UNIVERSITÀ DELLA CALABRIA
il Campus per eccellenza

Student Projects – Lessons Learned

Mario Alviano

Choices and drop-down menus

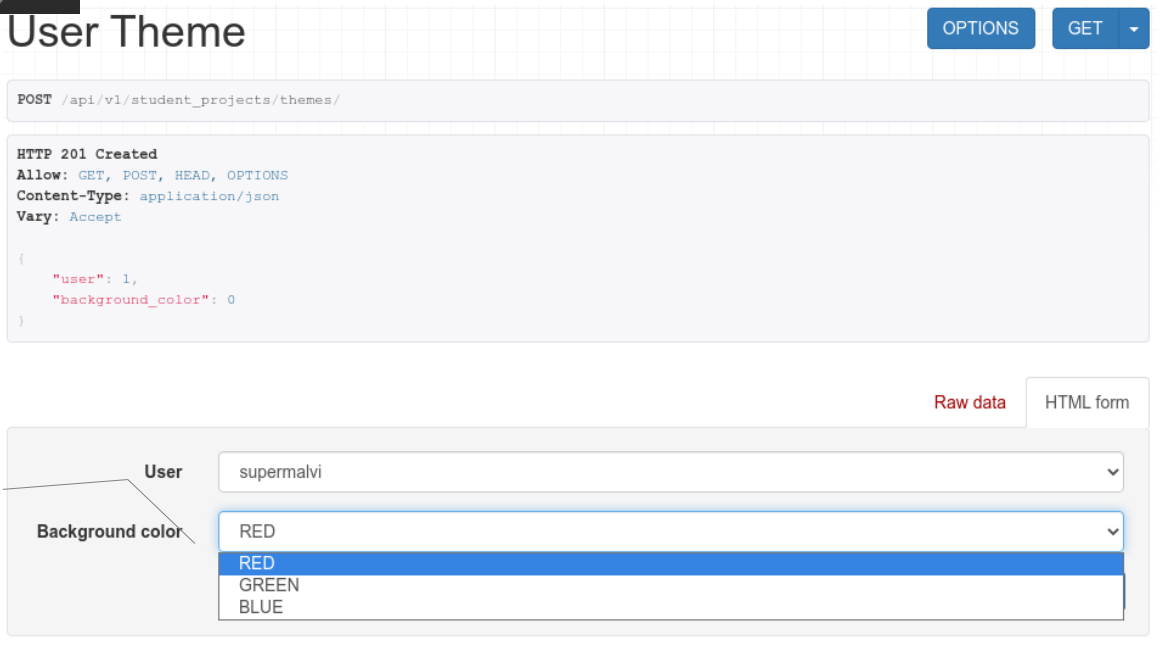
```python
def color_choices():
    choices = ['RED', 'GREEN', 'BLUE']
    return [(i, choices[i]) for i in range(len(choices))]


class UserTheme(models.Model):
    user = models.ForeignKey(get_user_model(), on_delete=models.CASCADE)
    background_color = models.IntegerField(choices=color_choices())
```

Use a list of pairs (key, value) to define enumerations

```python
class UserThemeSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserTheme
        fields = ('user', 'background_color')
```

```python
class UserThemeView(ListCreateAPIView):
    serializer_class = UserThemeSerializer
    queryset = UserTheme.objects.all()
```

Serialization of the field introduces a drop-down menu in the Browsable API and in the admin site



Computed (read-only) fields

Suggested way: declare a property
in the model and use it as any other field

```python
class UserTheme(models.Model):
    user = models.ForeignKey(get_user_model(), on_delete=models.CASCADE)
    background_color = models.IntegerField(choices=color_choices())

    @property
    def background_color_human_readable(self):
        return color_choices()[self.background_color][1]
```

```python
class UserThemeSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserTheme
        fields = (
            'user',
            'background_color',
            'username',
            'background_color_human_readable',    # directly access a property of the model (suggested)
            'background_color_human_readable_using_source',
        )

    username = serializers.SerializerMethodField()

    def get_username(self, obj):
        return obj.user.username

    background_color_human_readable_using_source = serializers.ReadOnlyField(source='background_color_human_readable')
```

Fields can be computed
by a serializer get_ method

Field or property
in the model
(source is available also
in other field types)

User Theme

OPTIONS GET

```
GET /api/v1/student_projects/themes/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "user": 1,
        "background_color": 0,
        "username": "supermalvi",
        "background_color_human_readable": "RED",
        "background_color_human_readable_using_source": "RED"
    }
]
```

Computed fields in the response

Raw data | HTML form

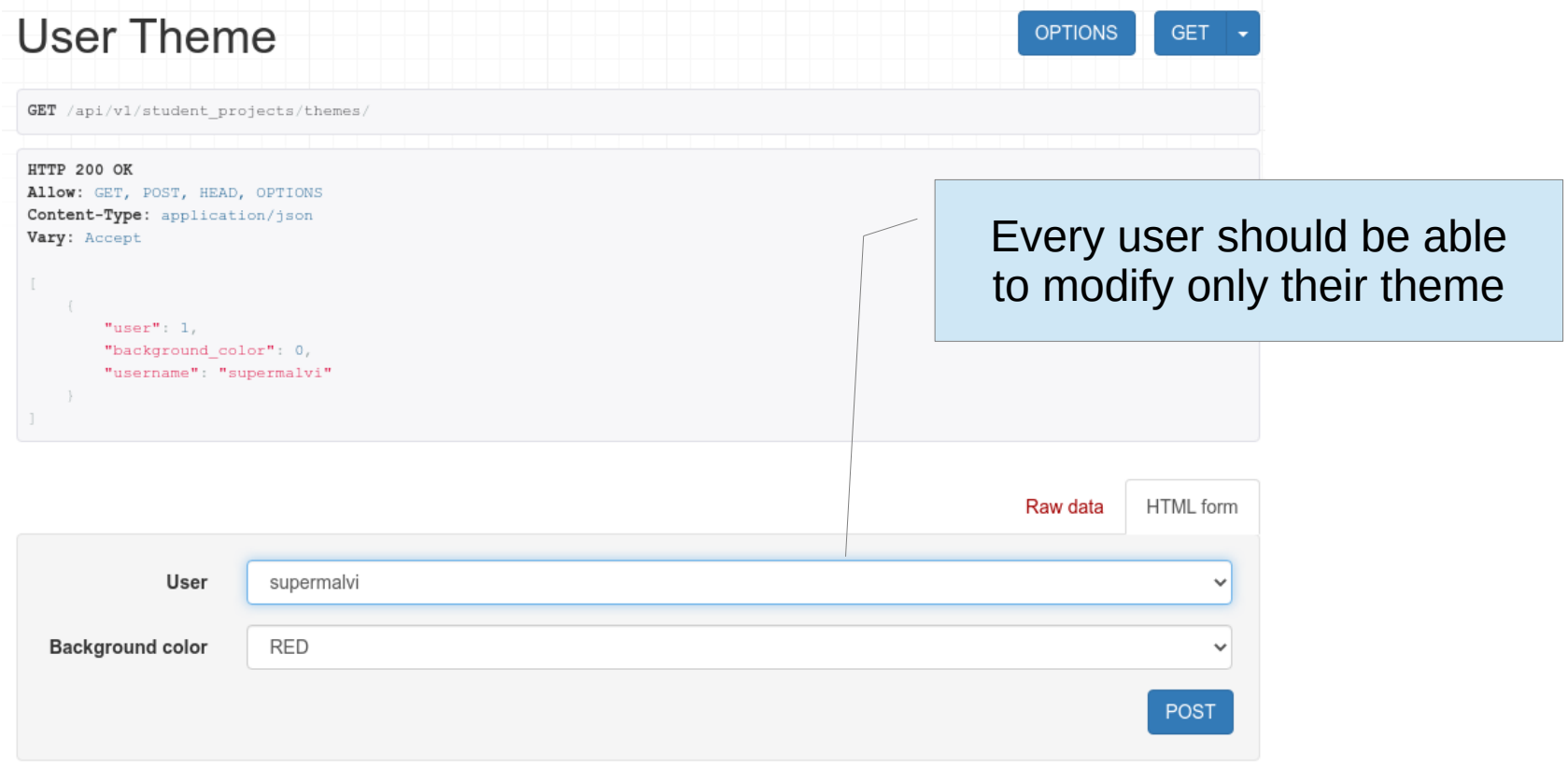
User supermalvi

Background color RED

POST

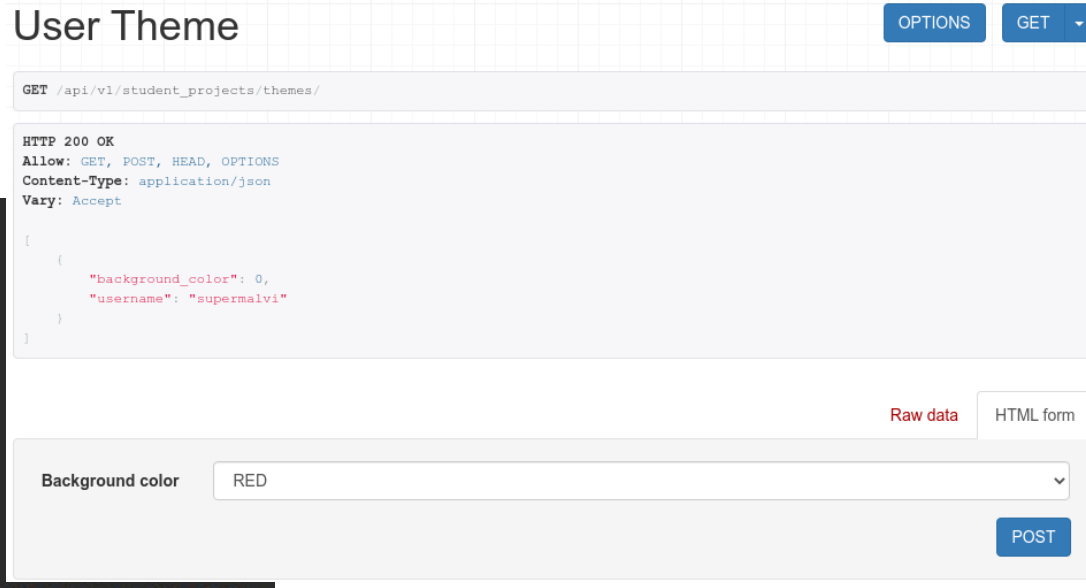
They are read-only,
so they cannot be part of the request

Implicit fields
(eg. get user from request)



Suggested way

```python
class UserThemeSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserTheme
        fields = (
            'user',
            'background_color',
            'username',
        )

    user = serializers.HiddenField(default=UserFromRequestFieldValue())
```




```python
@dataclass
class UserFromRequestFieldValue:
    value: int = field(init=False)

    def set_context(self, serializer_field):
        self.value = serializer_field.context.get('request').user

    def __call__(self):
        return self.value
```

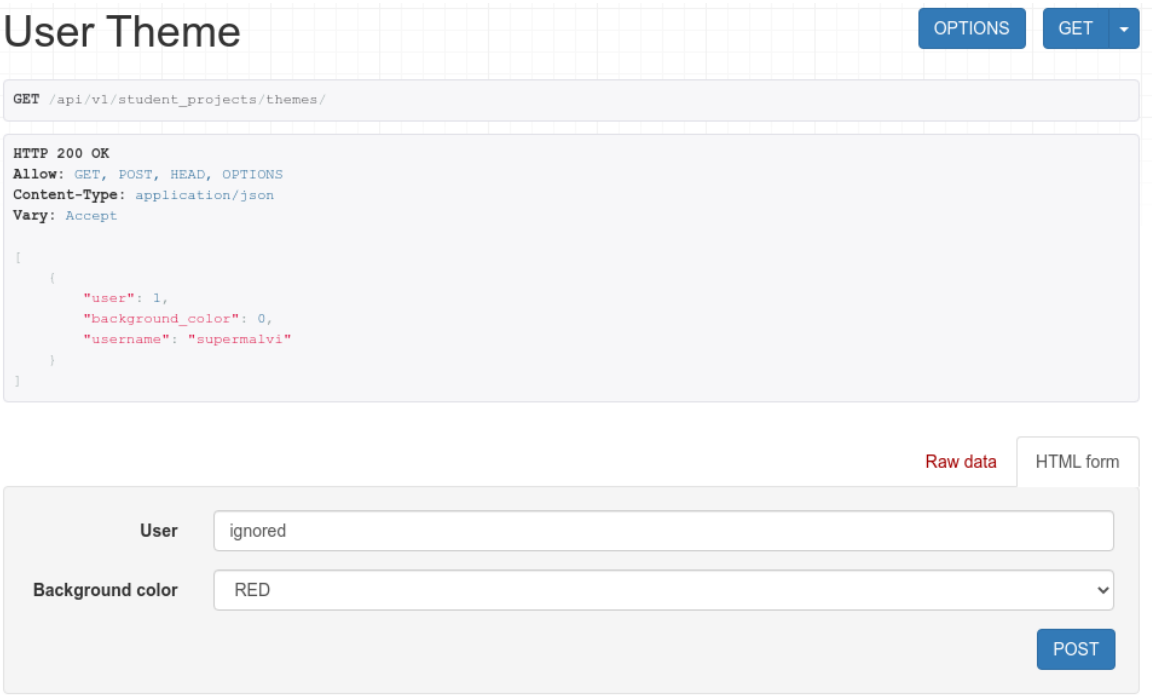
Use a hidden field
an give value by defining a dataclass

```python
class UserThemeSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserTheme
        fields = (
            'user',
            'background_color',
            'username',
        )

    user = UserFromRequestField(initial='ignored')
```

```python
class UserFromRequestField(serializers.Field):
    def to_representation(self, value):
        return self.context.get('request').user.id

    def to_internal_value(self, data):
        return self.context.get('request').user.id
```

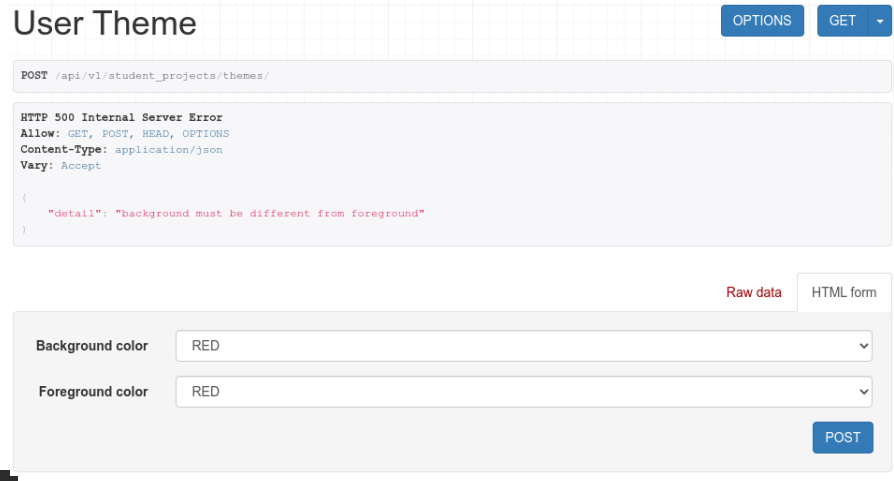
Define a custom field (in the Browsable API show that the provided value is ignored)

A bit hugly!



Validation involving multiple fields

```python
class UserThemeSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserTheme
        fields = ('user', 'background_color', 'foreground_color')

    user = serializers.HiddenField(default=UserFromRequestFieldValue())
```




```python
class UserTheme(models.Model):
    user = models.ForeignKey(get_user_model(), on_delete=models.CASCADE)
    background_color = models.IntegerField(choices=color_choices())
    foreground_color = models.IntegerField(choices=color_choices())

    def validate(self):
        if self.background_color == self.foreground_color:
            raise APIException('background must be different from foreground')

    def save(self, *args, **kwargs):
        self.validate()
        super(UserTheme, self).save(*args, **kwargs)
```

Add them in a method

Call that method
before saving the instance

Patch requests methods

```python
def mock_response(status_code, data):
    res = Mock()
    res.status_code = status_code
    res.json.return_value = [x.__dict__ for x in data]
    return res


@patch('requests.get', side_effect=[mock_response(200, [])])
@patch('builtins.input', side_effect=['0'])
@patch('builtins.print')
def test_app_main(mocked_print, mocked_input, mocked_requests_get):
    main('__main__')
    mocked_print.assert_any_call('*** Foo TUI ***')
    mocked_print.assert_any_call('0:\tExit')
    mocked_print.assert_any_call('Bye!')
    mocked_input.assert_called()
    mocked_requests_get.assert_called()
```

As we did for builtins.input, use side_effect to provide returned values of calls to requests.get and requests.post

Add everything you need to a mock object

If you want to mock a method of a mock object, add an attribute with the name of the method and specify the returned value by assigning return_value

```python
@patch('requests.get', side_effect=[mock_response(200, [Foo('red'), Foo('blue')])])
@patch('builtins.input', side_effect=['0'])
@patch('builtins.print')
def test_app_main_load(mocked_print, mocked_input, mocked_requests_get):
    main('__main__')
    assert list(filter(lambda x: '1 red' in str(x), mocked_print.mock_calls))
    assert list(filter(lambda x: '2 blue' in str(x), mocked_print.mock_calls))
    mocked_input.assert_called()
    mocked_requests_get.assert_called()
```

Return a list of objects and
check that the app is doing is job
by printing those objects as expected

Patch post requests by returing
the expected response
(actually, a mock, because we don't want
to really create a response object)

```
37  @patch('requests.post', side_effect=[mock_response(201, [Foo('green')])])
38  @patch('requests.get', side_effect=[
39      mock_response(200, [Foo('red'), Foo('blue')]),
40      mock_response(200, [Foo('red'), Foo('blue'), Foo('green')]),
41  ])
42  @patch('builtins.input', side_effect=['1', 'green', '0'])
43  @patch('builtins.print')
44  def test_app_main_add(mocked_print, mocked_input, mocked_requests_get, mocked_requests_post):
45      main('__main__')
46      assert list(filter(lambda x: '3 green' in str(x), mocked_print.mock_calls))
47      mocked_input.assert_called()
48      mocked_requests_get.assert_called()
49      mocked_requests_post.assert_called()
50      mocked_requests_post.assert_called_once_with(url=App.foo_endpoint(), data=Foo('green').__dict__)
```

Simulate creation of new object
to trigger requests.post

If there are two object when the app starts,
there must be three objects after adding one object

Check that the expected
requests are triggered

Questions