Object Mappers

When working with legacy or external code-bases you may run into an API where you end up doing a lot of domain object mapping, from one object format to another. The Vanilla Object Mappers framework can help simplify the mappings. It comes in two forms, a dynamic runtime implementation and a compile-time generated implementation.

Generated `ObjectMapper` instances are stateless and thread-safe.

Object Mapper DSL

The DSL used by the `ObjectMapper` framework is shared between both the runtime and compiled implementations. There are four supported mapping statements:

`map <source-name>`

> Maps the specified source property into the destination property of the same name, with no conversion.

`map <source-name> into <destination-name>`

> Maps the specified source property into the specified destination property, with no conversion.

`map <source-name> into <destination-name> using <closure>`

> Maps the specified source property into the specified destination property, with the given conversion closure.

`map <source-name> using <closure>`

> Maps the specified source property int the destination property of the same name, with the given conversion closure.

The conversion closures configured by the `using` clause are standard Groovy closures that are allowed to accept up to three arguments:

- Argument one - source property value
- Argument two - source object reference
- Argument three - destination object reference

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

```
map 'name'
map 'age' into 'years'
map 'startDate' using { Date.parse('MM/dd/yyyy', it) }
map 'birthDate' into 'birthday' using { LocalDate d -> d.format(BASIC_ISO_DATE) }
```

Runtime Mappers

The `RuntimeObjectMapper` is created using its static `mapper(Closure)` method, where the `Closure` contains the configuration DSL. An example would be something like:

```
ObjectMapper inividualToPerson = RuntimeObjectMapper.mapper {
    map 'id'
    map 'givenName' into 'firstName'
    map 'familyName' into 'lastName'
    map 'birthDate' using { d-> LocaleDate.parse(d) }
}
```

Compiled Mappers

Sometimes you need to squeeze a bit more performance out of a mapping operation, or you just want to generate cleaner code. The `InjectCompiledObjectMapper` annotation is used to annotate a field or property to inject a compile-time generated `ObjectMapper` implementation (using AST Transformations) based on the supplied DSL configuration.

Using the compiled approach is as simple as the runtime approach, you just write the DSL code in the `@InjectObjectMapper` annotation on a method, or field:

```
class ObjectMappers {

    @InjectObjectMapper({
        map 'id'
        map 'givenName' into 'firstName'
        map 'familyName' into 'lastName'
        map 'birthDate' using { d-> LocaleDate.parse(d) }
    })
    static final ObjectMapper personMapper(){}
}
```

When the code compiles, a new implementation of ObjectMapper will be created and installed as the return value for the personMapper() method. The compiled version of the DSL has all of the same functionality of the dynamic version except that it does not support using ObjectMappers directly in the using command; however, a workaround for this is to use a closure to wrap another `ObjectMapper`.

Usage

Once you have created an `ObjectMapper` using either the runtime or compile-time implementations, objects can be copied using the `copy(Object,Object)` method, which will copy the properties of the source object into the destination object according to the configured mapping information in the DSL.

```
def people = individuals.collect { indiv->
    Person person = new Person()
    individualToPerson.copy(indiv, person)
    person
}
```

The `Person` objects will now contain the correctly mapped property values from the `Individual` objects. The `ObjectMapper` also has a `create(Object, Class)` method to help simplify the use case where you are simply creating a new populated instance of the destination object:

```
def people = individuals.collect { indiv->
    individualToPerson.create(indiv, Person)
}
```

When using the `create()` method, the destination object must have a default constructor.

The third, slightly more useful option in this specific collector case is to use the `collector(Class)` method, which again takes the type of the destination object (with a default constructor):

```
def people = individuals.collect( individualToPerson.collector(Person) )
```

The collector(Class) method returns a Closure that is also a shortcut to the conversion code shown previously. It’s mostly syntactic sugar, but it is nice and clean to work with.