TYPO3.Flow - Feature # 32106

Status:	Accepted	**Priority:**	Should have
Author:	Marc Neuhaus	**Category:**	Property
Created:	2011-11-26	**Assigned To:**	
Updated:	2013-06-24	**Due date:**	
PHP Version:			
Has patch:	Yes		
Complexity:	hard		
Subject:	Support for Object source in PropertyMapper		

Description

Hi,

i'm trying to ditch my old TypeConverters in my Admin Package. I'm trying to replace them with the core TypeConverters by added some Converters to convert from float, boolean, etc to string.

One thing i can't do with the default propertyMapper is to convert from a object like DateTime or a Collection.

Currently i use an extended version of the propertyMapper that allows objects as source type in the determineSourceType function.
I attached a diff for the changes i made.

Would be great to get this into core.

Greetings Marc

History

#1 - 2011-11-28 06:01 - Sebastian Kurfuerst

Hey Marc,

what are you trying to archieve? Do you want to convert an object to other objects; or objects to strings etc?

The reason why I am asking is that Christopher and I have made a concept of making the Property Mapper aware of the reverse direction "Object --> Simple Type"; which makes the behavior again deterministic.

If you need the "Object to Object" conversion case, I'd be curious about some more detailed use-cases.

I am afraid that the "get_class" which you do in the patch is not enough, as it does not take object hierarchies / inheritance into account...

Greets,
Sebastian

#2 - 2011-11-28 10:08 - Marc Neuhaus

I'm trying to archive a conversion back to a simple string to display the Entities, Collections, Datetimes,... without additional Fluid markup.

After implementing the Entity -> String converter yesterday i realized as well, that the get_class doesn't suffice for this because of the undetermined amount of Entity classes it's capable to convert.

What i have done now is to return "object" from the determineSourceType function and make the detailed check in the canConvertFrom.

```
    protected function determineSourceType($source) {
        if (is_string($source)) {
            return 'string';
        } elseif (is_array($source)) {
            return 'array';
        } elseif (is_float($source)) {
            return 'float';
        } elseif (is_integer($source)) {
            return 'integer';
        } elseif (is_bool($source)) {
            return 'boolean';
        } elseif (is_object($source)) {
            return "object";
        } else {
            throw new \TYPO3\FLOW3\Property\Exception\InvalidSourceException('The source is not of type string, array, object, float, integer or
boolean, but of type "' . gettype($source) . '"', 1297773150);
        }
    }
```

Here are 3 examples:

DateTime

```
    protected $sourceTypes = array('object');
    public function canConvertFrom($source, $targetType) {
        if(is_object($source) && $source instanceof \DateTime && $targetType == "string")
            return TRUE;
    }
```

Collection

```
    protected $sourceTypes = array('object', 'array', 'ArrayObject', 'SplObjectStorage', 'Doctrine\Common\Collections\Collection',
'Doctrine\Common\Collections\ArrayCollection', 'Doctrine\ORM\PersistentCollection');
    public function canConvertFrom($source, $targetType) {
        if(is_array($source))
            return TRUE;
        return in_array(get_class($source), $this->sourceTypes);
    }
```

Object

```
    protected $sourceTypes = array('object');
    public function canConvertFrom($source, $targetType) {
        return method_exists($source, "__toString");
    }
```

Greetings Marc

#3 - 2011-11-29 09:06 - Sebastian Kurfuerst

Thanks Marc for your detailed explanations,

As already said, Christopher and me have made a very flexible draft about converting from objects back to simple types, also taking possible recursion etc. into account.

Because of this, I'd rather not integrate your patch right now, but instead implement a clean solution later. If you'd like to work on this, let me know; then I'll post our ideas in a more detailed fashion.

Hope that makes sense for you :-)

Greets, Sebastian

#4 - 2011-11-29 09:41 - Marc Neuhaus

Yea, sure post you ideas. I'll take a shot at implementing them :)

Greetings Marc

#5 - 2011-12-01 16:46 - Sebastian Kurfuerst

Concept for implementing a reverse direction into the Property Mapper

We'll take a little mathematical look at the Property Mapper, as it helps (at least for me) to get its functionality clear.

Mathematical view of Property Mapping

- This is already implemented; it's just for clarification.
- The property mapper is a function **p** which converts from source to dest: dest = p(source)
- We want to implement the other direction (from dest to source), so we are looking for a reverse function p-1 with the following properties:
  - p-1(p(s)) = s does NOT hold in all cases, as we show by example:
    - let's convert the string "x1" into a number.
    - This number will be 0 because of invalid input.
    - Thus, if the number 0 is converted back into a string, this is "0". And "0" != "x1"
  - Instead, the following Invariant has to be true **always**: p(source) = p(p-1(p(source)))
    - This reads as: "If you convert from source to destination, that must result in the same as converting from source to dest, back to source and back to dest again"
    - Take the number example from above, this is true here.
- The property mapping function **p** uses multiple **type converters**t1...tn

Type Converter

- Each type converter contains:
  - n source types
  - 1 target type
  - canConvert function
- It works in the following way:
  1. First, the Type Converter is asked for the nested **source properties** of the input, and their target types.
     1. Then, the property mapper converta the nested source properties recursively.
  2. Then, the target type is built, using the already converted sub-properties.

Extending Type Converters into the other direction

- Basically, this also works when converting from target to source, in the same manner:

    - Each type converter is responsible for **1** incoming type (which is the **target type from above**)
    - Each type converter is responsible for **n** outgoing types (which are the **source types from above**)
- Then, it should work in the following way (similar as above):
    1. First, the type converter is asked for the nested properties which need to be converted to simple types; and their expected simple type
        1. Then, the property mapper converts the nested target properties recursively into the expected simple type
    2. Then, the source type is built, using the already-converted sub-properties

Summary

- sourceTypes of simpleType->object direction become **target types** of object->simpleType direction
- targetType of simpleType->object direction becomes **source Type** of object->simpleType direction
- There need to be two seperate canConvert(From|To) functions, one for each direction
- There need to be two seperate get(Source|Target)ChildPropertiesToBeConverted functions, one for each direction
- There need to be two seperate getTypeOfChildProperty / getTargetTypeOfChildProperty functions, one for each direction
- There need to be two seperate convertFrom|convertTo methods, one for each direction.

The API for the "Back" direction should be specified in a **new** interface; such that a type converter can decide if it should be only used in the one or in both directions.

Development Process

- Development of the reverse direction should be done test-driven, convered with functional and unit tests.

#6 - 2011-12-05 06:14 - Sebastian Kurfuerst

Hey Marc,

If you need any help, do not hesitate to contact me :)

Greets,
Sebastian

#7 - 2012-01-12 07:16 - Sebastian Kurfuerst

Hey Marc,

do you need any help on this, or anything else I can do to support you here?

Greets,
Sebastian

#8 - 2012-01-12 11:27 - Marc Neuhaus

Hey Sebastian,

unfortunately i've been drowning in Work the last 3 Weeks :/

i promise to get back to this as soon as i can!

Greetings Marc

#9 - 2012-01-12 13:01 - Sebastian Kurfuerst

cool Marc, thanks for the update :-)

Greetings, Sebastian

#10 - 2012-02-07 14:12 - Sebastian Kurfuerst

Hey,

any news already? :)

Greets, Sebastian

#11 - 2012-02-14 10:57 - Marc Neuhaus

Hey Sebastian,

sorry for my late Reply.

Sadly i'm still drowning in an ocean of work and can't see an end at the moment.
My FLOW3 Admin is stalling because of this as well currently :/
I think for now i'll have to postpone this indefinitely, i'm very sorry.

Greetings Marc

#12 - 2012-03-09 09:17 - Adrian Föder
- Status changed from New to Accepted
- Complexity set to hard

(I felt free to set to "Accepted" because I think there's enough agree for the need for it)

Sebastian, I read a bit through your summary above; a basic question came up: it seems you tend to use the same class/"file" for both directions, by adding additional methods.
Although that seems fine in theory, I think we may come up to pretty large classes, especially when adding helper methods.
For the reason I didn't get the big picture of the mapper (although I already digged through when writing my OpenGraph package) let me ask the question: could the current concrete TypeConverter classes **functionality** be reused? Because if not, it might be really an idea to add an additional namespace "NonPrimitive" (or such) where the opposite directions are implemented, with the same function set, but for the opposite direction of conversion.

In the mailing list, you named a "context", allow me quoting you here regarding the idea of an Object -> Json mapper:

> - *Extend Property Mapping to support the reverse direction "object ->*

> Simple Type", and also under a specific "context".
> - *For JSON, you'd then call property mapping "object -> array" under*

the context "json"
Where do you exactly see this context "attached"? Or, what **is** a context? It could even be an object itself which contains the converting directives; would you agree?
So there might be a pretty raw Object -> Array converter, but intercepted by the \TYPO3.FLOW3\Property\TypeConverter\Context\JsonContext?

#13 - 2013-06-24 09:59 - Sebastian Kurfuerst

> *Sebastian, I read a bit through your summary above; a basic question came up: it seems you tend to use the same class/"file" for both directions, by adding additional methods.*
> *Although that seems fine in theory, I think we may come up to pretty large classes, especially when adding helper methods.*
> *For the reason I didn't get the big picture of the mapper (although I already digged through when writing my OpenGraph package) let me ask the question: could the current concrete TypeConverter classes **functionality** be reused?*

Good question. Probably not. However the "to object" and "from object" directions must be kept closely in sync; that's why I am still a little bit in favour of having them inside one class or at least closely related.

> *Because if not, it might be really an idea to add an additional namespace "NonPrimitive" (or such) where the opposite directions are implemented, with the same function set, but for the opposite direction of conversion.*

I'd be OK with having a "Reverse*" naming or the like, or "ArrayReverseConverter", "ReverseArrayConverter" or anything like that.

> *In the mailing list, you named a "context", allow me quoting you here regarding the idea of an Object -> Json mapper:*
>> - *Extend Property Mapping to support the reverse direction "object ->*
>
> *Simple Type", and also under a specific "context".*
>>
>> - *For JSON, you'd then call property mapping "object -> array" under*
>
> *the context "json"*
>
> *Where do you exactly see this context "attached"? Or, what **is** a context? It could even be an object itself which contains the converting directives; would you agree?*

A "Context" for me would be just a string (maybe hierarchical like "json" or "json/specialconversion") which is attached to the property mapping configuration (I think).

The "context" would influence the choice of the type converter, just like source, destination type and priority.

> *So there might be a pretty raw Object -> Array converter, but intercepted by the \TYPO3.FLOW3\Property\TypeConverter\Context\JsonContext?*

Hm, I feel that's dangerous because it would make the conversion process quite a lot more intransparent.

Greets,
Sebastian

Files

PropertyMapper.diff	449 Bytes	2011-11-26	Marc Neuhaus