql, we may use position numbers or variable names.

Have a look on the following sections for examples.

Perform a query

The most convenient way to process a ResultSet is to use a cursor which reads all rows and creates a data object for every row automatically.

We store a select statement in a resource named 'select_customers.sql':

```
select   name
       , birth_date
       , debit
from     customers
where    debit >= :limit
```

A data class corresponds to a row. The column names of the select statement have to match the names of setter methods in the data class. Underlines are replaced by camelcase names.

```
public class CustomerData {

  private String name;
  private Date birthDate;
  private double debit;

  // must have a parameterless public constructor

  public CustomerData() {

  }

  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  // ... other getter and setter methods ...
}
```

The application's code looks like:

```
// create a prepared statement
Query query = db.createQuery(getClass(), "select_customers");

// set the variables (if there are any variables)
query.setDouble("limit", config.getLimit());

// execute the statement and process the rows of the result set
for (CustomerData customer : query.cursor(CustomerData.class)) {
  process(customer);
}

// close the prepared statement
query.close();
```

Retrieve a single row

There is a shortcut to read the result of a query with a one-row result set.

```
Integer count = db.select("select count(*) from customers", Integer.class);
```

Perform insert, update or delete

We have an insert statement in a resource named 'insert_customers.sql':

```
insert into customers (
  name
, birth_date
, debit
) values (
  :name
, :birth_date
, :debit
)
```

There is a corresponding data class like in the query example above.

The application code for a mass insert looks like this:

```
// init: create a prepared statement
Update insert = db.createUpdate(getClass(), "insert_customers");

// for every row: set variables and add data to batch
CustomerData customer = createNextCustomer();
insert.setVariables(customer);
insert.addBatch();

// finish: write to database and cleanup
insert.executeBatch();
insert.close();

db.commit();
```

An Update may be executed immediately for every single row (not recommended for high volume data):

```
Update delete = db.createUpdate("delete from customers where debit > :limit");
delete.setDouble("limit", config.getLimit());
delete.executeUpdate();
delete.close();

db.commit();
```

Execute other SQL

Other SQL statements than select, insert, update or delete may be executed with a Database instance:

```
db.execute("revoke select from big_brother");
```

Those statements may not contain variables!

CSV

CsvReader

The task is to read the following CSV file:

```
Nachname;Geburtstag
Müller;13.01.1830
Meier;23.12.2002
```

There are several ways to read a CSV source with a CsvReader. The most flexible is to use a CsvConfig.

```
<customer file="/path_to_data/input.csv"
          separator=";"
          datePattern="dd.MM.yyyy">
  <col name="name" header="Nachname"/>
  <col name="birth_date" header="Geburtstag"/>
</customer>
```

There is an adequate data class:

```
public class Person {

  private String name;
  private Date birthDate;

  // ... parameterless public constructor and setter / getter methods ...

}
```

The application code looks similar to that of an SQL query:

```
CsvReader reader = new CsvReader(config.getCsv());
for (Person person : reader.cursor(Person.class)) {
  process(person);
}
reader.close();
```

If the CSV source was created by men instead by a reliable tool, there may be wrong data which can't be converted to the type of the data class' setter method. There is a way to handle erroneous data:

```
CsvReader reader = new CsvReader(config.getCsv());

CollectingErrorHandler errorHandler = new CollectingErrorHandler();
reader.setErrorHandler(errorHandler);

for (Person person : reader.cursor(Person.class)) {

  if (errorHandler.hasErrors()) {

    // person object exists, but its content is incomplete
    logBadRow(errorHandler);
    errorHandler.reset();

  } else {

    process(person);

  }

}

reader.close();
```

CsvWriter

As with CsvReader, there are several ways to write CSV output. If we have the same configuration as in the example above, we may produce a file with a code like this:

```
// init
CsvWriter writer = new CsvWriter(config.getCsv));

// for every row
writer.writeRow(person);

// finish
writer.close();
```

sort

Sorter

The Sorter class is able to sort a high volume of objects. If necessary, objects are serialized to a temporary file. Therefore, the type of the data objects have to implement the Serializable interface. Here's an example how to sort data objects:

```
// create a Comparator
Comparator<CustomerData> comparator = new Comparator<CustomerData>() {
  public int compare(CustomerData a, CustomerData b) {
    return Util.compare(a.getBirthday(), b.getBirthday());
  }
};
// create the Sorter
Sorter<CustomerData> sorter = new Sorter<CustomerData>(comparator);
// for every data object: put it to the sort process
sorter.add(getNextData());
// get the sorted objects
for (CustomerData customer : sorter) {
  process(customer);
}
```

It is recommended to use the Sorter constructor with a SortConfig parameter to gain more control over some parameters at runtime.

Matcher

We may use Matcher for the classic master/update-pattern or for other tasks where any number of sources have to be matched by key. The sources must be sorted before they are added to a Matcher.

Often, the source data types differ, but contain a common key. Then we need a KeyFactory to extract the key from the data object.

```
// setup
Matcher<Long> matcher = new Matcher<Long>(new NaturalComparator<Long>());
MatchSource<Customer> cs = matcher.addSource(customers(), customerKeyFactory());
MatchSource<Account> as = matcher.addSource(accounts(), accountKeyFactory());
// iterate distinct customer numbers
for (Long customerNumber : matcher) {
  if (cs.matches() && as.matches()) {
    Customer customer = cs.get(); // get matching customer object from source
    List<Account> accounts = as.getList(); // get matching accounts from source
    process(customer, account);
  }
}
// key factory for customer source
private KeyFactory<Customer, Long> customerKeyFactory() {
  return new KeyFactory<Customer, Long>() {
    public Long createKey(Customer data) {
      return data.getCustomerNumber();
    }
  };
}
```

```
// key factory for account source
private KeyFactory<Account, Long> accountKeyFactory() {
  return new KeyFactory<Account, Long>() {
    public Long createKey(Account data) {
      return data.getCustomerNumber();
    }
  };
}

// some provider for sorted customer objects
private Iterable<Customer> getCustomers() {

}

// some provider for sorted account objects
private Iterable<Account> getAccounts() {

}
```

util

An application's main class

Typically, the static main method in an application's main class instantiates an object and calls a worker method. Any Exceptions should be catched and logged. In case of failure, the exit code should be different from 0. It is useful to have a start and an end message in the log.

The abstract class Executor handles these tasks. All we have to do is to implement the execute method and call start:

```
import de.ufinke.cubaja.util.Executor;

public class Main extends Executor {

  static public void main(String[] args) {

    new Main().start();
  }

  protected void execute() throws Exception {

    // ... do the work ...
  }
}
```

Don't forget to have Apache CommonsLogging in the classpath and to activate a logging framework like log4j.