ResultSet Mappers

A common operation while working with a databases is the mapping of row data to objects in code. This has always seemed very repetitive and error prone. The Vanilla library provides a simple DSL-based solution for quickly implementing code that extracts, and optionally transforms, the data from the database table (via a `ResultSet`) and uses it to populate the object representation of your choice.

There are two implementations available, one that is dynamic and does all the extraction and conversion work at runtime, and a second which pre-compiles the extraction code at compile time using AST transformations.

Mapping Style

There are two styles of mapping available to both implementations of the mapping framework:

- `MappingStyle.IMPLICIT` - All properties of the mapped object are mapped by default, though the DSL may be used to alter property extraction or ignore properties.
- `MappingStyle.EXPLICIT` - No properties of the mapped object are mapped by default. All mapping must be explicitly configured by the DSL.

DSL

The DSL used by the `ResultSetMapper` framework is shared between both the runtime and Compiled Implementations. There are five supported mapping statements:

`ignore <property-name>[,<property-name>…]`
: Used to ignore one or more of the mapped objects properties. This operation is meaningless for explicit mappers.

`map <property-name>`
: Maps the specified object property with the default extraction type (uses `from <field-name>`) and no conversion.

`map <property-name> from[TYPE] <field-name>`
: Maps the specified object property with the specified extraction type and field name with no conversion.

`map <property-name> from[TYPE] <field-name> using <closure>`
: Maps the specified object property with the specified extraction type with the provided conversion closure.

`map <property-name> using <closure>`
: Maps the specified object property with the default extraction type (uses `from <field-name>`) with the provided conversion closure.

Auto-conversions of the property name to a database field name are done as camel-case to underscore-based notation similar to the following:

```
something           --becomes--> something
somethingInteresting --becomes--> something_interesting
anotherPropertyName  --becomes--> another_property_name
```

The extraction types are defined analogous to the getter methods provided in the `ResultSet` but with the "from" prefix in place of "get". The supported extractors are: `from`, `fromObject`, `fromString`, `fromBoolean`, `fromByte`, `fromShort`, `fromInt`, `fromLong`, `fromFloat`, `fromDouble`, `fromBytes`, `fromDate`, `fromTime`, `fromTimestamp`, `fromAsciiStream`, `fromUnicodeStream`, `fromBinaryStream`, `fromCharacterStream`, `fromBigDecimal`, `fromRef`, `fromBlob`, `fromClob`, `fromArray`, `fromURL`, `fromMapper`.

Each extractor takes either the database field name or position value as a required parameter and extracts the database value as you would expect from its name, though you may want to consult the `ResultSet` documentation for more details on the behavior of a specific getter method.

The exceptional extractor case is the `fromMapper` extractor. This extractor accepts a `ResultSetMapper` instance which will be used to extract the object for the specified target property. An example of such a mapping would be the following:

```
map 'foo' fromMapper Foo.mapper('foo_')
```

where `Foo.mapper(String)` is a mapper factory method.

The conversion closures configured by the `using` clause are standard Groovy closures that are passed in one or two arguments:

- 1-argument: the extracted database value at runtime which allows the closure to do additional conversion of the database value.
- 2-argument: the reference to the `ResultSet` being extracted.

The closure should return the desired mapped value for the property.

A sample of the DSL pulled from the unit tests is shown below:

```
ignore 'bankPin', 'pet'
ignore 'children'
map 'birthDate' fromDate 'birth_date'
map 'age' from 2 using { a -> a - 5 }
map 'name' from 'name'
```

Runtime Implementation

The runtime implementation uses the static `mapper(Class, MappingStyle, Closure)` method from `ResultSetMapperBuilder` as its entry point. The object type being mapped is provided, as well as the mapping style and an optional configuration closure (DSL). Using our example code from the DSL section, you would have something like the following for an `IMPLICIT` mapping:

```
ResultSetMapper mapper = ResultSetMapperBuilder.mapper(Person){
Ê   ignore 'bankPin', 'pet'
Ê   ignore 'children'
Ê   map 'birthDate' fromDate 'birth_date'
Ê   map 'age' from 2 using { a -> a - 5 }
Ê   map 'name' from 'name'
}
```

An `IMPLICIT` mapping has a very clean format for cases where your mapped object aligns with the database fields and types. You can simply map it with no configuration DSL.

```
ResultSetMapper mapper = ResultSetMapperBuilder.mapper(Person)
```

An `EXPLICIT` mapping for the same configuration could be something like:

```
ResultSetMapper mapper = ResultSetMapperBuilder.mapper(Person, MappingStyle.EXPLICIT){
Ê   map 'birthDate' fromDate 'birth_date'
Ê   map 'age' from 2 using { a -> a - 5 }
Ê   map 'name' using { n-> "Name: $n"}
}
```

The mapping, extraction and conversion operations are all performed at runtime. Some pre-computing or caching may be performed; however, the bulk of the mapping is done at runtime with each mapped object creation.

The created mappers are reusable and thread-safe.

TIP	While the dynamic runtime implementation is fully supported and maintained, it is generally advisable to use the Compiled Implementation to generation more performant code.

Compiled Implementation

The Compiled Implementation uses the `InjectResultSetMapper` annotation to inject a configured `ResultSetMapper` into a field or method of a class.

Using the example from the Runtime Implementation section, we could have an `IMPLICIT` mapper available as a static method as follows:

```
class Mappers {

Ê   @InjectResultSetMapper(
Ê       value=Person,
Ê       config={
Ê           ignore 'bankPin', 'pet'
Ê           ignore 'children'
Ê           map 'birthDate' fromDate 'birth_date'
Ê           map 'age' from 2 using { a -> a - 5 }
Ê           map 'name' from 'name'
Ê       }
Ê   )
Ê   static ResultSetMapper personMapper(){}
}
```

The `personMapper()` method returns the same `ResultSetMapper` instance for every call and the mapper itself is configured at compile-time via AST transformations so that the extraction calls are generated at compile-time rather than for each mapping call; however, the conversion closures are still executed at runtime.

For `IMPLICIT` mappings where the object property names and type align with the database fields, you can have a very simple `IMPLICIT` mapping configuration:

```
class Mappers {

Ê   @InjectResultSetMapper(Person)
Ê   static ResultSetMapper personMapper(){}
}
```

Creation of `EXPLICIT` mappers follows a similar style:

```
class Mappers {

Ê   @InjectResultSetMapper(
Ê       value=Person,
Ê       style=MappingStyle.EXPLICIT,
Ê       config={
Ê           map 'birthDate' fromDate 'birth_date'
Ê           map 'age' from 2 using { a -> a - 5 }
Ê           map 'name' using { n-> "Name: $n"}
Ê       }
Ê   )
Ê   static ResultSetMapper personMapper(){}
}
```

The method or field used to provide the compiled mapper does not need to be static; however, it is advisable, since the underlying instance created will be a static field of the enclosing class.

The generated mapper class is created in the same package as the mapped object type. The `InjectResultSetMapper` annotation also provides a `name` property which may be used to provide an alternate name for the generated mapper class. By default, the name of the mapped object type is used with the added "Mapper" suffix.

The generated mapper class may be used directly; however, the method or field injection is required to create the mapper class and it is recommended to use the field or method as your access point to the generated class.

NOTE When using a factory method to create the `ResultSetMapper` instances, note that the method is annotated with `@Memoized` so that multiple calls to the method will return the same instance. This is also true when a prefix is used (method parameter); calls to the method with the same prefix value will return the same instance of the mapper.

Usage

Once you have created a `ResultSetMapper`, you can use it anywhere you have a `ResultSet` or with the `groovy.sql.Sql` class as follows:

```
def sql = Sql.newInstance(db.url, db.user, db.password, db.driver)

def people = []

sql.eachRow('select * from people'){ rs->
Ê   people << Mappers.personMapper().call(rs)
}
```

Where we are using the compiled mapper implementation exposed by the `Mappers.personMapper()` method. The `call(ResultSet)` method performs the mapping of the current `ResultSet` data into a `Person` object.

An alias to the `call(ResultSet)` method is provided as the `mapRow(ResultSet, int)` method, which allows for simple interaction with the Spring Framework as a `RowMapper`, for example:

```
List<Person> people = jdbcTemplate.query(
Ê   'select * from people',
Ê   Mappers.personMapper() as RowMapper
)
```

The `ResultSetMapper` may be cast as a `RowMapper` and then used as one.

Field Name Prefix

Both implementations of the `ResultSetMapper` support an optional field name prefix, which will be applied to all field names on field lookup. So for example a property named "somethingInteresting" with a prefix of "foo_" would look for a database field named "foo_something_interesting". If no prefix is specified an empty string will be used.

The prefix is *not* overridden by the "fromXXX" mapping methods. This is intentional since the main goal of the prefix support is to allow the same mapper to be useful across different mapping scenarios such as in a join where each column name in the query result has been prefixed by some common known prefix.

This does not remove the ability to use numerical from indices. If a number is detected in the "from" statement, the prefix will not be applied.

Mapping Properties from Multiple Fields

It is often necessary to map an object field that is not directly represented in the database fields or is made up of more than on field. This is can be done by using the 2-argument version of the "using" closure. Consider the example of a `GeoLocation` object being mapped:

```
@Canonical
class GeoLocation {
Ê   double latitude
Ê   double longitude
}

class Somewhere {
Ê   GeoLocation location
}
```

The DSL code would look something like the following:

```
{
Ê   map 'location' fromDouble 'latitude' using { lat,rs->
Ê       new GeoLocation(lat, rs.getDouble('longitude'))
Ê   }
}
```

This allows the flexibility to populate the target object from various database fields.