PLURALSIGHT

Custom WTForms Field tutorial for:

CREATING AND PROCESSING WEB FORMS WITH FLASK

Mateo Prigl

Custom Select Field

Let's create a custom select field by extending the functionality from the WTForms package.
This field should populate the select field choices by taking them straight from the database.

So this is the end product. For example, someone created an application for students. This applications needs to have a select field for choosing students. They want to use our custom field to take the students from the database.
The select field should take this from the table `students`. The value of the options inside of the select field will be the students `id` column. Options shown to the user should be taken from the `full_name` column.

This is an example of the needed select field:

```
<select id="students">
        <option value="1">Maria Lopez</option>
        <option value="2">Mark Stark</option>
        <option value="3">Jason Junior</option>
</select>
```

The custom select field needs to know which table it should look at and which columns should be used for populating the options.

We will extend the WTForms `SelectField` class.

This is the `__init__` method for the `SelectField`.

```
def __init__(
        self,
        label=None,
        validators=None,
        coerce=text_type,
        choices=None,
        validate_choice=True,
        **kwargs
    ):
        super(SelectField, self).__init__(label, validators, **kwargs)
        self.coerce = coerce
        self.choices = list(choices) if choices is not None else None
        self.validate_choice = validate_choice
```

Our `__init__` method will also take these parameters and pass them to the SelectField.

```
class CustomSelectField(SelectField):
        def __init__(self, label=None, validators=None, coerce=int, choices=None, **kwargs):
        super(CustomSelectField, self).__init__(label, validators, coerce, choices, **kwargs)
```

We will add new parameters to the initialization. `table` will hold the name of the SQlite table which needs to be queried. `columns` is a list. It should have two elements, each one representing the columns we need to get from the database. I will also add `allow_blank`. If this parameter is true, we will add a placeholder option as the first option of the select field. This is something we did in the course to by adding `---`.

```
class CustomSelectField(SelectField):
        def __init__(self, label=None, validators=None, coerce=int, choices=None, table=None, columns=[], allow_blank=False, **kwargs):
                super(CustomSelectField, self).__init__(label, validators, coerce, choices, **kwargs)
                self.allow_blank = allow_blank
                if not table:
                        raise AttributeError("CustomSelectField does not work without the table parameter.")
                if not len(columns):
                        raise AttributeError("CustomSelectField does not work without the list of columns.")
                self.table  = table
                self.columns = columns
```

If the table or the columns parameters are not present, raise an attribute error. We need them for the custom select field.

Now we can create a custom method for getting the information from the database. I will call it `get_rows`.

```
def get_rows(self):
    c = get_db().cursor()
    try:
        c.execute("SELECT {}, {} FROM {}".format(self.columns[0], self.columns[1], self.table))
    except:
        raise AttributeError("Something went wrong.")
    rows = c.fetchall()
```

```
    if self.allow_blank:
        rows.insert(0, (0, "---"))
    return rows
```

This method will do a select query and fetch all of the choices from the database. You can see that we injected the table name and the name of the columns from the columns list. If the `allow_blank` option was set to true, we will append the placeholder option too. Finally return the rows.

Now we need to use this function in two important `SelectField` methods: `iter_choices` and `prevalidate`. `iter_choices` method passes the select field choices to the widget of the select field. We need to override this and pass our own choices from the database.

```
def iter_choices(self):
    rows = self.get_rows()
    for value, label in rows:
        yield (value, label, self.coerce(value) == self.data)
```

We also need to override the `pre_validate` method. We mentioned this method when we were learning about form validation. It will validate if the picked choice is one of the allowed choices and raise an error if it isn't. We want this method to look at our choices instead:

```
def pre_validate(self, form):
    rows = self.get_rows()
    for v, _ in rows:
        if self.data == v:
            break
    else:
        raise ValueError("The chosen option does not exist.")
```

Here is the final blueprint of our `CustomSelectField` class.

```
class CustomSelectField(SelectField):
    def __init__(self, label=None, validators=None, coerce=int, choices=None, table=None, columns=[], allow_blank=False, **kwargs):
        super(CustomSelectField, self).__init__(label, validators, coerce, choices, **kwargs)
        self.allow_blank = allow_blank
        if not table:
            raise AttributeError("CustomSelectField does not work without the table parameter.")
        if not len(columns):
            raise AttributeError("CustomSelectField does not work without the list of columns.")
        self.table  = table
        self.columns = columns

    def iter_choices(self):
        rows = self.get_rows()
        for value, label in rows:
            yield (value, label, self.coerce(value) == self.data)

    def pre_validate(self, form):
        rows = self.get_rows()
        for v, _ in rows:
            if self.data == v:
                break
        else:
            raise ValueError("The chosen option does not exist.")

    def get_rows(self):
        c = get_db().cursor()
        try:
            c.execute("SELECT {}, {} FROM {}".format(self.columns[0], self.columns[1], self.table))
        except:
            raise AttributeError("Something went wrong.")
        rows = c.fetchall()
        if self.allow_blank:
            rows.insert(0, (0, "---"))
        return rows
```

Now we can replace the `SelectField` for the `category` and `subcategory` fields. Add the appropriate `table` and `columns` parameters.

```
category = CustomSelectField(
                "Category",
                coerce=int,
                table="categories",
                columns=["id", "name"]
)

subcategory = CustomSelectField(
                "Subcategory",
                coerce=int,
                table="subcategories",
                columns=["id", "name"],
                validators=[
                            BelongsToOtherFieldOption(table="subcategories", belongs_to="category", message="choice does not belong to the chosen category.")
                ]
)
```

When you add this field to the `FilterForm` fields, you can also add the `allow_blank=True` parameter, since those fields need a placeholder.

The example with the students from the beginning would look something like this:

```
students = CustomSelectField("Students", coerce=int, table="students", columns=["id", "full_name"])
```

WTForms package is powerful but simple to extend. Anytime you need a special case field, you can just check out the source code and override the needed methods.

If you want to incorporate this as you code along with the course, you are free to do so. It will not break anything in the application. If you use this new field, you can delete the fetching of the categories and subcategories from the views.
Happy learning!
Mateo