# Field Type API and best practices

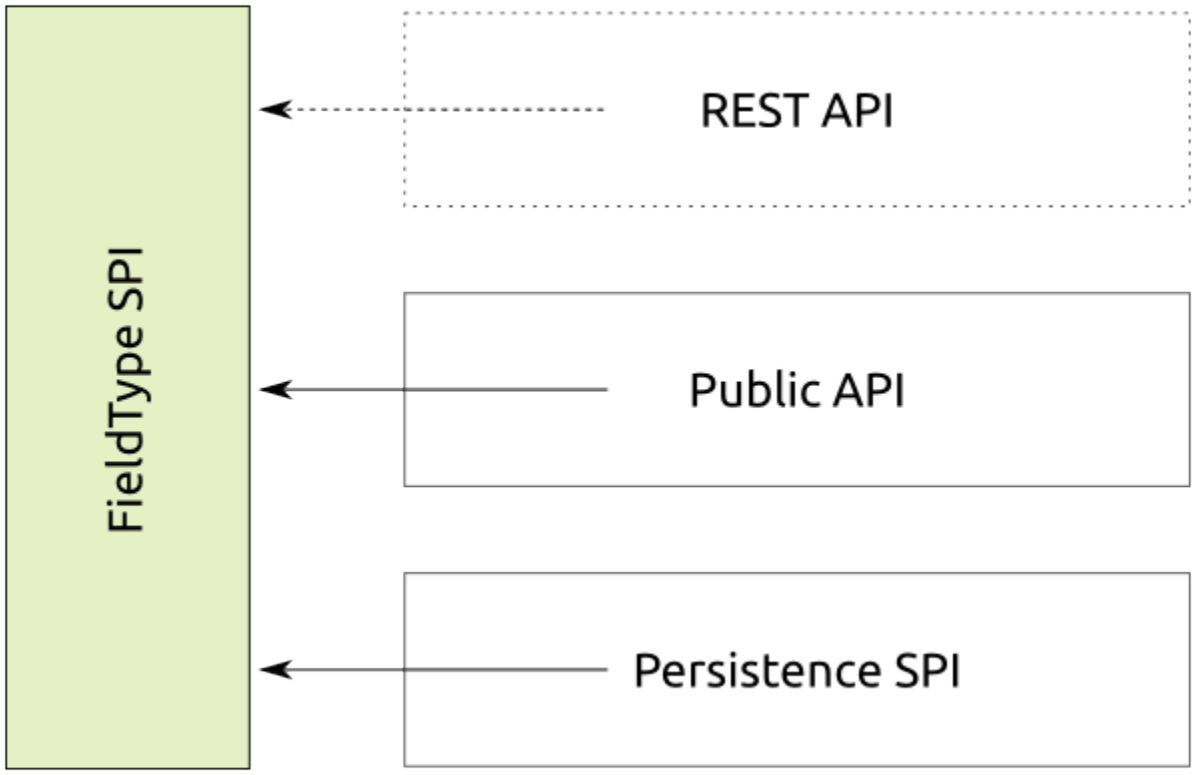


## Field Type API & best practices

The eZ Publish CMS can support arbitrary data to be stored in the fields of a content object. In order to support custom data, besides the standard data types, a developer needs to create a custom **FieldType**.

The implementation of a custom FieldType is done based on the FieldType SPI and its interfaces. These can be found under `eZ\Publish\SPI\FieldType`.

In order to provide custom functionality for a FieldType, the SPI interacts with multiple layers of the eZ Publish architecture, as shown in the following diagram:



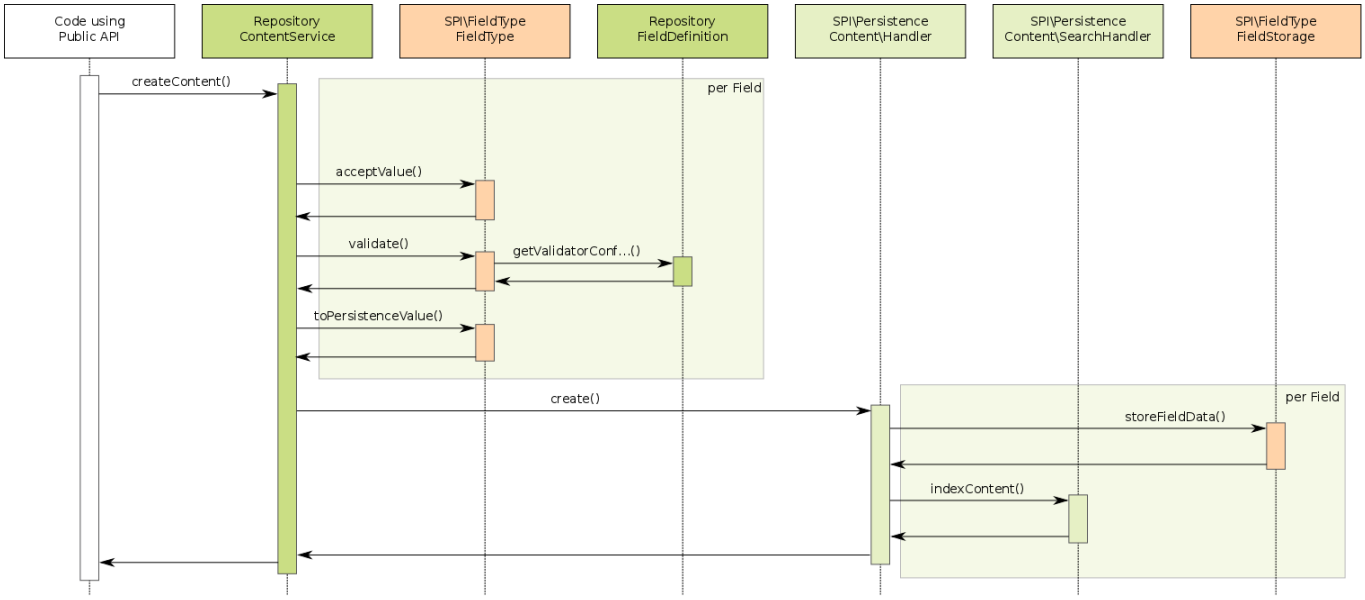
On the top layer, the FieldType needs to provide conversion from and to a simple PHP hash value to support the REST API. The generated hash value may only consist of scalar values and hashes. It must not contain objects or arrays with numerical indexes that aren't sequential and/or don't start with zero.

Below that, the FieldType must support the Public API implementation (aka Business Layer), regarding:

- Settings definition for FieldDefinitions
- Value creation and validation
- Communication with the Persistence SPI

On the bottom level, a FieldType can additionally hook into the Persistence SPI, in order to store data from a FieldValue in an external service. Note that all non-standard eZ Publish database tables (e.g. `ezurl`) are also considered "external storage" from now on.

The following sequence diagram visualizes the process of creating a new `Content` across all layers, especially focused on the interaction with a `FieldType`.




In the next lines/pages, this document explains how to implement a custom FieldType based on the SPI and what is expected from it. As a code example, please refer to the Url FieldType, which has been implemented as a reference.

# Public API interaction

The interaction with the Public API is done through the interface `eZ\Publish\SPI\FieldType\FieldType`. A custom FieldType must provide an implementation of this interface. In addition, it is considered best practice to provide a value object class for storing the custom field value provided by the FieldType.

## FieldDefinition handling

In order to make use of a custom FieldType, the user must apply it in a `eZ\Publish\API\Repository\Values\ContentType\FieldDefinition` of a custom ContentType. The user may in addition provide settings for the FieldType and a validator configuration. Since the Public API cannot know anything about these, their handling is delegated to the FieldType itself through the following methods:

`getFieldTypeIdentifier()`

> Returns a unique identifier for the custom FieldType, which is used to assign the type to a FieldDefinition. By convention for the returned type identifier string should be prefixed by a unique shortcut for the vendor (e.g. `ez` for eZ Systems).

`getSettingsSchema()`

> Using this method, the Public API retrieves a schema for the field type settings. A typical setting would e.g. be a default value. The settings structure defined by this schema is stored in the `FieldDefinition`. Since it is not possible to define a generic format for such a schema, the FieldType is free to return any serializable data structure from this method.

`getValidatorConfigurationSchema()`

In addition to normal settings, the FieldType should provide a schema settings for it's validation process. The schema describes, what kind of validation can be performed by the FieldType and which settings the user can specify to these validation methods. For example, the `ezstring` type can validate minimum and maximum length of the string. It therefore provides a schema to indicate to the user that he might specify the corresponding restrictions, when creating a `FieldDefinition` with this type. Again, the schema does not underly any regulations, except for that it must be serializable.

`validateFieldSettings()`

Before the Public API stores settings for the `FieldType` in a `FieldDefinition`, the type is asked to validate the settings (which were provided by the user). As a result, the `FieldType` must return if the given settings comply to the schema defined by `getSettingsSchema()`. `validateValidatorConfiguration()` Analog to `validateFieldSettings()`, this method verifies that the given validator configuration complies to the schema provided by `getValidatorConfigurationSchema()`.

It is important to note that while the schema definitions of the `FieldType` maybe both be of arbitrary, serializable format, it is highly recommended to use a simple hash structure. It is highly recommended to follow the Best practices in order to create future proof schemas.

**Note:** Since it is not possible to enforce a schema format, the code using a specific `FieldType` must basically know all `FieldTypes` it deals with.

This will also apply to all user interfaces and the REST API, which therefore must provide extension points to register handling code for custom `FieldType`. These extensions are not defined, yet.

## Value handling

A field type needs to deal with the custom value format provided by it. In order for the public API to work properly, it delegates working with such custom field values to the corresponding FieldType. The `SPI\FieldType\FieldType` interface therefore provides the following methods:

`acceptValue()`

This method is responsible for accepting and converting user input for the field. It checks the input structure it accepts and might build and return a different structure holding the data. An example would be, that the user just provides an HTTP link as a string, which is converted to the value object of the Url FieldType. Unlike the `FieldType\Value` constructor, it is perfectly acceptable to make this method aware of multiple input types (object or primitive, for instance).

Note: The method must assert structural consistency of the value, but must not validate plausibility of the value.

`getEmptyValue()`

Through settings, the `FieldType` can specify, that the user may define a default value for the `Field` of the type. If no such default is provided by the user, the `FieldType` itself is asked for an "empty value" as the final fallback. The value chain for a specific field is therefore like this, when a `Field` of the `FieldType` is filled out:

a. Is a value provided by the filling user?
b. Is a default provided by the `FieldDefinition`?
c. Take the empty value provided by the `FieldType`

`validate()`

In contrast to `acceptValue()` this method validates the plausibility of the given value, based on the `FieldType` settings and validator configuration, stored in the corresponding `FieldDefinition`.

# Storage conversion

As said above, the value format of a `FieldType` is free form. However, in order to make eZ Publish store the value in it's database, it must comply to certain rules at storage time. To not restrict the value itself, a `FieldValue` must be converted to the storage specific format used by the Persistence SPI: `eZ\Publish\SPI\Persistence\Content\FieldValue`. After restoring a Field of `FieldType`, the conversion must be undone. The following methods of the `FieldType` are responsible for that:

`toPersistenceValue()`

This method receives the value of a `Field` of `FieldType` and must return an SPI `FieldValue`, which can be stored.

`fromPersistenceValue()`

As the counterpart, this method receives an SPI `FieldValue` and must reconstruct the original value of the `Field` from it.

The SPI FieldValue struct has several properties, which might be used by the FieldType as follows:

`$data`

The data to be stored in the eZ Publish database. This may either be a scalar value, a hash map or a simple, serializable object.

`$externalData`

The arbitrary data stored in this field will not be touched by any of the eZ Publish components directly, but will be hold available for Storing external data.

`$sortKey`

An value which can be used to sort `Content` by the field.
*Note: TBD: Where will you register the Indexable implementations?*

# Searching

Fields, or a custom field type, might contain or maintain data which is relevant for user searches. To make the search engine aware of the data in your field type you need to implement an additional interface and register the implementation.

If your field type does not maintain any data, which should be available to search engines, feel free to just ignore this section.

The `eZ\Publish\SPI\FieldType\Indexable` defines two methods, which are required to be implemented, if the field type provides data relevant to search engines. The interface defines two methods for this:

`getIndexData( Field $field )`

This method is supposed to return the actual index data for the provided `eZ\Publish\SPI\Persistence\Content\Field`. The index data consists of an array of `eZ\Publish\SPI\Persistence\Content\Search\Field` instances. They are described below in further detail.

`getIndexDefinition()`

To be able to query data properly an indexable field type also is required to return search specification. You must return a hash map of `eZ\Publish\SPI\Persistence\Content\Search\FieldType` instances from this method, which could look like:

```
array(
    'url'  => new Search\FieldType\StringField(),
    'text' => new Search\FieldType\StringField(),
)
```

This example from the `Url` field type shows that the field type will always return two indexable values, both strings. They have the names `url` and `text` respectively.

## Search Field Values

The search field values, returned by the `getIndexData` method are simple value objects consisting of the following properties:

`$name`

The name of the field

`$value`

The value of the field

`$type`

An `eZ\Publish\SPI\Persistence\Content\Search\FieldType` instance, describing the type information of the field.

## Search Field Types

There are bunch of available search field types, which are automagically handled by our Search backend configuration. When using those there is no requirement to adapt , for example, the Solr configuration in any way. You can always use custom field types, though, but these might require re-configuration of the search backend. For Solr this would mean adapting the schema.xml.

The default available search field types are:

`StringField.php`

Standard string values. Will also be queries by full text searches.

`TextField.php`

Standard text values. Will be queried by full text searches. Configured text normalizations in the search backend apply.

`BooleanField.php`

Boolean values.

`DateField.php`

Date field. Can be used for date range queries.

`FloatField.php`

Field for floating point numbers.

`IntegerField.php`

Field for integer numbers.

`PriceField.php`

Field for price values. Currency conversion might be applied by the search backends. Might require careful configuration.

`IdentifierField.php`

Field used for IDs. Basically acts like the string field, but will not be queried by fulltext searches

`CustomField.php`

Custom field, for custom search data types. Will probably require additional configuration in the search backend.

## Configuring Solr

As mentioned before, if you are using the standard type definitions **there is no need to configure the search backend in any way**. Everything will work fine. The field definitions are handled using `dynamicField` definitions in Solr, for example.

If you want to configure the handling of your field, you can always add a special field definition the Solr `schema.xml`. The field type names, which are used by the Solr search backend look like this for fields: `<content_type_identifier>/<field_identifier>/<search_field_name>_<type>`. You can, of course define custom `dynamicField` definitions to match, for example, on your custom `_<type>` definition.

You could also define a custom field definition dedicatedly for certain fields, like for the name field in an article:

```
<field name="article/name/value_s" type="string" indexed="true" stored="true"
required="false"/>
```

If you want to learn more about the Solr implementation and detailed information about configuring it, check out the Solr Search Service Implementation Notes.

## Storing external data

A `FieldType` may store arbitrary data in external data sources and is in fact encouraged to do so. External storages can be e.g. a web service, a file in the file system, another database or even the eZ Publish database itself (in form of a non-standard table). In order to perform this task, the `FieldType` will interact with the Persistence SPI, which can be found in `eZ\Publish\SPI\Persistence`, through the `eZ\Publish\SPI\FieldType\FieldStorage` interface.

Whenever the internal storage of a Content that includes a Field of the `FieldType` is accessed, one of the following methods is called to also access the external data:

`hasFieldData()`

Returns if the `FieldType` stores extrnal data at all.

`storeFieldData()`

Called right before a `Field` of `FieldType` is stored. The method should perform the storing of `$externalData`. The method must return `true`, if the call manipulated **internal data** of the given `Field`, so that it is updated in the internal database.

`getFieldData()`

Is called after a `Field` has been restored from the database in order to restore `$externalData`.

`deleteFieldData()`

Must delete external data for the given `Field`, if exists.

`getIndexData()`

See search service

Each of the above methods receive a $context array, which contains information on the underlying storage and the environment. This context can be used to store data in the eZ Publish data storage, but outside of the normal structures (e.g. a custom table in an SQL database). Note that the

FieldType must take care on it's own for being compliant to different data sources and that 3rd parties can extend the data source support easily. For more information about this, take a look at the Best practices section.

# Legacy Storage conversion

The `FieldType` system is designed for future storage back ends of eZ Publish. However, the old database schema (*Legacy Storage*) must still be supported. Since this database cannot store arbitrary value information as provided by a `FieldType`, another conversion step must take place if the Legacy Storage is used.

The conversion takes place through the interface `eZ\Publish\Core\Persistence\Legacy\Content\FieldValue\Converter`, which you must provide an implementation of with your `FieldType`. The following methods are contained in the interface:

`toStorageValue()`

: Converts a Persistence `Value` into a legacy storage specific value.

`fromStorageValue()`

: Converts the other way around.

`toStorageFieldDefinition()`

: Converts a Persistence `FieldDefinition` to a storage specific one.

`fromStorageFieldDefinition`

: Converts the other way around.

`getIndexColumn()`

: Returns the storage column which is used for indexing.

## Registering a converter

The registration of a `Converter` currently works through the `$config` parameter of `eZ\Publish\Core\Persistence\Legacy\Handler`. See the class documentation for further details.

For global service container integration, see Register FieldType.

# REST API interaction

When REST API is used, conversion needs to be done for `FieldType` values, settings and validator configurations. These are converted to and from a simple hash format that can be encoded in REST payload (typically XML or JSON). As conversion needs to be done both when transmitting and receiving data through REST, `FieldType` implements following pairs of methods:

`toHash()`

: Converts FieldType Value into a plain hash format.

`fromHash()`

: Converts the other way around.

`fieldSettingsToHash()`

: Converts FieldType settings to a simple hash format.

`fieldSettingsFromHash()`

: Converts the other way around.

`validatorConfigurationToHash()`

: Converts FieldType validator configuration to a simple hash format.

`validatorConfigurationFromHash()`

: Converts the other way around.

## Extension points

Some `FieldTypes` will require additional processing, for example a `FieldType` storing a binary file, or one having more complex settings or

validator configuration. For this purpose specific implementations of an abstract class `eZ\Publish\Core\REST\Common\FieldTypeProcessor` are used. This class provides following methods:

`preProcessValueHash()`

Performs manipulations on a received value hash, so that it conforms to the format expected by the `fromHash()` method described above.

`postProcessValueHash()`

Performs manipulations on a outgoing value hash, previously generated by the `toHash()` method described above.

`preProcessFieldSettingsHash()`

Performs manipulations on a received settings hash, so that it conforms to the format expected by the `fieldSettingsFromHash()` method described above.

`postProcessFieldSettingsHash()`

Performs manipulations on a outgoing settings hash, previously generated by the `fieldSettingsToHash()` method described above.

`preProcessValidatorConfigurationHash()`

Performs manipulations on a received validator configuration hash, so that it conforms to the format expected by the `validatorConfigurationFromHash()` method described above.

`postProcessValidatorConfigurationHash()`

Performs manipulations on a outgoing validator configuration hash, previously generated by the `validatorConfigurationToHash()` method described above.

Base implementations of these methods simply return the given hash, so you can implement only the methods your `FieldType` requires. Some `FieldTypes` coming with the eZ Publish installation already implement processors and you are encouraged to take a look at them.

For details on registering a `FieldType` processor, see Register FieldType page.

# Best practices

In this chapter, best practices for implementing a custom *—FieldType* are collected. We highly encourage following these practices to be future proof.

## Gateway based Storage

In order to allow the usage of a `FieldType` that uses external data with different data storages, it is recommended to implement a gateway infrastructure and a registry for the gateways. In order to ease this action, the Core implementation of `FieldType` s provides corresponding interfaces and base classes. These can also be used for custom field types.

The interface `eZ\Publish\Core\FieldType\StorageGateway` is implemented by gateways, in order to be handled correctly by the registry. It has only a single method:

`setConnection()`

The registry mechanism uses this method to set the SPI storage connection (e.g. the database connection to the Legacy Storage database) into the gateway, which might be used to store external data. The connection is retrieved from the `$context` array automatically by the registry.

Note that the Gateway implementation itself must take care about validating that it received a usable connection. If it did not, it should throw a `RuntimeException`.

The registry mechanism is realized as a base class for `FieldStorage` implementations: `eZ\Publish\Core\FieldType\GatewayBasedStorage`. For managing `StorageGateway` s, the following methods are already implemented in the base class:

`addGateway()`

Allows the registration of additional `StorageGateways` from the outside. Furthermore, a hash map of `StorageGateways` can be given to the constructor for basic initialization. This array should orginate from the Dependency Injection mechanism.

`getGateway()`

This protected method is used by the implementation to retrieve the correct `StorageGateway` for the current context.

As a reference for the usage of these infrastructure, the Keyword, Url and User types can be examined.

## Settings schema

It is recommended to use a simple hash map format for the settings schema retured by eZ\Publish\SPI\FieldType\FieldType::getSettingsSchema() , which follows these rules:

- The key of the hash map identifies a setting (e.g. `default`)
- Its value is a hash map (2nd level) describing the setting using
    - `type` to identify the setting type (e.g. `int` or `string`)
    - `default` containing the default setting value

An example schema could look like this:

```
array(
    'backupData' => array(
        'type' => 'bool',
        'default' => false
    ),
    'defaultValue' => array(
        'type' => 'string',
        'default' => 'Sindelfingen'
    )
);
```

### Validator schema

The schema for validator configuration should have a similar format than the settings schema has, except it has an additional level, to group settings for a certain validation mechanism:

- The key on the 1st level is a string, identifying a validator
- Assigned to that is a hash map (2nd level) of settings
- This hash map has a string key for each setting of the validator
- It is assigned to a 3rd level hashmap, the setting description
- This hash map should have the same format as for normal settings

For example, for the `ezstring` type, the validator schema could be:

```
array(
    'stringLength' => array(
        'minStringLength' => array(
            'type'    => 'int',
            'default' => 0,
        ),
        'maxStringLength' => array(
            'type'    => 'int'
            'default' => null,
        )
    ),
);
```

## Registering a FieldType

To register a FieldType, see Register FieldType.

To be integrated in unit and integration tests, FieldTypes need to be registered through the `service.ini` in `eZ/Publish/Core/settings`.

## Templating

A FieldType always need a piece of template to be correctly displayed. See FieldType template.

## Testing

`FieldType` s should be integration tested on 1 different levels:

1. Their integration with the Persistence SPI
2. Their integration with the Public API

For both test environments, infrastructure is already in place, so that you can easily implement the required tests for your custom `FieldType`

## Persistence SPI

This type of integration test ensures, that a FieldType stores its data properly on basis of different Persistence SPI implementations.

> **Note:** By now, only the Legacy Storage implementation exists.

The integration tests with the Persistence SPI can be found in `eZ\Publish\SPI\Tests\FieldType`. In order to implement a test for your custom `FieldType`, you need to extend the common base class `eZ\Publish\SPI\Tests\FieldType\BaseIntegrationTest` and implement it's abstract methods. As a reference the `KeywordIntegrationTest`, `UrlIntegrationTest` and `UserIntegrationTest` can deal.

Running the test is fairly simple: Just specify the global `phpunit.xml` for PHPUnit configuration and make it execute a single test or a directory of tests, for example:

```
$ phpunit -c phpunit.xml eZ/Publish/SPI/Tests/FieldType
```

in order to run all `FieldType` tests.

## Public API

On a second level, the interaction between an implementation of the Public API (aka the Business Layer) and the FieldType is tested. Again, there is a common base class as the infrastructural basis for such tests, which resides in `eZ\Publish\API\Repository\Tests\FieldType\BaseIntegrationTest`.

> Note that the In-Memory stubs for the Public API integration test suite, do not perform actual FieldType calls, but mainly emulate the behavior of a FieldType for simplicity reasons.
>
> If your FieldType needs to convert data between `storeFieldData()` and `getFieldData()`, you need to implement a `eZ\Publish\API\Repository\Tests\Stubs\PseudoExternalStorage` in addition, which performs this task. Running the tests against the Business Layer implementation of the Public API is not affected by this.