2.6. Document Validation

By design, MongoEngine strictly validates the documents right before they are inserted in MongoDB and makes sure they are consistent with the fields defined in your models.

MongoEngine makes the assumption that the documents that exists in the DB are compliant with the schema. This means that Mongoengine will not validate a document when an object is loaded from the DB into an instance of your model but this operation may fail under some circumstances (e.g. if there is a field in the document fetched from the database that is not defined in your model).

2.6.1. Built-in validation

Mongoengine provides different fields that encapsulate the corresponding validation out of the box. Validation runs when calling *.validate()* or *.save()*

```
from mongoengine import Document, EmailField, IntField

class User(Document):
    email = EmailField()
    age = IntField(min_value=0, max_value=99)

user = User(email='invalid@', age=24)
user.validate()     # raises ValidationError (Invalid email address: ['email'])
user.save()         # raises ValidationError (Invalid email address: ['email'])

user2 = User(email='john.doe@garbage.com', age=1000)
user2.save()        # raises ValidationError (Integer value is too large: ['age'])
```

2.6.2. Custom validation

The following feature can be used to customize the validation:

Field *validation* parameter

```
def not_john_doe(name):
    if name == 'John Doe':
        raise ValidationError("John Doe is not a valid name")

class Person(Document):
    full_name = StringField(validation=not_john_doe)

Person(full_name='Billy Doe').save()
Person(full_name='John Doe').save()  # raises ValidationError (John Doe is not a valid
name)
```

Document *clean* method

This method is called as part of `save()` and should be used to provide custom model validation and/or to modify some of the field values prior to validation. For instance, you could use it to automatically provide a value for a field, or to do validation that requires access to more than a single field.

```
class Essay(Document):
    status = StringField(choices=('Published', 'Draft'), required=True)
    pub_date = DateTimeField()

    def clean(self):
        # Validate that only published essays have a `pub_date`
        if self.status == 'Draft' and self.pub_date is not None:
            raise ValidationError('Draft entries should not have a publication date.')
        # Set the pub_date for published items if not set.
        if self.status == 'Published' and self.pub_date is None:
            self.pub_date = datetime.now()
```

Note

Cleaning is only called if validation is turned on and when calling `save()`.

Adding custom Field classes

We recommend as much as possible to use fields provided by MongoEngine. However, it is also possible to subclass a Field and encapsulate some validation by overriding the *validate* method

```
class AgeField(IntField):

    def validate(self, value):
        super(AgeField, self).validate(value)     # let IntField.validate run first
        if value == 60:
            self.error('60 is not allowed')

class Person(Document):
    age = AgeField(min_value=0, max_value=99)

Person(age=20).save()   # passes
Person(age=1000).save() # raises ValidationError (Integer value is too large: ['age'])
Person(age=60).save()   # raises ValidationError (Person:None) (60 is not allowed:
['age'])
```

Note

When overriding *validate*, use *self.error("your-custom-error")* instead of raising ValidationError explicitly, it will provide a better context with the error message

2.6.3. Skipping validation

Although discouraged as it allows to violate fields constraints, if for some reason you need to disable the validation and cleaning of a document when you call `save()`, you can use *.save(validate=False)*.

```
class Person(Document):
    age = IntField(max_value=100)

Person(age=1000).save()    # raises ValidationError (Integer value is too large)

Person(age=1000).save(validate=False)
person = Person.objects.first()
assert person.age == 1000
```