Implementing the Tweet\Type class

As said in the introduction, the Type class of a FieldType must implement `eZ\Publish\SPI\FieldType\FieldType` (later referred to as "FieldType interface").

All native FieldTypes also extend the `eZ\Publish\Core\FieldType\FieldType` abstract class, that implements this interface, and provides implementation facilities through a set of abstract methods of its own. In this case, Type classes implement a mix of methods from the FieldType interface and from the abstract FieldType .

Let's go over those methods and their implementation.

Identification method: `getFieldTypeIdentifier()`

It must return the string that uniquely identifies this FieldType (DataTypeString in eZ Publish 4). We will use "`eztweet`":

eZ/FieldType/Tweet/Type

```
public function getFieldTypeIdentifier()
{
   return 'eztweet';
}
```

Value handling methods: `createValueFromInput()` and `checkValueStructure()`

Both methods are used by the abstract FieldType implementation of `acceptValue()`. This FieldType interface method checks and transforms various input values into the type's own Value class: `eZ\FieldType\Tweet\Value`. This method must:

- either return the Value object it was able to create out of the input value,
- or return this value untouched. The API will detect this, and inform that the input value was not accepted.

The only acceptable value for our type is the URL of a tweet (we could of course imagine more possibilities). This should do:

```
protected function createValueFromInput( $inputValue )
{
   if ( is_string( $inputValue ) )
   {
       $inputValue = new Value( array( 'url' => $inputValue ) );
   }

   return $inputValue;
}
```

Use this method to provide convenient ways to set an attribute's value using the API. This can be anything from primitives to complex business objects.

Next, we will implement `checkValueStructure()`. It is called by the abstract FieldType to ensure that the Value fed to the Type is acceptable. In our case, we want to be sure that `Tweet\Value::$url` is a string:

```
protected function checkValueStructure( BaseValue $value )
{
   if ( !is_string( $value->url ) )
   {
       throw new eZ\Publish\Core\Base\Exceptions\InvalidArgumentType(
           '$value->url',
           'string',
           $value->url
       );
   }
}
```

Yes, we execute the same check than in `createValueFromInput()`. But both methods aren't responsible for the same thing. The first will, *if given something else than a Value of its type*, try to convert it to one. `checkValueStructure()` will always be used, even if the FieldType is directly fed a `Value` object, and not a string.

Value initialization: `getEmptyValue()`

This method provides what is considered as an empty value of this type, depending on our business requirements. No extra initialization is required in our case.

```
public function getEmptyValue()
{
   return new Value;
}
```

If you run the unit tests at this point, you should get about five failures, all of them on the `fromHash()` or `toHash()` methods.

Validation methods: `validateValidatorConfiguration()` and `validate()`

The Type class is also responsible for validating input data (to a `Field`), as well as configuration input data (to a `FieldDefinition`). In this tutorial, we will run two validation operations on input data:

- validate submitted urls, ensuring they actually reference a twitter status;
- limit input to a known list of authors, as an optional validation step.

`validateValidatorConfiguration()` will be called when an instance of the FieldType is added to a ContentType, to ensure that the validator configuration is valid. For a TextLine (length validation), it means checking that both min length and max length are positive integers, and that min is lower than max.

When an instance of the type is added to a content type, `validateValidatorConfiguration()` receives the configuration for the validators used by the Type as an array. It must return an array of error messages if errors are found in the configuration, and an empty array if no errors were found.

For TextLine, the provided array looks like this:

```
array(
   'StringLengthValidator' => array(
       'minStringLength' => 0,
       'maxStringLength' => 100
   )
);
```

The structure of this array is totally free, and up to each type implementation. We will in this tutorial mimic what is done in native FieldTypes:

Each level one key is the name of a validator, as acknowledged by the Type. That key contains a set of parameter name / parameter value rows. We must check that:

- all the validators in this array are known to the type
- arguments for those validators are valid and have sane values

We do not need to include mandatory validators if they don't have options. Here is an example of what our Type expects as validation configuration:

```
array(
   'TweetAuthorValidator' => array(
       'AuthorList' => array( 'johndoe', 'janedoe' )
   )
);
```

The configuration says that tweets must be either by johndoe or by janedoe. If we had not provided TweetAuthorValidator at all, it would have been ignored.

We will iterate over the items in `$validatorConfiguration`, and:

- add errors for those we don't know about;
- check that provided arguments are known and valid:
    - TweetAuthorValidator accepts a non-empty array of valid twitter usernames

```
public function validateValidatorConfiguration( $validatorConfiguration )
{
   $validationErrors = array();

   foreach ( $validatorConfiguration as $validatorIdentifier => $constraints )
   {
       // Report unknown validators
       if ( !$validatorIdentifier != 'TweetAuthorValidator' )
       {
           $validationErrors[] = new ValidationError( "Validator
'$validatorIdentifier' is unknown" );
           continue;
       }

       // Validate arguments from TweetAuthorValidator
       if ( !isset( $constraints['AuthorList'] ) || !is_array(
$constraints['AuthorList'] ) )
       {
           $validationErrors[] = new ValidationError( "Missing or invalid AuthorList
argument" );
           continue;
       }

       foreach ( $constraints['AuthorList'] as $authorName )
       {
           if ( !preg_match( '/^[a-z0-9_]{1,15}$/i', $authorName ) )
           {
               $validationErrors[] = new ValidationError( "Invalid twitter username"
);
           }
       }
   }


   return $validationErrors;
}
```

`validate()` is the method that runs the actual validation on data, when a content item is created with a field of this type:

```
   public function validate( FieldDefinition $fieldDefinition, SPIValue $fieldValue )
   {
       $errors = array();

       if ( $this->isEmptyValue( $fieldValue ) )
       {
           return $errors;
       }

       // Tweet Url validation
       if ( !preg_match( '#^https?://twitter.com/([^/]+)/status/[0-9]+$#',
$fieldValue->url, $m ) )
           $errors[] = new ValidationError( "Invalid twitter status url %url%", null,
array( $fieldValue->url ) );

       $validatorConfiguration = $fieldDefinition->getValidatorConfiguration();
       if ( isset( $validatorConfiguration['TweetAuthorValidator'] ) )
       {
           if ( !in_array( $m[1],
$validatorConfiguration['TweetAuthorValidator']['AuthorList'] ) )
           {
               $errors[] = new ValidationError(
                   "Twitter user %user% is not in the approved author list",
                   null,
                   array( $m[1] )
               );
           }
       }

       return $errors;
   }
```

First, we validate the url with a regular expression. If it doesn't match, we add an instance of `ValidationError` to the return array. Note that the tested value isn't directly embedded in the message but passed as an argument. This ensures that the variable is properly encoded in order to prevent attacks, and allows for singular/plural phrases using the 2nd parameter.

Then, if our FieldType instance's configuration contains a `TweetAuthorValidator` key, we check that the username in the status url matches one of the valid authors.

Metadata handling methods: `getName()` and `getSortInfo()`.

FieldTypes require two methods related to Field metadata:

- `getName()` is used to generate a name out of a field value, either to name a content item (naming pattern in legacy) or to generate a part for an URL Alias.
- `getSortInfo()` is used by the persistence layer to obtain the value it can use to sort & filter on a field of this type

Obviously, a tweet's full URL isn't really suitable as a name. Let's use a subset of it: `<username>-<tweetId>` should be reasonable enough, and suitable for both sorting and naming.

We can assume that this method will not be called if the field is empty, and will assume that the URL is a valid twitter URL:

```
public function getName( SPIValue $value )
{
   return preg_replace(
       '#^https?://twitter\.com/([^/]+)/status/([0-9]+)$#',
       '$1-$2',
       (string)$value->url );
}


protected function getSortInfo( CoreValue $value )
{
   return $this->getName( $value );
}
```

In `getName()`, we run a regular expression replace on the URL to extract the part we're interested in.

This name is a perfect match for `getSortInfo()`, as it allows us to sort on the tweet's author and on the tweet's ID.

FieldType serialization methods: `fromHash()` and `toHash()`

Both methods, defined in the FieldType interface, are core to the REST API. They are used to export values to serializable hashes.

In our case, it is quite easy:

- `toHash()` will build a hash with every property from `Tweet\Value`;
- `fromHash()` will instantiate a `Tweet\Value` with the hash it receives.

```
public function fromHash( $hash )
{
   if ( $hash === null )
   {
       return $this->getEmptyValue();
   }
   return new Value( $hash );
}

public function toHash( SPIValue $value )
{
   if ( $this->isEmptyValue( $value ) )
   {
       return null;
   }
   return array(
       'url' => $value->url
   );
}
```

Persistence methods: `fromPersistenceValue` and `toPersistenceValue`

Storage of fieldType data is done through the persistence layer (SPI).

FieldTypes use their own Value objects to expose their contents using their own domain language. However, to store those objects, the Type needs to map this custom object to a structure understood by the persistence layer: `PersistenceValue`. This simple value object has three properties:

- `data` standard data, stored using the storage engine's native features
- `externalData` external data, stored using a custom storage handler
- `sortKey` sort value used for sorting

The role of those mapping methods is to convert a `Value` of the FieldType into a `PersistenceValue`, and the other way around.

> **"About external storage"**
> Whatever is stored in {{externalData}} requires an external storage handler to be written. Read more about external storage on Field Type API and best practices.
>
> External storage is beyond the scope of this tutorial, but many examples can be found in existing FieldTypes.

We will follow a simple implementation here: the `Tweet\Value` object will be serialized as an array to the `code` property using `fromHash()` and `toHash()`:

Tweet\Type

```
/**
 * @param \EzSystems\TweetFieldTypeBundle\eZ\Publish\FieldType\Tweet\Value $value
 * @return \eZ\Publish\SPI\Persistence\Content\FieldValue
 */
public function toPersistenceValue( SPIValue $value )
{
    if ( $value === null )
    {
        return new PersistenceValue(
            array(
                "data" => null,
                "externalData" => null,
                "sortKey" => null,
            )
        );
    }
    return new PersistenceValue(
        array(
            "data" => $this->toHash( $value ),
            "sortKey" => $this->getSortInfo( $value ),
        )
    );
}
/**
 * @param \eZ\Publish\SPI\Persistence\Content\FieldValue $fieldValue
 * @return \EzSystems\TweetFieldTypeBundle\eZ\Publish\FieldType\Tweet\Value
 */
public function fromPersistenceValue( PersistenceValue $fieldValue )
{
    if ( $fieldValue->data === null )
    {
        return $this->getEmptyValue();
    }
    return new Value( $fieldValue->data );
}
```

Fetching data from the twitter API

As explained in the tutorial's introduction, we will enrich our tweet's URL with the embed version, fetched using the twitter API. To do so, we will, when `toPersistenceValue()` is called, fill in the value's contents property from this method, before creating the `PersistenceValue` object.

First, we need a twitter client in `Tweet\Type`. For convenience, we provide one in this tutorial's bundle:

- The `Twitter\TwitterClient` class:
- The `Twitter\TwitterClientInterface` interface
- An `ezsystems.tweetbundle.twitter.client` service that uses the class above.

The interface has one method: `getEmbed( $statusUrl )` , that, given a tweet's URL, returns the embed code as a string. The implementation is very simple, for the sake of simplicity, but gets the job done. Ideally, it should at the very least handle errors, but it is not necessary here.

Injecting the twitter client into `Tweet\Type`

Our FieldType doesn't have a constructor yet. We will create one, with an instance of `Twitter\TwitterClientInterface` as the argument, and store it in a new protected property:

eZ/Publish/FieldType/Tweet/Type.php:

```
use EzSystems\TweetFieldTypeBundle\Twitter\TwitterClientInterface;

class Type extends FieldType
{
    /** @var TwitterClientInterface */
    protected $twitterClient;

    public function __construct( TwitterClientInterface $twitterClient )
    {
        $this->twitterClient = $twitterClient;
    }
}
```

Completing the value using the twitter client

As described above, before creating the `PersistenceValue` object in `toPersistenceValue`, we will fetch the tweet's embed contents using the client, and assign it to `Tweet\Value::$data`:

eZ/Publish/FieldType/Tweet/Type.php

```
 public function toPersistenceValue( SPIValue $value )
{
    // if ( $value === null )
    // {...}


    if ( $value->contents === null )
    {
        $value->contents = $this->twitterClient->getEmbed( $value->url );
    }
    return new PersistenceValue(
    // array(...)
}
```

And that's it ! When the persistence layer stores content from our type, the value will be completed with what the twitter API returns.

Previous: Implementing the Tweet\Value class

Next: Registering the FieldType as a service