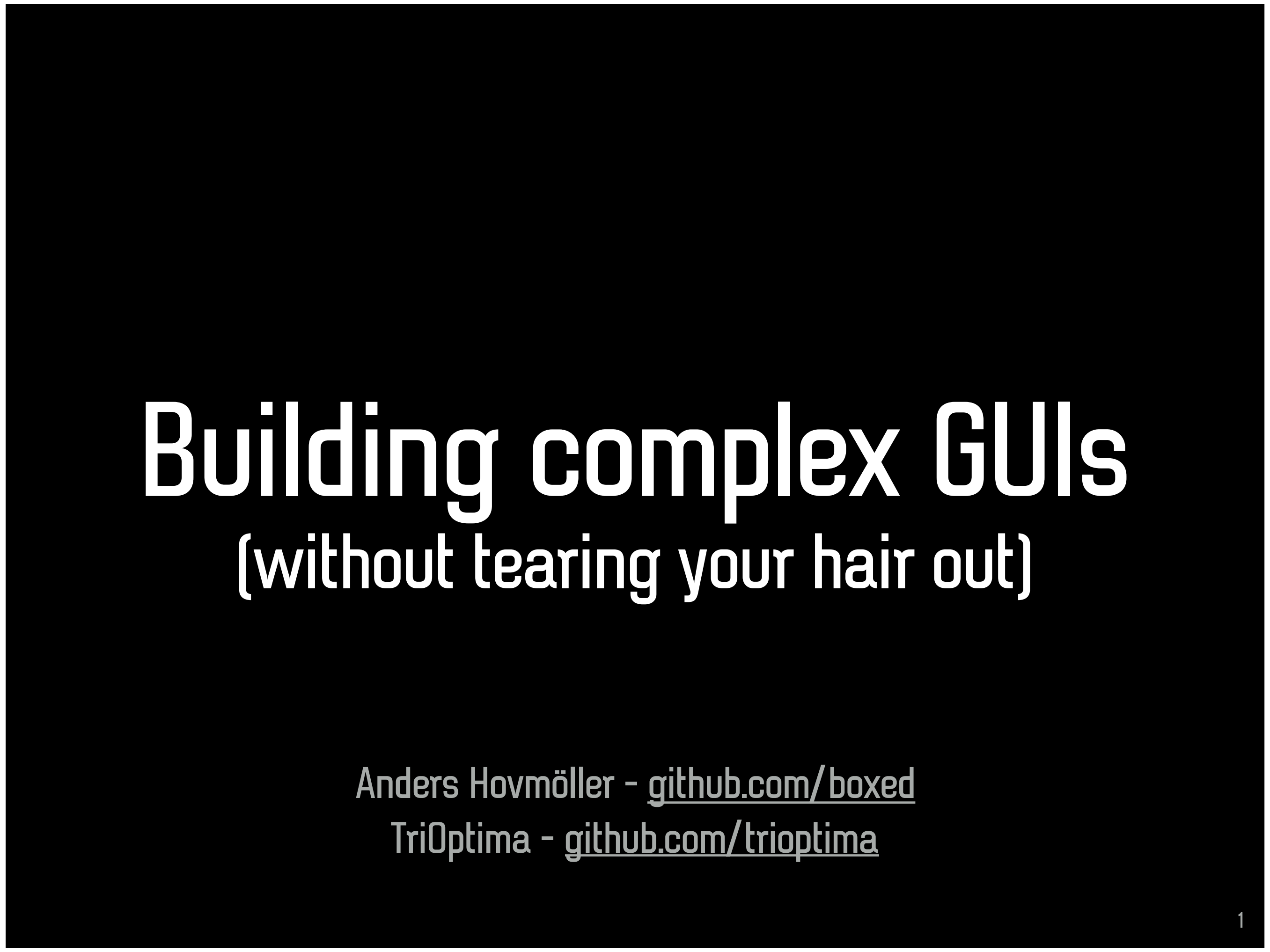


Building complex GUIs

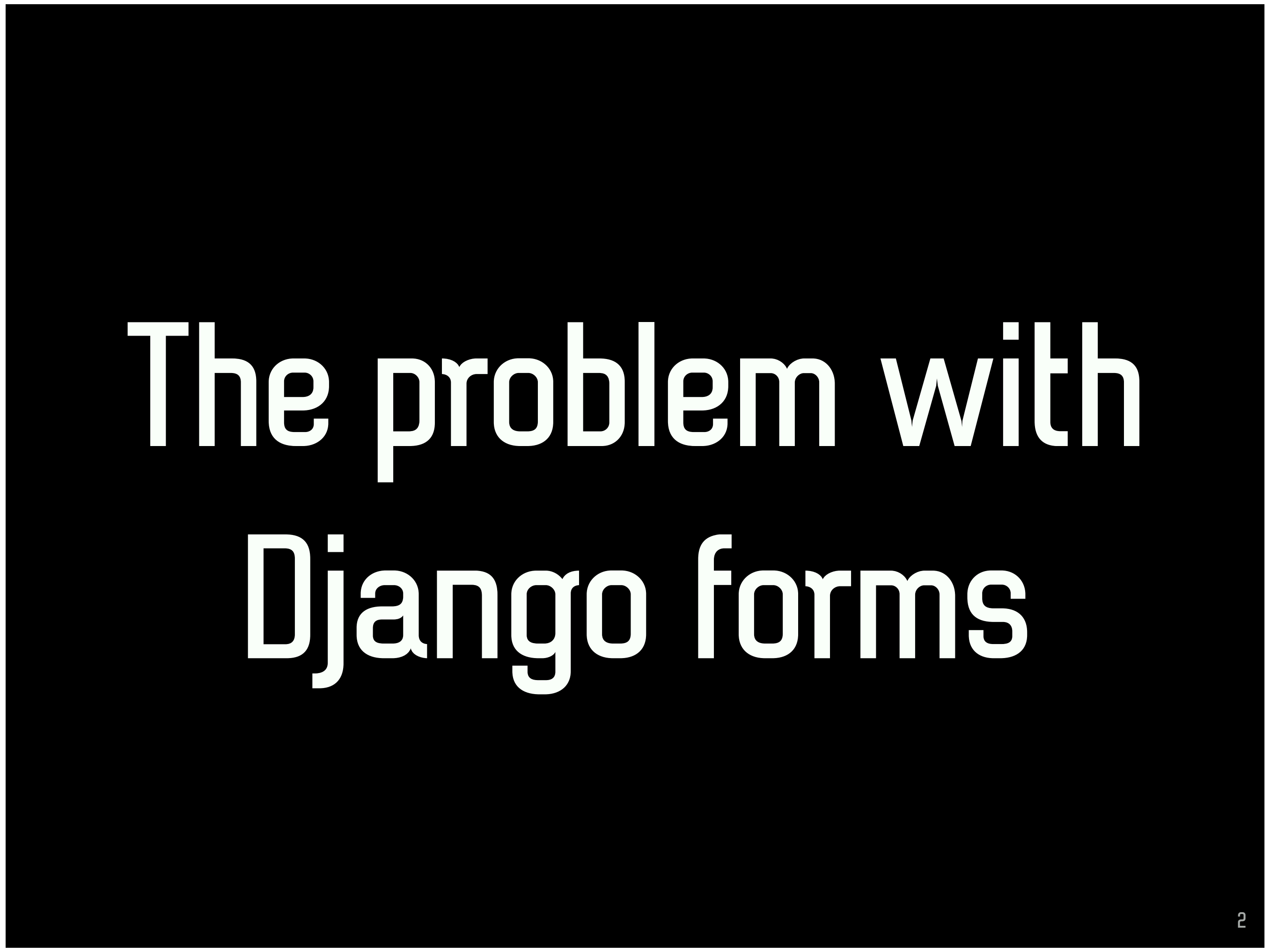
(without tearing your hair out)

Anders Hovmöller - github.com/boxed
TriOptima - github.com/trioptima

Hi! My name is Anders, and I'm going to talk about a pattern slash philosophy me and my colleague Johan Lübcke have developed. This design philosophy is especially nice for building GUIs.

I'm going to start very concretely by showing progressively more complex Django form examples and we'll see how we push up against the limitations of it's basic design style. And I'll be clear here: this is the basic design style of pretty much all modern software. Django is the standard way implemented very well.

Then I'll introduce tri.form and showcase how it deals with those problems. Later on I'll move on to tri.query and tri.table, and then I'll show how to implement this style yourself.

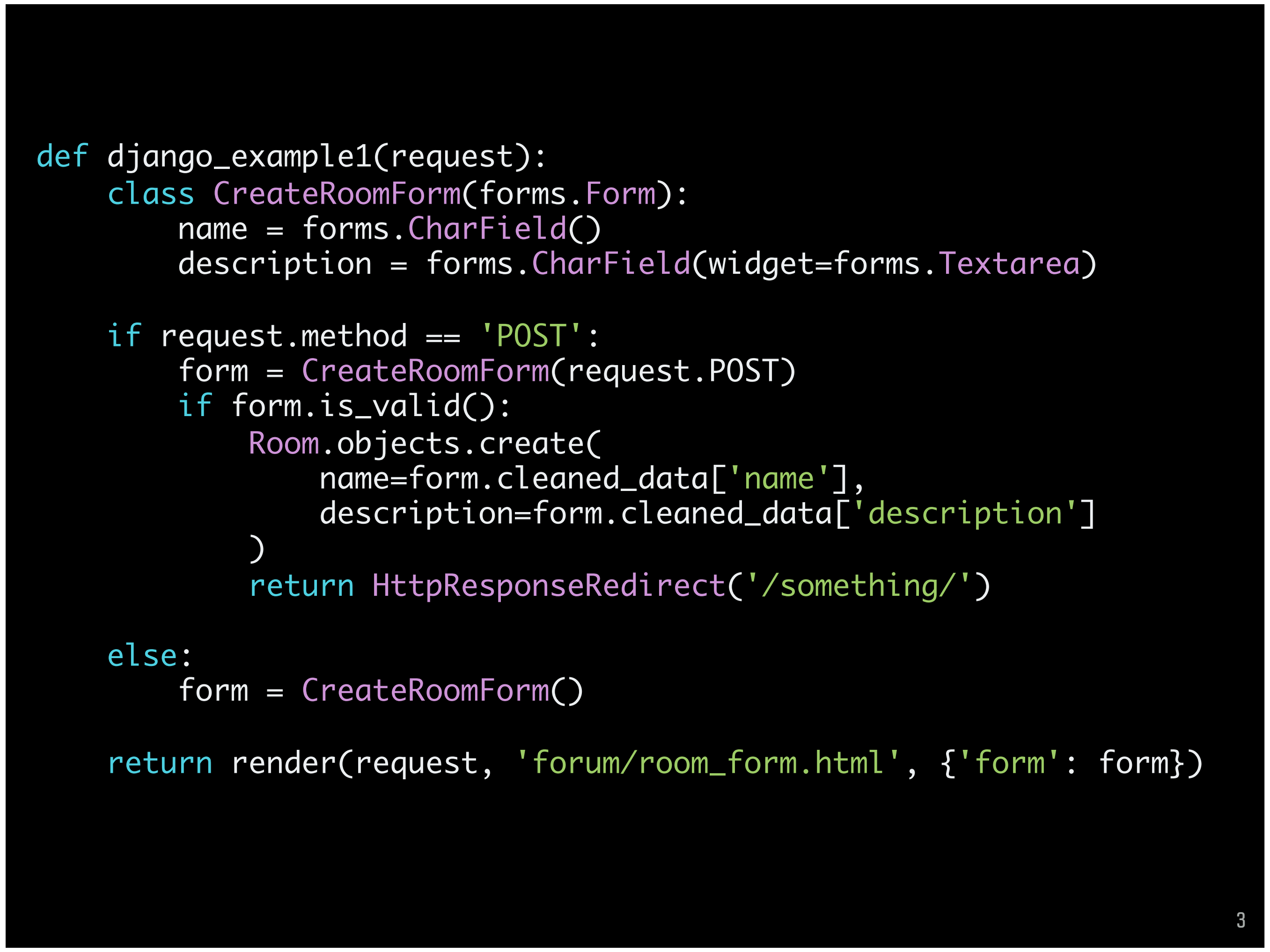


The problem with Django forms

Some background on the product I work on, it's a pretty old code base and it weighs in at roughly 280k dry lines of python code.

We have over 200 forms, most pretty simple, but a few have grown over the years to some real monsters. We tried very hard to not develop our own form library, but it became inevitable at a certain point due to the pain points we were having.

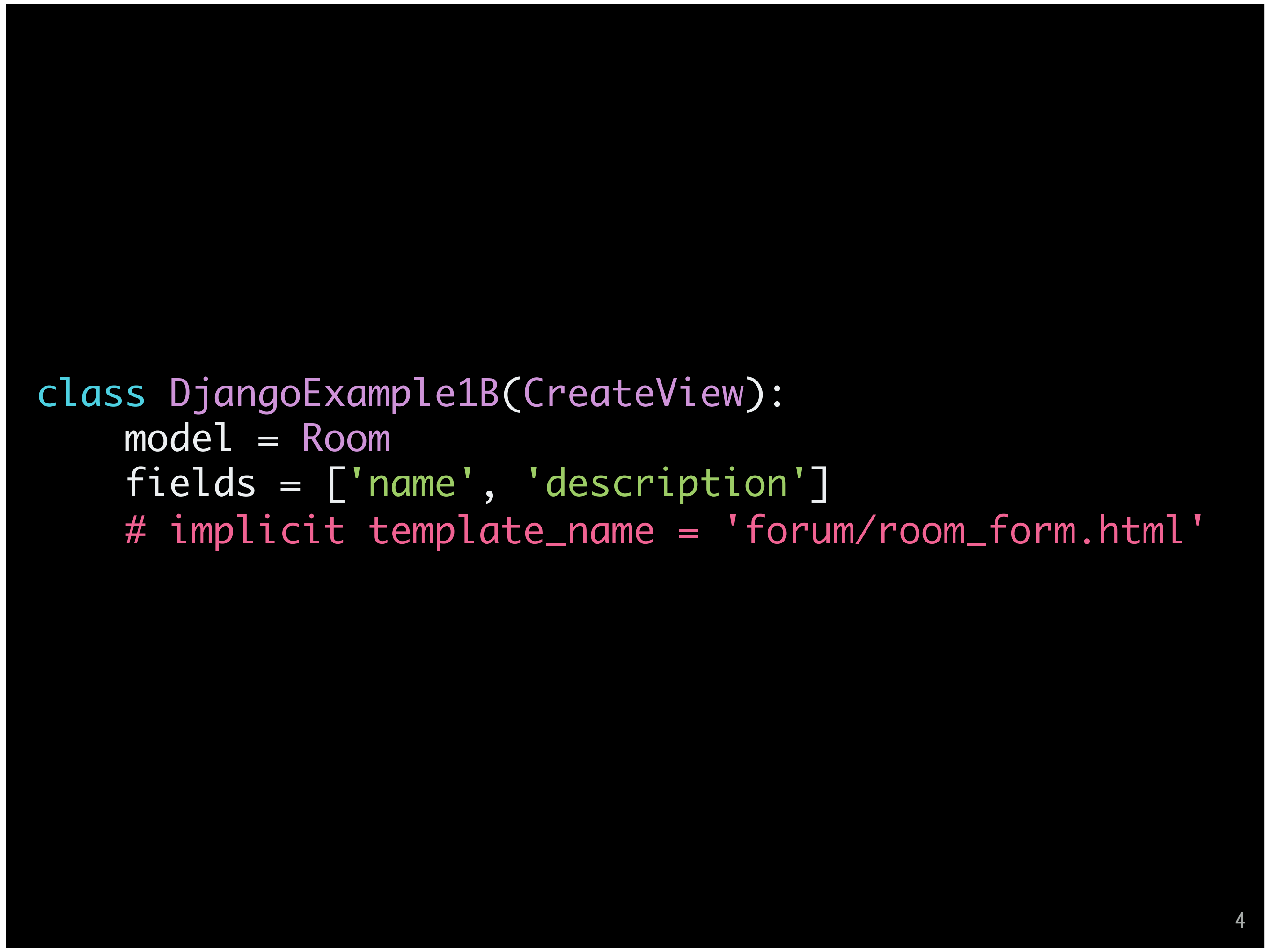
To give you a taste of why we'd do this I'd like to walk through some progressive examples, starting with the most simple case, pretty much straight from the Django docs:



```
def django_example1(request):
    class CreateRoomForm(forms.Form):
        name = forms.CharField()
        description = forms.CharField(widget=forms.Textarea)

    if request.method == 'POST':
        form = CreateRoomForm(request.POST)
        if form.is_valid():
            Room.objects.create(
                name=form.cleaned_data['name'],
                description=form.cleaned_data['description']
            )
            return HttpResponseRedirect('/something/')

    else:
        form = CreateRoomForm()

    return render(request, 'forum/room_form.html', {'form': form})
```

This is pretty good! We like this.

Moving on to the more automatic case where we use class based views from Django to get more for free.



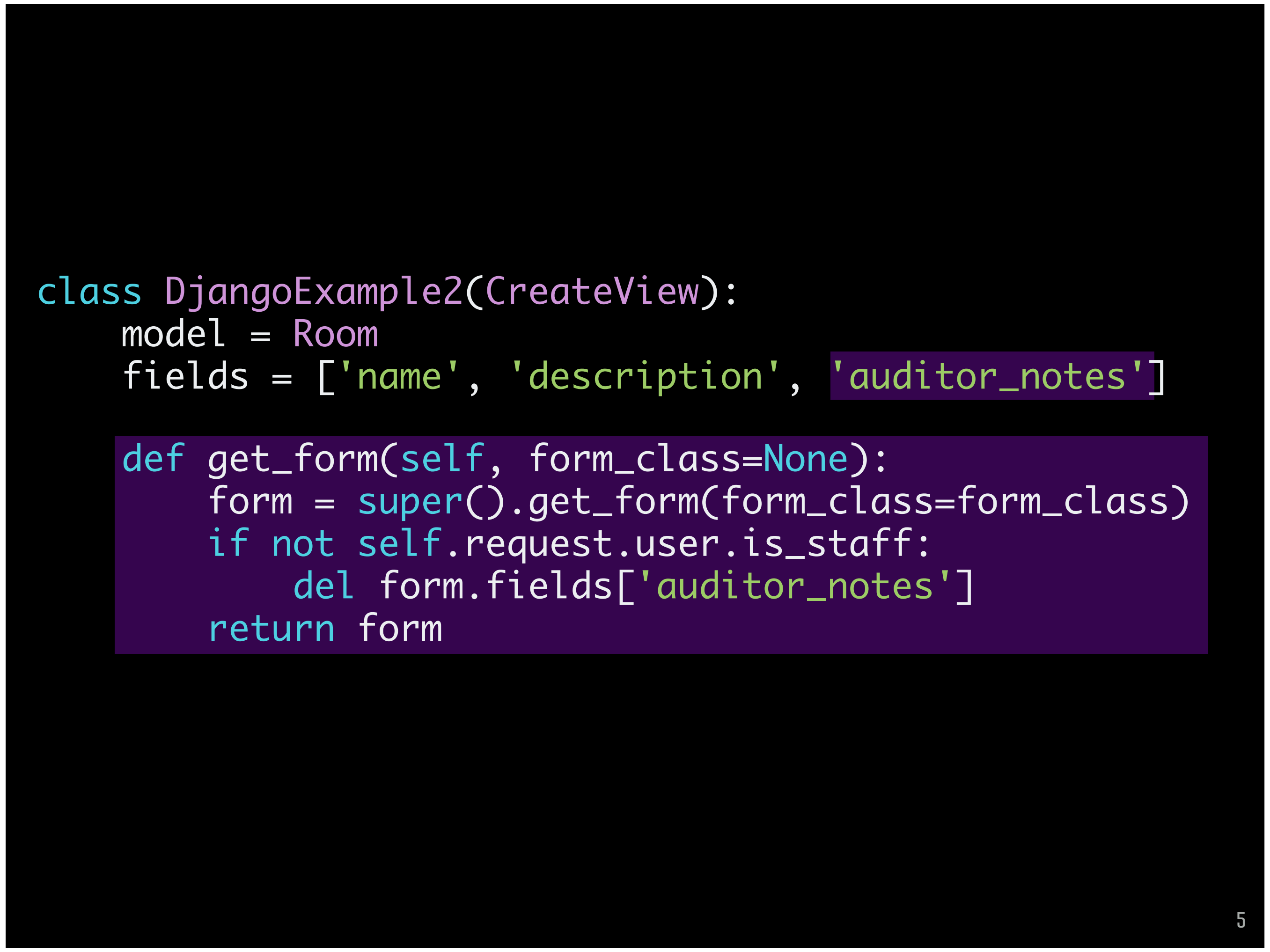
```
class DjangoExample1B(CreateView):
    model = Room
    fields = ['name', 'description']
    # implicit template_name = 'forum/room_form.html'
```

This creates the form fields of the correct type based on the types in the model.

This is nicer than the last view, but there are some things we don't like here. Most notably the implicit template name. Implicit makes it harder to realize that there even IS a template, it makes it easy to make mistakes like deleting the template because when grepping the code base there is no reference to it. This can also mean old obsolete templates hang around forever because no one dares deletes templates.

This is too loosely coupled components. Things should be well coupled. Not too much, but not so loose that everything becomes jelly.

Let's increase the complexity just a little bit. We'll use a forum app as an example. So we have rooms and we'd like to audit them from time to time for acceptable content. We'll start by just introducing a field `auditor_notes`, that is only writable by staff:

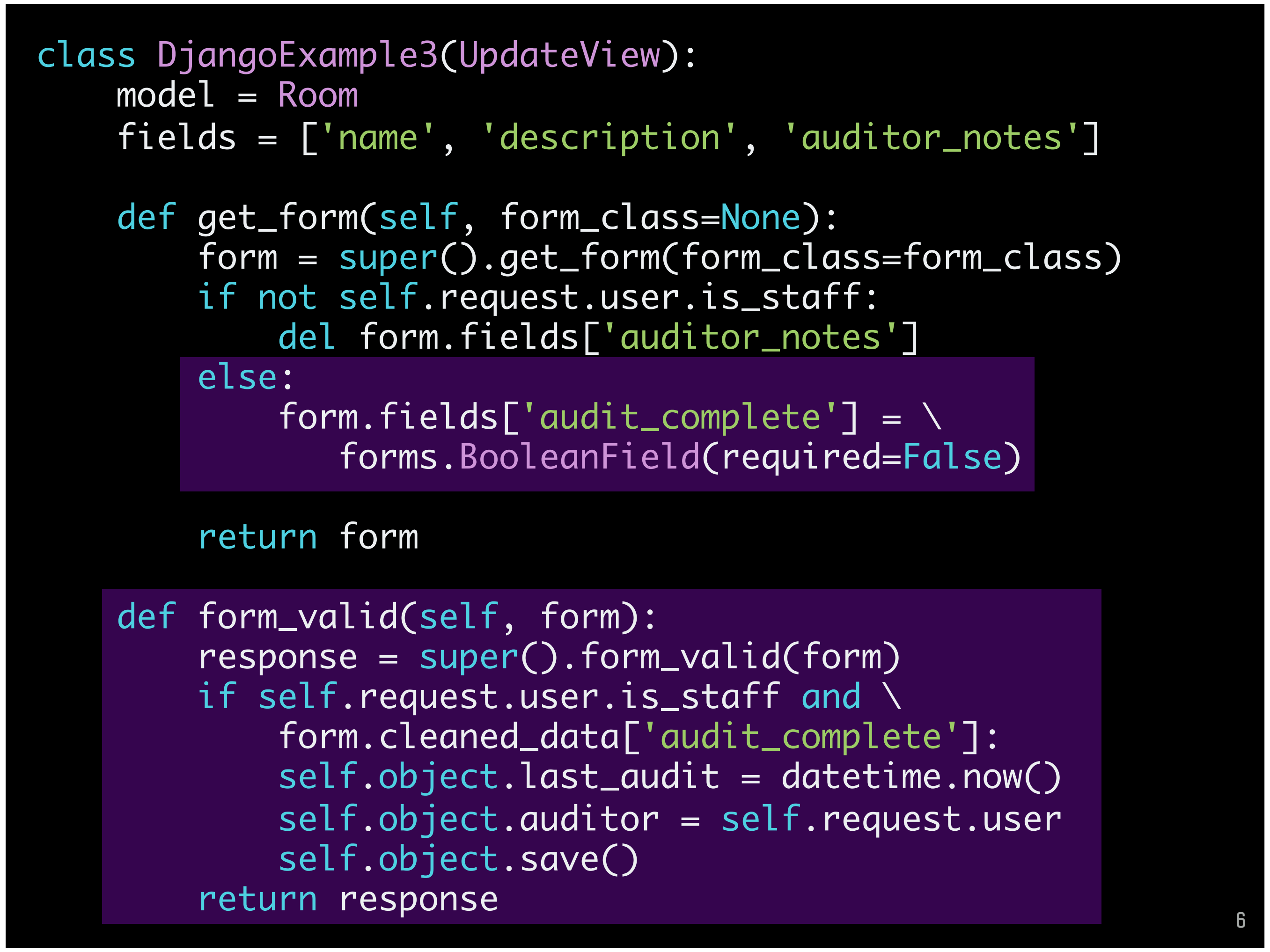


```python
class DjangoExample2(CreateView):
    model = Room
    fields = ['name', 'description', 'auditor_notes']

    def get_form(self, form_class=None):
        form = super().get_form(form_class=form_class)
        if not self.request.user.is_staff:
            del form.fields['auditor_notes']
        return form
```

We override a method and add 5 lines, 3 of which are boilerplate. If we misspell `get_form` here we just silently get all fields.

It's not super bad, but we're starting to see that the code is getting messy a bit faster than we're adding business logic.

In the next example we add the feature that the staff user can check a checkbox to say that they have audited the room. This will save the time of the completed audit and who did the audit.

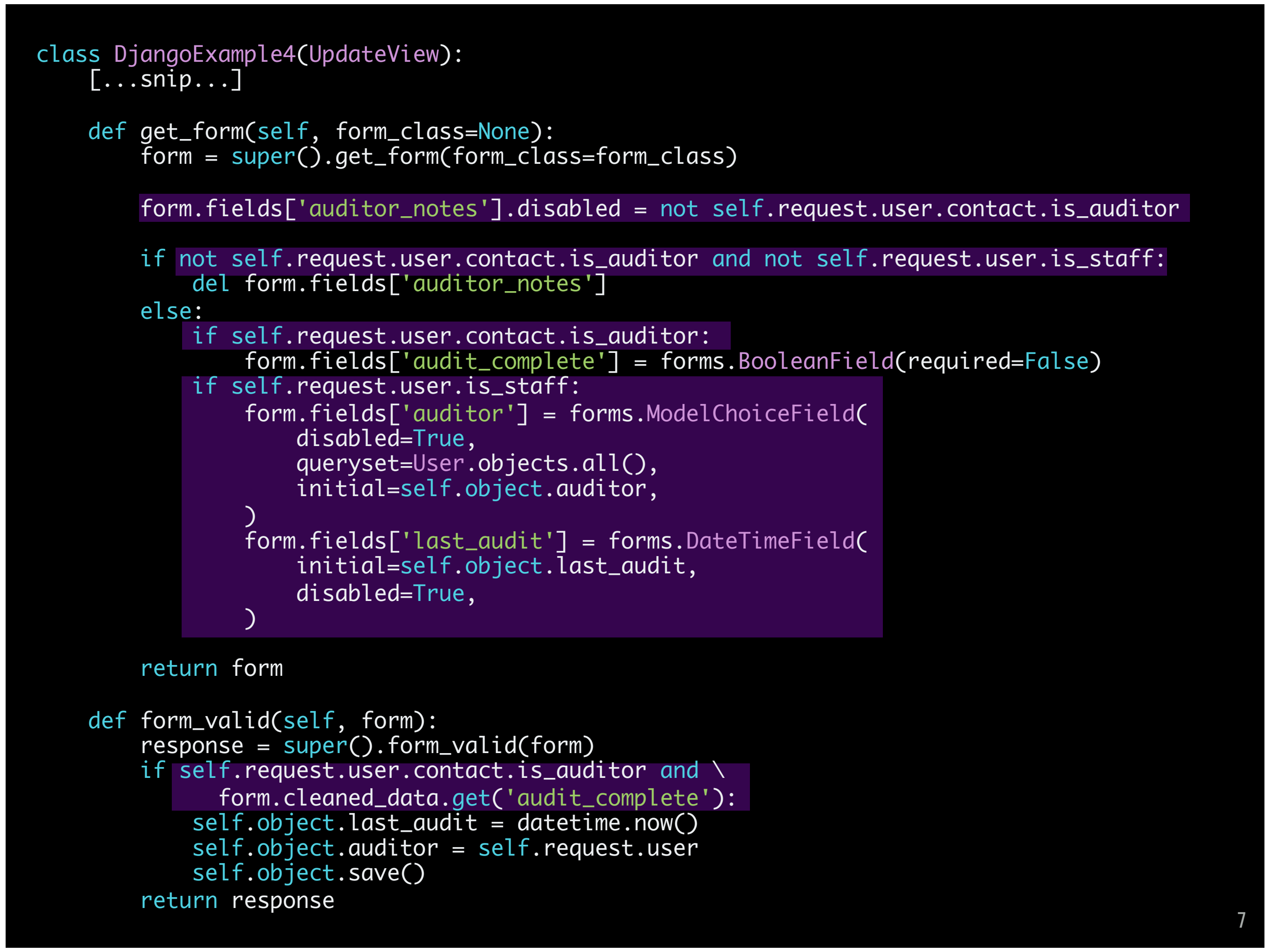


```
class DjangoExample3(UpdateView):
    model = Room
    fields = ['name', 'description', 'auditor_notes']

    def get_form(self, form_class=None):
        form = super().get_form(form_class=form_class)
        if not self.request.user.is_staff:
            del form.fields['auditor_notes']
        else:
            form.fields['audit_complete'] = \
               forms.BooleanField(required=False)

        return form

    def form_valid(self, form):
        response = super().form_valid(form)
        if self.request.user.is_staff and \
            form.cleaned_data['audit_complete']:
            self.object.last_audit = datetime.now()
            self.object.auditor = self.request.user
            self.object.save()
        return response
```

The get_form method gets a little bit more complex, and the form_valid() method is added. Two of these added lines are actual business logic.

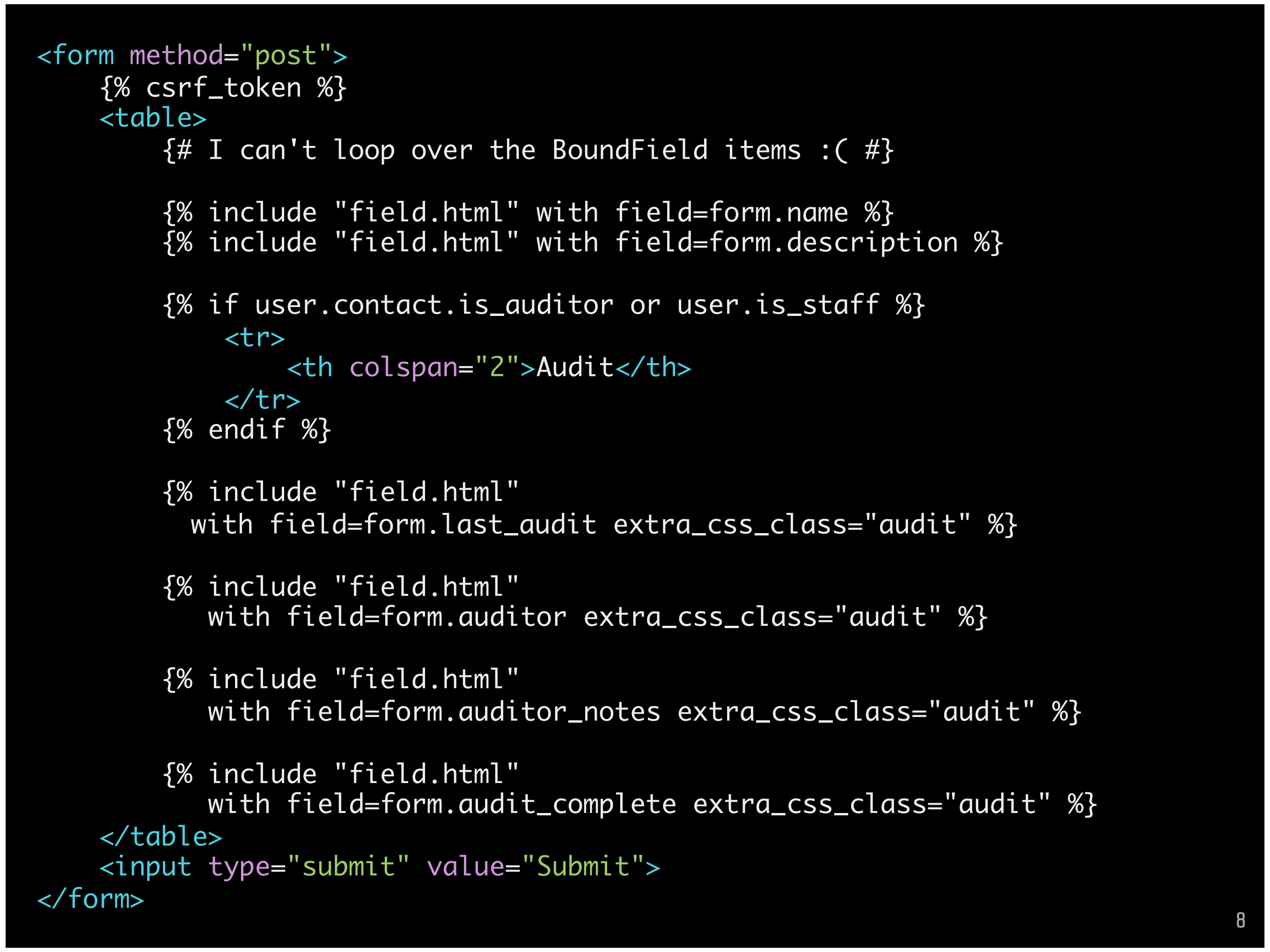
For the next step we'll add 3 requirements:

1. Separate roles of staff and auditors. Now staff should be able to read auditor notes but not edit them, and auditors should be able to perform audits
2. Insert a header above the audit fields to group them nicely
3. Style audit fields with an "admin" css class



```
class DjangoExample4(UpdateView):
    [...snip...]

    def get_form(self, form_class=None):
        form = super().get_form(form_class=form_class)

        form.fields['auditor_notes'].disabled = not self.request.user.contact.is_auditor

        if not self.request.user.contact.is_auditor and not self.request.user.is_staff:
            del form.fields['auditor_notes']
        else:
            if self.request.user.contact.is_auditor:
                form.fields['audit_complete'] = forms.BooleanField(required=False)
            if self.request.user.is_staff:
                form.fields['auditor'] = forms.ModelChoiceField(
                    disabled=True,
                    queryset=User.objects.all(),
                    initial=self.object.auditor,
                )
                form.fields['last_audit'] = forms.DateTimeField(
                    initial=self.object.last_audit,
                    disabled=True,
                )

        return form

    def form_valid(self, form):
        response = super().form_valid(form)
        if self.request.user.contact.is_auditor and \
              form.cleaned_data.get('audit_complete'):
            self.object.last_audit = datetime.now()
            self.object.auditor = self.request.user
            self.object.save()
        return response
```

This is, in my opinion, needlessly complex, but not catastrophic. This implements one of the requirements we wanted. The problem is the template we had to introduce to implement the last two points on our feature list. Let's look at the horrors within!



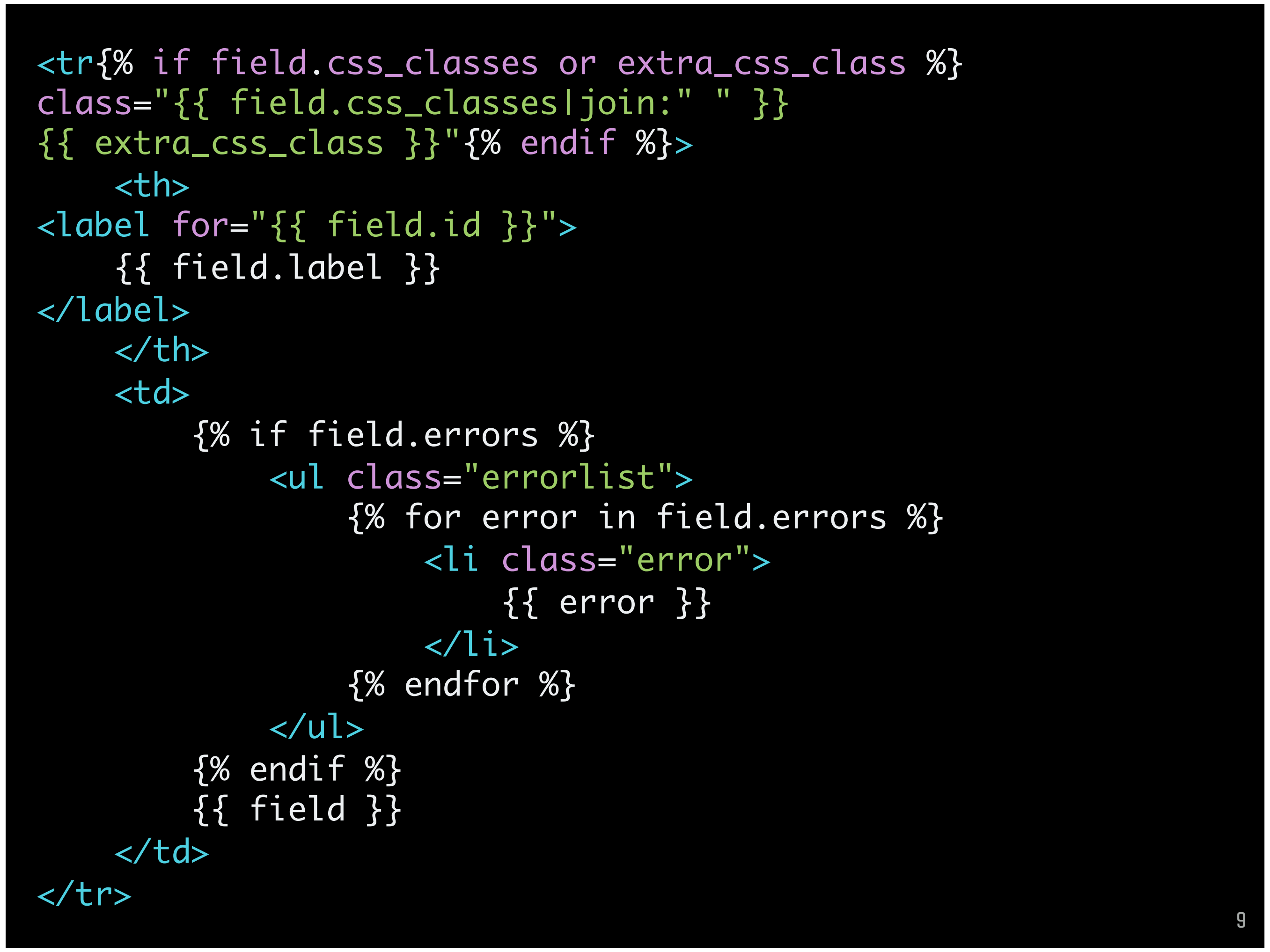
```
<form method="post">
    {% csrf_token %}
    <table>
        {# I can't loop over the BoundField items :( #}

        {% include "field.html" with field=form.name %}
        {% include "field.html" with field=form.description %}

        {% if user.contact.is_auditor or user.is_staff %}
            <tr>
                <th colspan="2">Audit</th>
            </tr>
        {% endif %}

        {% include "field.html"
          with field=form.last_audit extra_css_class="audit" %}

        {% include "field.html"
           with field=form.auditor extra_css_class="audit" %}

        {% include "field.html"
           with field=form.auditor_notes extra_css_class="audit" %}

        {% include "field.html"
           with field=form.audit_complete extra_css_class="audit" %}
    </table>
    <input type="submit" value="Submit">
</form>
```

The fields of a form gets translated into bound fields which are the things we need to render. But we can't access them! So we have to hardcode the fields here. This is DRY violation.

We're also accessing fields blindly and relying on Django rendering empty data when it crashes in the include tag. We find this to be a terrible idea because it makes us used to seeing crashes and errors and we stop seeing them when we really need to. But in this case, it's preferable to the alternative of duplicating the logic an extra time and manually making sure the view and the template are in sync.

The CSS classes for the audit fields are passed as parameters to the include, hardcoded in the template.

The worst part is the included field template though:



```
<tr{% if field.css_classes or extra_css_class %}
class="{{ field.css_classes|join:" " }}
{{ extra_css_class }}"{% endif %}>
    <th>
<label for="{{ field.id }}">
    {{ field.label }}
</label>
    </th>
    <td>
        {% if field.errors %}
            <ul class="errorlist">
                {% for error in field.errors %}
                    <li class="error">
                        {{ error }}
                    </li>
                {% endfor %}
            </ul>
        {% endif %}
        {{ field }}
    </td>
</tr>
```

The label tag rendering is actually wrong here, but since you can't call functions with arguments in Django templates we can't do the same thing as what BaseForm._html_output does, this will have to be close enough.

This template glosses over many details. In reality it should be more complex, but I decided to stop here because it just became too annoying to write all this just for this presentation :P I think this is horrible enough!

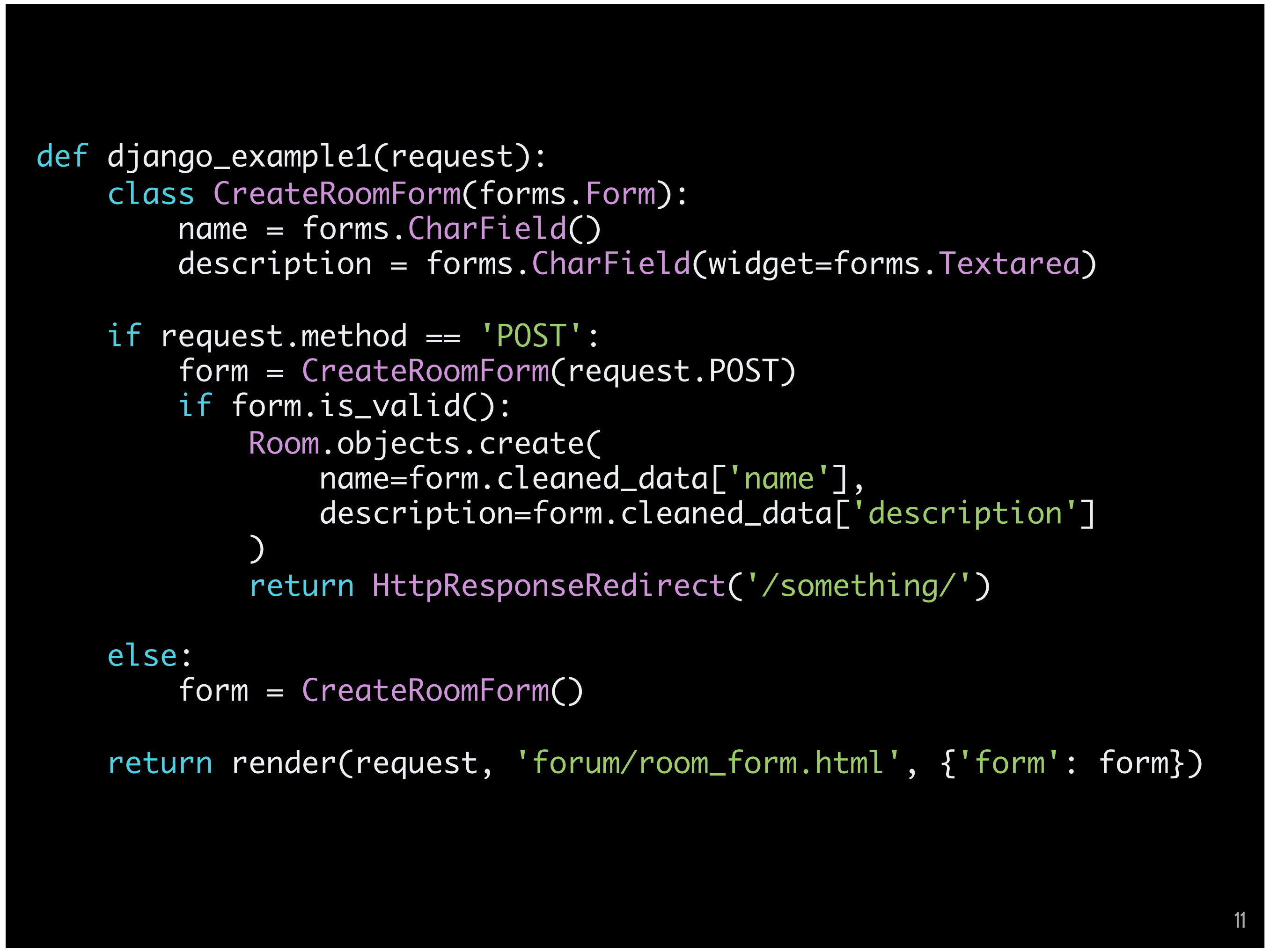
But even all this isn't the worst in my opinion. The real kicker here is that CreateView/UpdateView create their forms via django.forms.models.modelform_factory which has a lot of parameters, but you can't actually use them because there is no way to pass parameters down that chain. "exclude" being maybe the most egregious case. There’s all this nice functionality but we can’t use it.

Ok, so now that I've shown some problems I'll start to talk about solutions!

tri.form

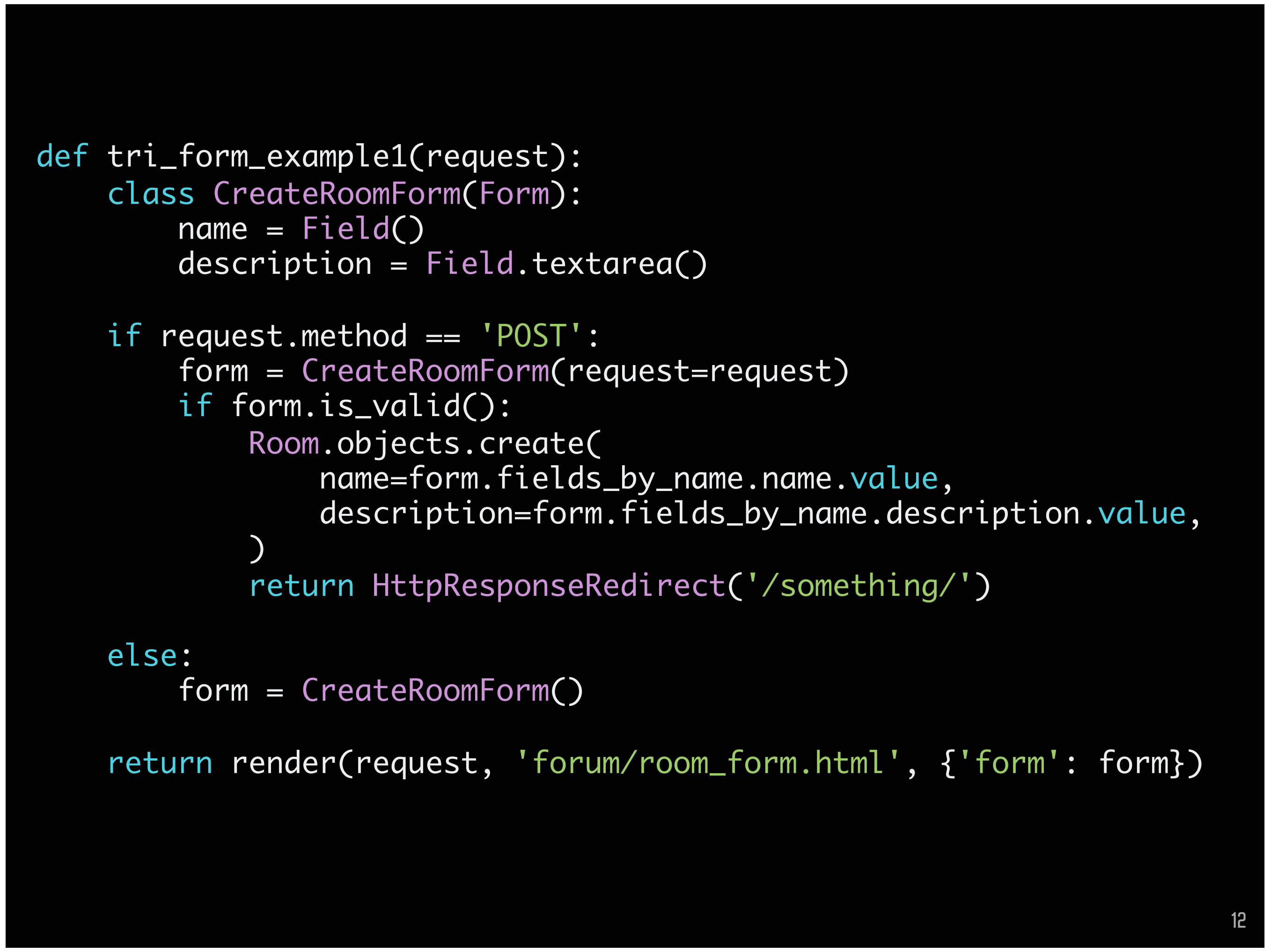
github.com/trioptima/tri.form

tri.form is the library we developed for forms. Now, I've just talked about what I don't like with Django forms but we all really LIKE the basic API. It's just that it doesn't scale with complexity. So what we want is Django forms but just a little bit better. So, let's go back and look at the first Django forms example:



```python
def django_example1(request):
    class CreateRoomForm(forms.Form):
        name = forms.CharField()
        description = forms.CharField(widget=forms.Textarea)

    if request.method == 'POST':
        form = CreateRoomForm(request.POST)
        if form.is_valid():
            Room.objects.create(
                name=form.cleaned_data['name'],
                description=form.cleaned_data['description']
            )
            return HttpResponseRedirect('/something/')

    else:
        form = CreateRoomForm()

    return render(request, 'forum/room_form.html', {'form': form})
```

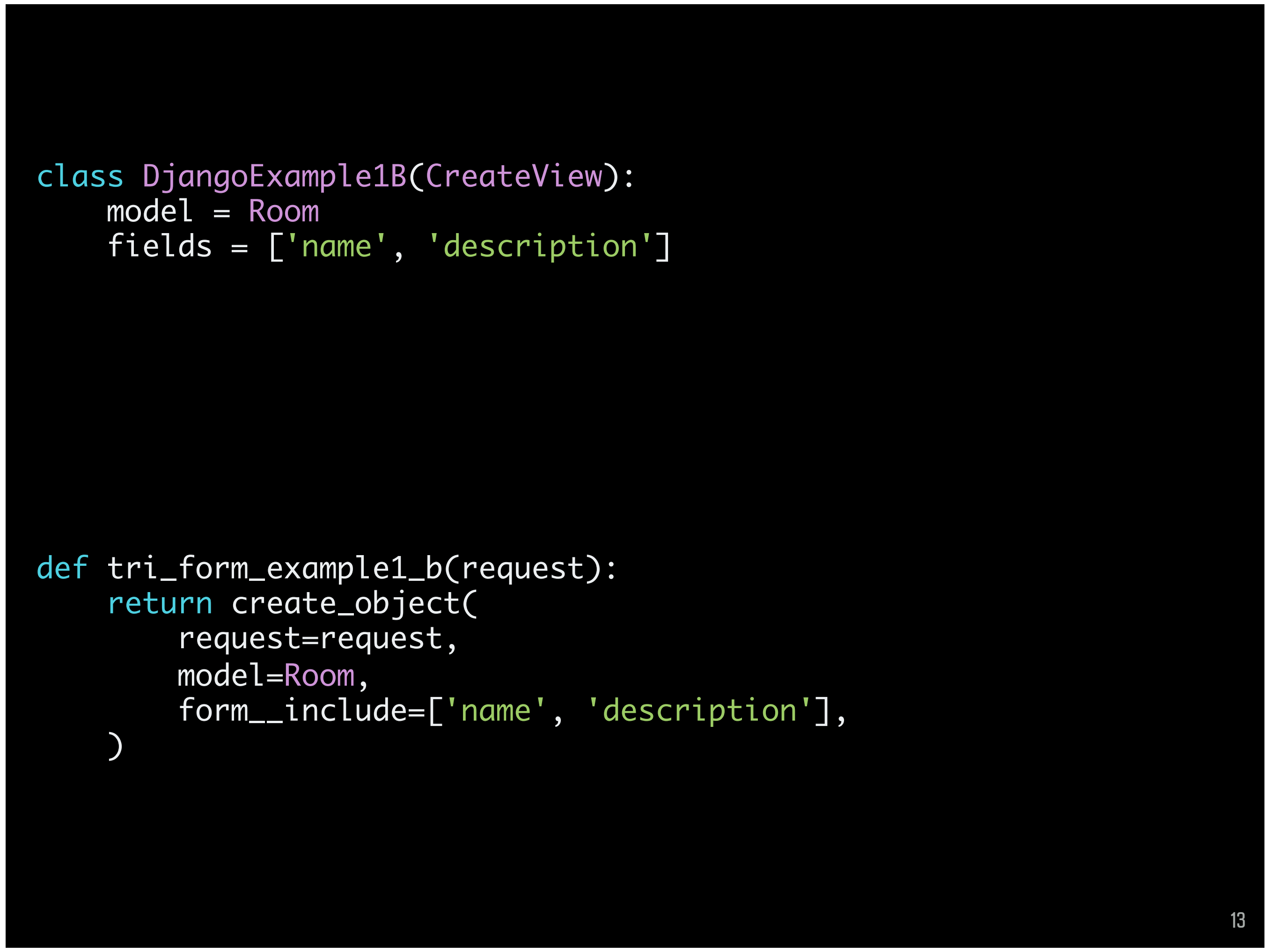
Now let's look at the exact same example in tri.form.



```
def tri_form_example1(request):
    class CreateRoomForm(Form):
        name = Field()
        description = Field.textarea()

    if request.method == 'POST':
        form = CreateRoomForm(request=request)
        if form.is_valid():
            Room.objects.create(
                name=form.fields_by_name.name.value,
                description=form.fields_by_name.description.value,
            )
            return HttpResponseRedirect('/something/')

    else:
        form = CreateRoomForm()

    return render(request, 'forum/room_form.html', {'form': form})
```

As you can see there are minimal differences. Due to our design we don't need full custom classes for different field types, instead we supply some prepackaged configuration bundles under the Field class itself. The textarea "shortcut" as we call it, is a simple configuration that points to a one line template.

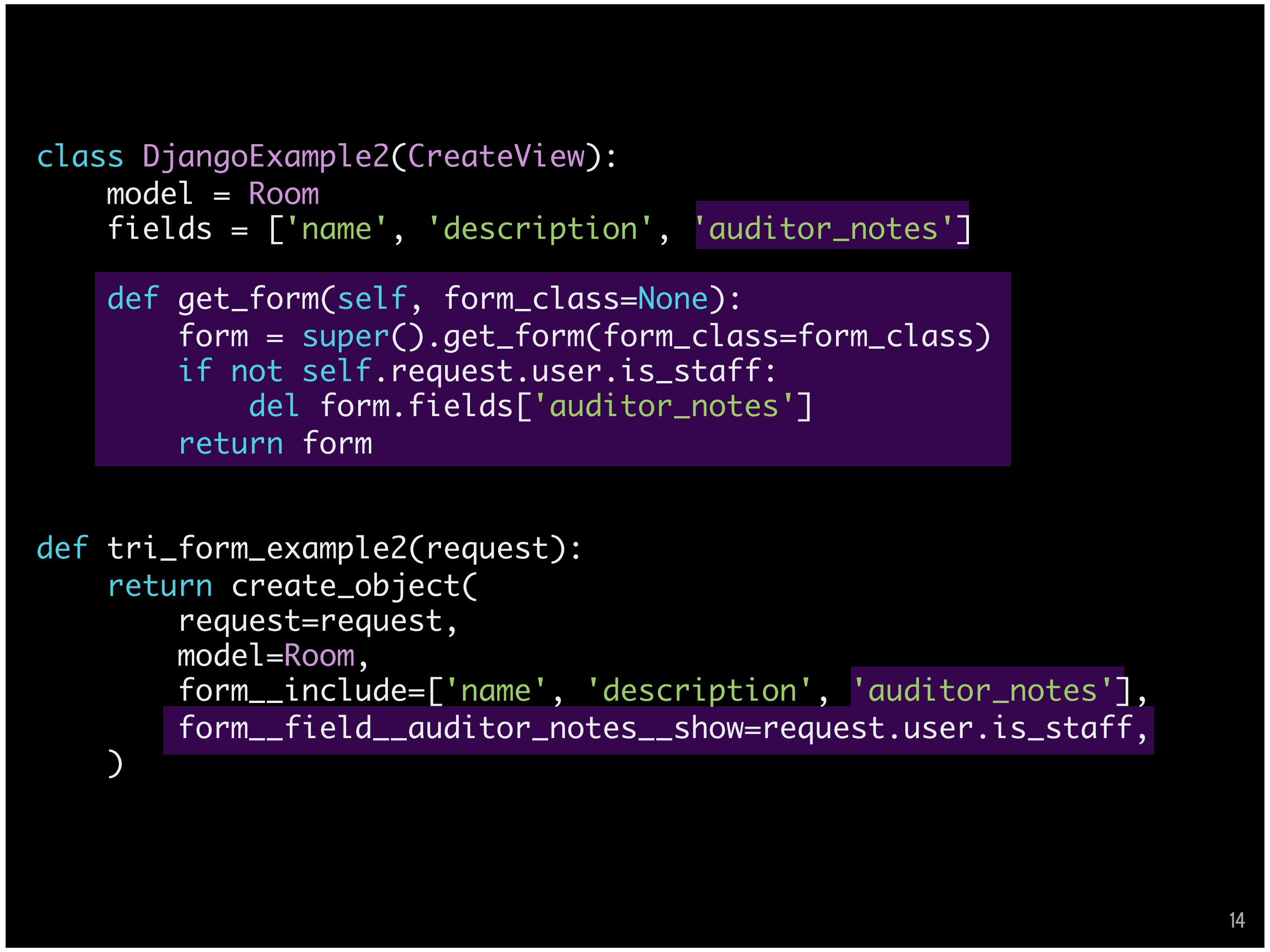
Next I'll move on to the automatic case where the form fields are generated based on the model. I will show both the Django example I showed before and compare it to tri.form:



```python
class DjangoExample1B(CreateView):
    model = Room
    fields = ['name', 'description']




def tri_form_example1_b(request):
    return create_object(
        request=request,
        model=Room,
        form__include=['name', 'description'],
    )
```

Now the design of tri.form and Django start to differ more. We are pretty old school and haven't switched to class based views, which leaks through here. The most notable thing here is `form__include`, which is a pattern we'll see more of. We really like the double underscore-path syntax of the Django query language to navigate tree-like data structures. We have extended this throughout our libraries. In this case `form` is a namespace that can be configured, and we send it the `include` parameter. Also we don't need a template.

Let's move to the next example.

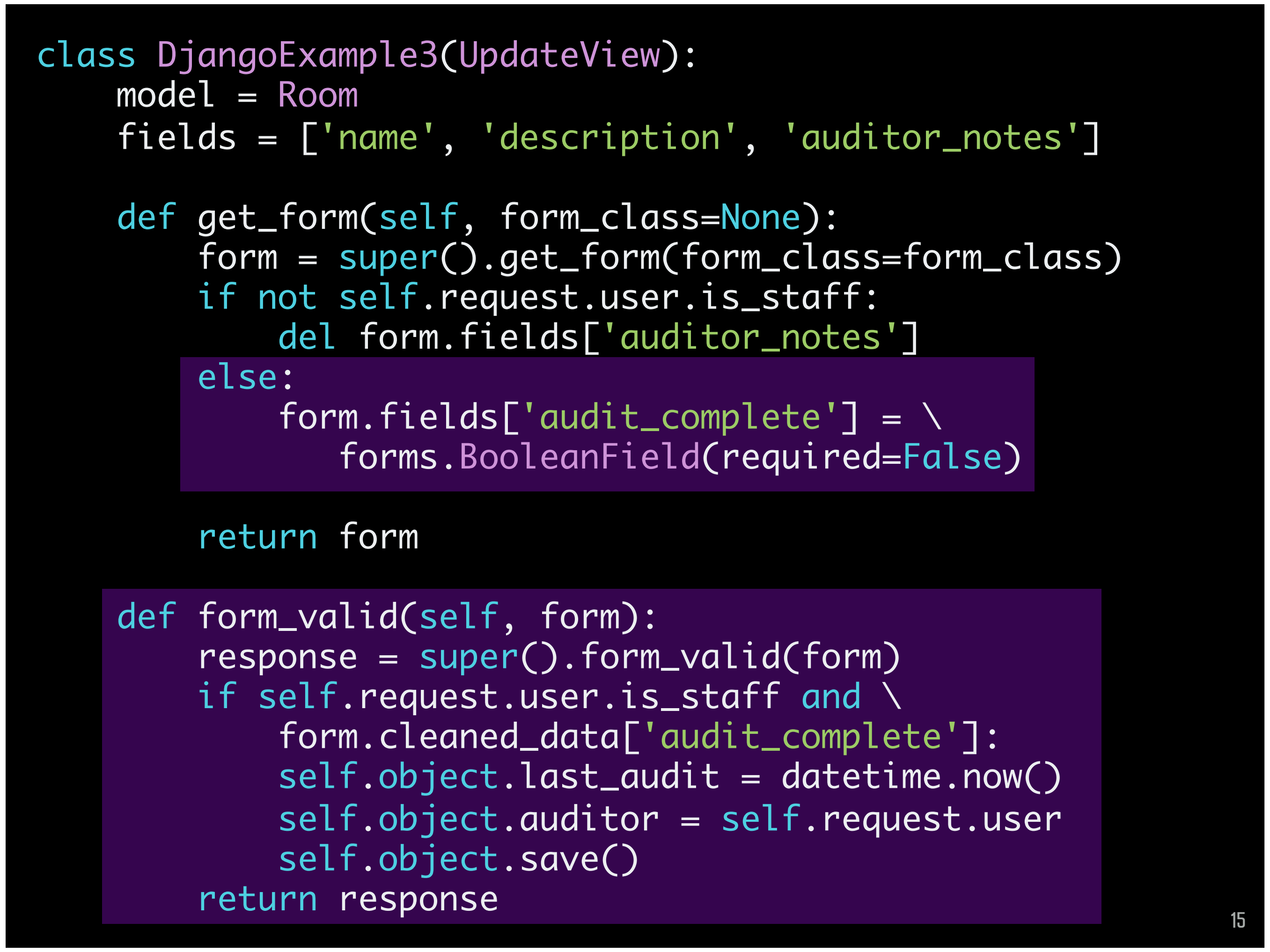


```
class DjangoExample2(CreateView):
    model = Room
    fields = ['name', 'description', 'auditor_notes']

    def get_form(self, form_class=None):
        form = super().get_form(form_class=form_class)
        if not self.request.user.is_staff:
            del form.fields['auditor_notes']
        return form


def tri_form_example2(request):
    return create_object(
        request=request,
        model=Room,
        form__include=['name', 'description', 'auditor_notes'],
        form__field__auditor_notes__show=request.user.is_staff,
    )
```

In the tri.form case we tell tri.form we also want the auditor_notes field, and then we add an override of a default value `form__field__auditor_notes__show` from true to is_staff. This shows more of the nested namespaces I talked about above. The path here means: the form, then the field "auditor notes" then the parameter "show".

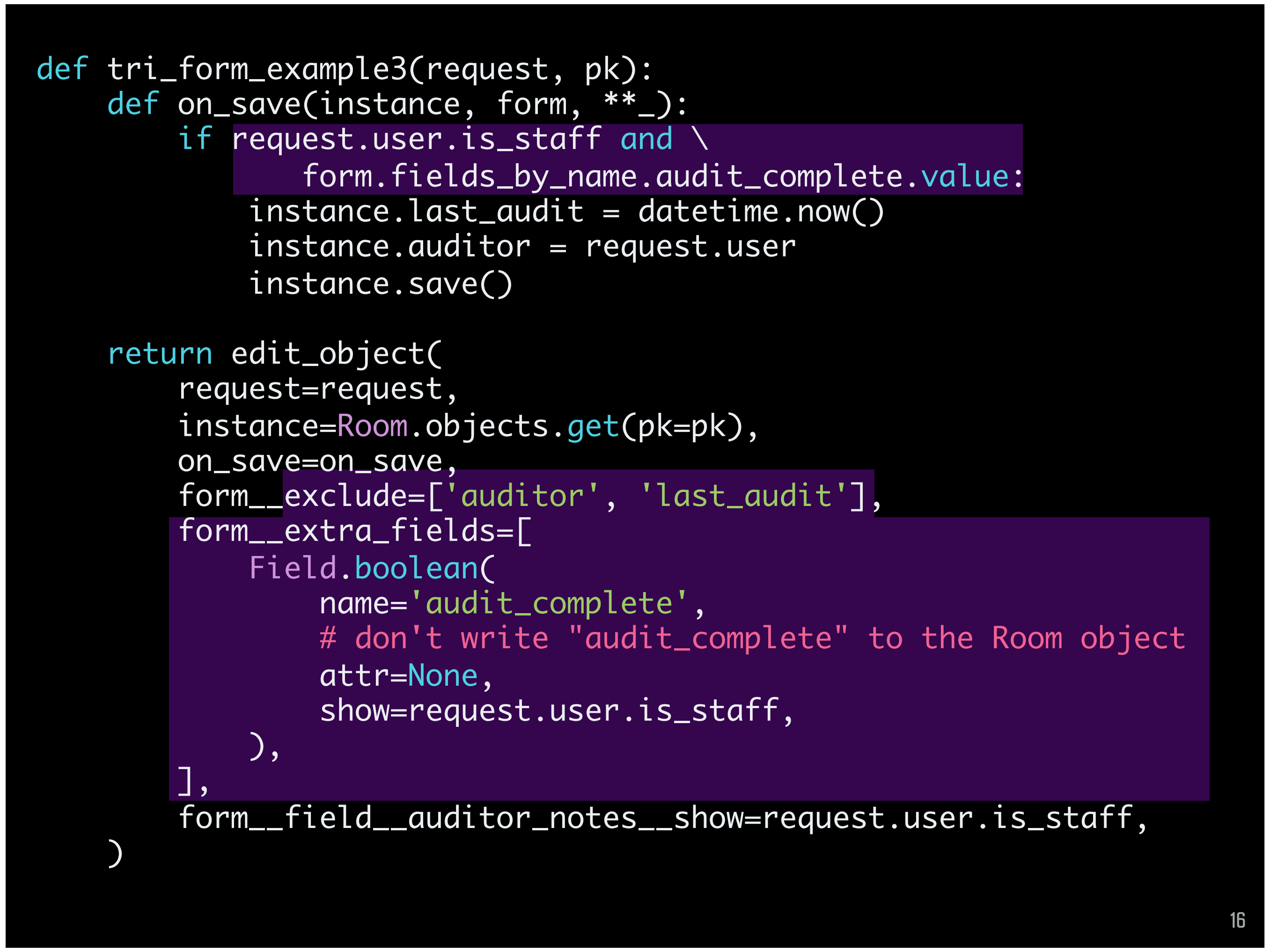
Already I think we're starting to see that the tri.form way leads to cleaner code, but it's not that big of a difference yet. In the next example we add the feature that the staff user can check a checkbox to say that they have audited the room. Just like before this will save the time of the completed audit and user who did the audit.

First the Django code to refresh our memory:



```python
class DjangoExample3(UpdateView):
    model = Room
    fields = ['name', 'description', 'auditor_notes']

    def get_form(self, form_class=None):
        form = super().get_form(form_class=form_class)
        if not self.request.user.is_staff:
            del form.fields['auditor_notes']
        else:
            form.fields['audit_complete'] = \
               forms.BooleanField(required=False)

        return form

    def form_valid(self, form):
        response = super().form_valid(form)
        if self.request.user.is_staff and \
            form.cleaned_data['audit_complete']:
            self.object.last_audit = datetime.now()
            self.object.auditor = self.request.user
            self.object.save()
        return response
```

11 added lines. Ok now let’s look at what we had to do in tri.form:

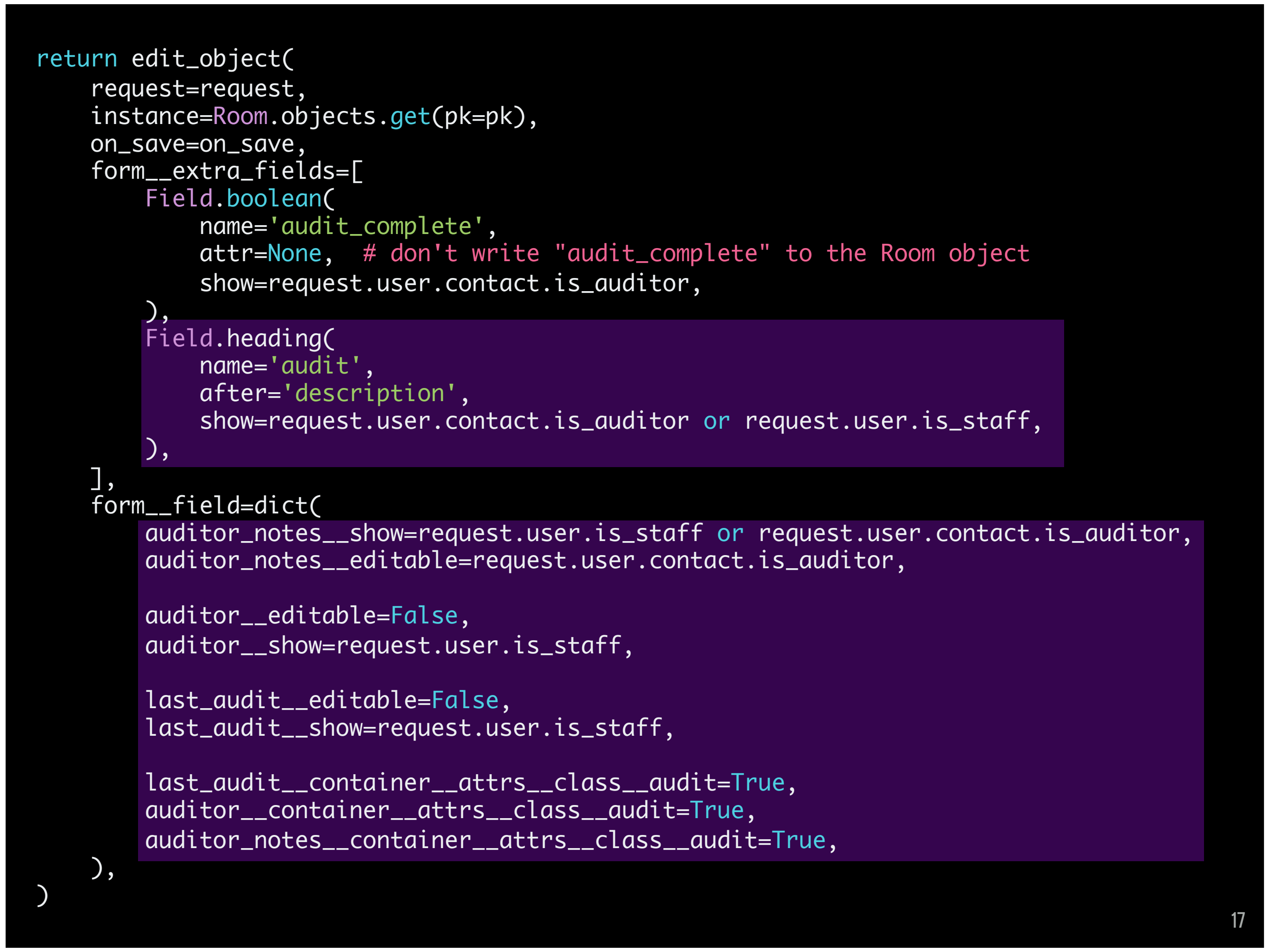


```
def tri_form_example3(request, pk):
    def on_save(instance, form, **_):
        if request.user.is_staff and \
                form.fields_by_name.audit_complete.value:
            instance.last_audit = datetime.now()
            instance.auditor = request.user
            instance.save()

    return edit_object(
        request=request,
        instance=Room.objects.get(pk=pk),
        on_save=on_save,
        form__exclude=['auditor', 'last_audit'],
        form__extra_fields=[
            Field.boolean(
                name='audit_complete',
                # don't write "audit_complete" to the Room object
                attr=None,
                show=request.user.is_staff,
            ),
        ],
        form__field__auditor_notes__show=request.user.is_staff,
    )
```

I've also taken the opportunity here to show that we can do exclude instead of include if that makes more sense to our use case. If we don't specify include or exclude we get all the fields.

We can add new fields with the "extra_fields" parameter. I think this is nicer than mutating an existing form like in Django.

This is not a big difference compared to Django forms in number of lines of Python, but let's move on to example 4:

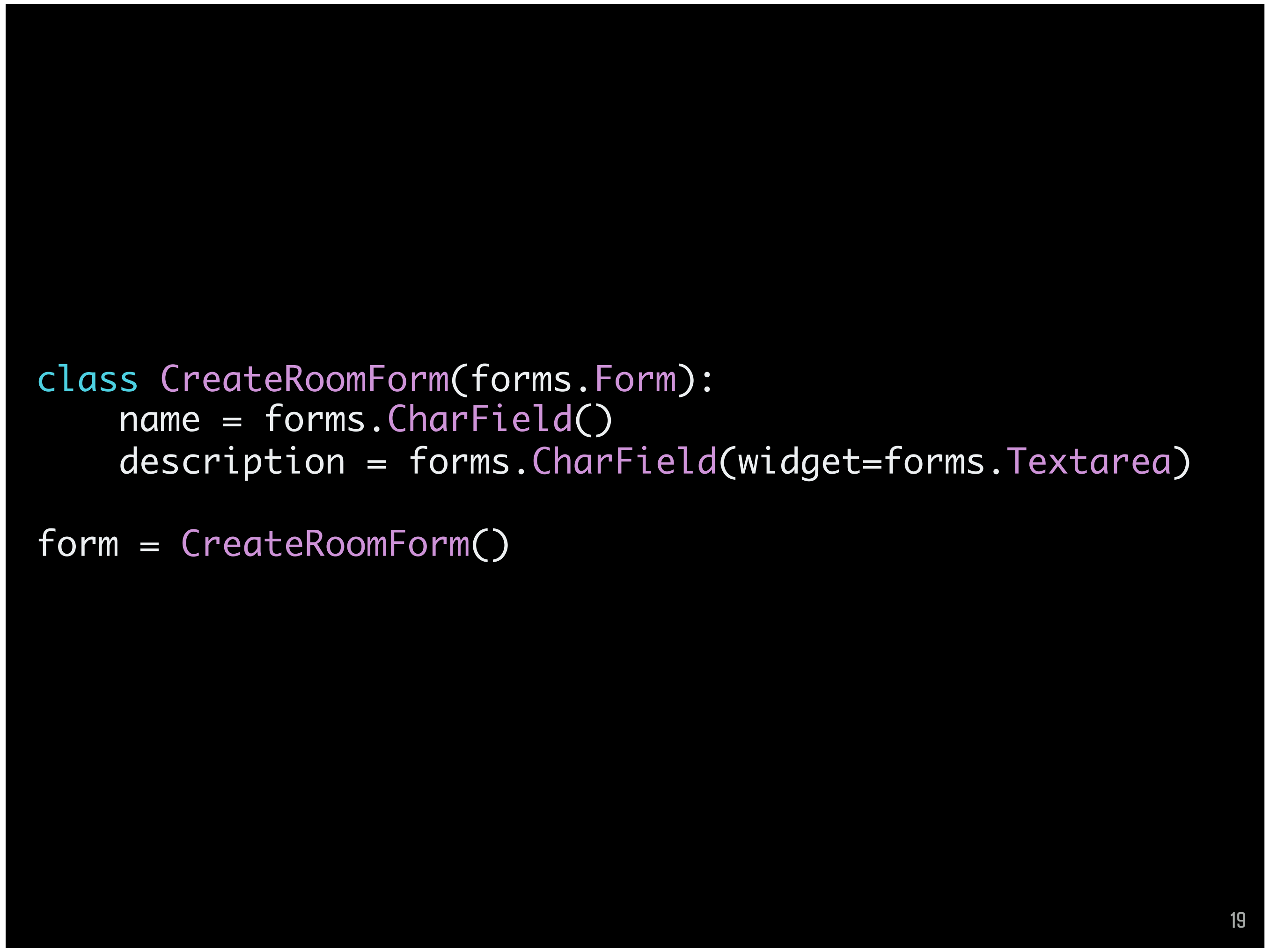


```
return edit_object(
    request=request,
    instance=Room.objects.get(pk=pk),
    on_save=on_save,
    form__extra_fields=[
        Field.boolean(
            name='audit_complete',
            attr=None,  # don't write "audit_complete" to the Room object
            show=request.user.contact.is_auditor,
        ),
        Field.heading(
            name='audit',
            after='description',
            show=request.user.contact.is_auditor or request.user.is_staff,
        ),
    ],
    form__field=dict(
        auditor_notes__show=request.user.is_staff or request.user.contact.is_auditor,
        auditor_notes__editable=request.user.contact.is_auditor,

        auditor__editable=False,
        auditor__show=request.user.is_staff,

        last_audit__editable=False,
        last_audit__show=request.user.is_staff,

        last_audit__container__attrs__class__audit=True,
        auditor__container__attrs__class__audit=True,
        auditor_notes__container__attrs__class__audit=True,
    ),
)
```

Here the added configuration scales linearly with business logic and we don't need to add any templates.

Programmatically creating forms

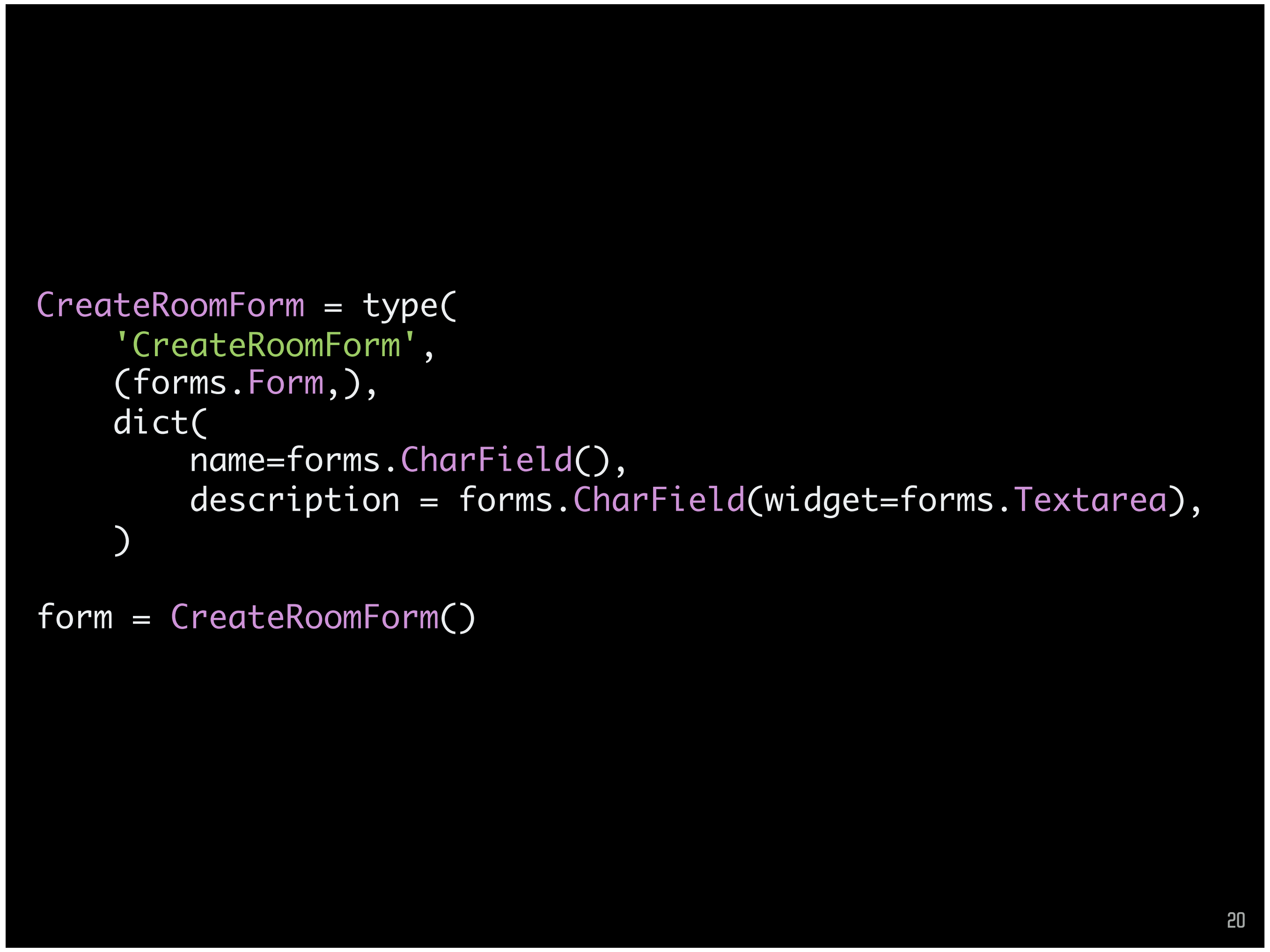
There is another use case that I haven't touched on so far that initially drove me to develop the first implementation of tri.form and that is: programmatically creating forms. With Django forms you need to use the type() constructor. This is very very icky. Let's look at an example, we'll take the very first example we used before.



```python
class CreateRoomForm(forms.Form):
    name = forms.CharField()
    description = forms.CharField(widget=forms.Textarea)

form = CreateRoomForm()
```

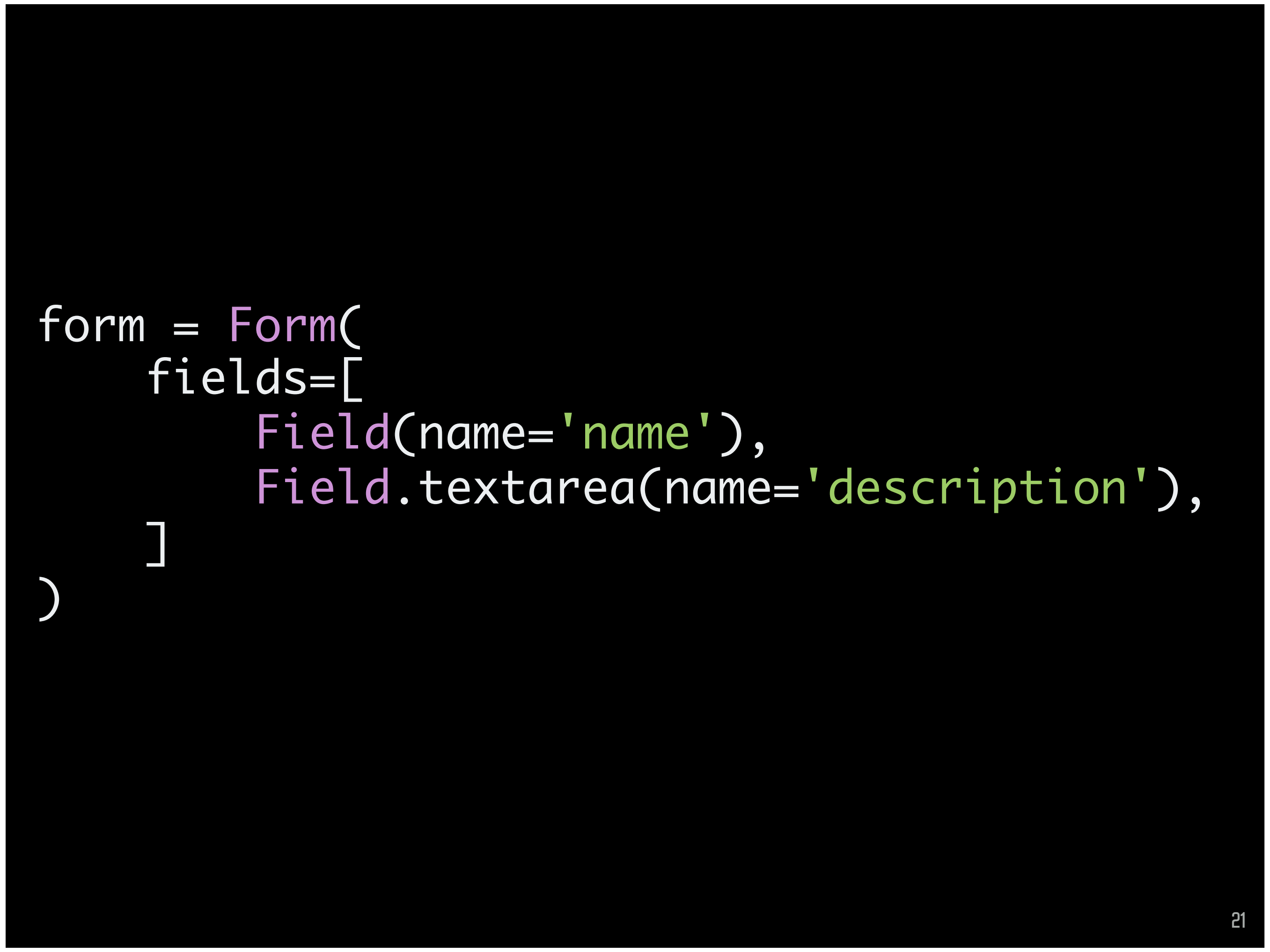
This is the code for the normal declarative syntax again to refresh your memory.

Now what if I have some data driven GUI so I have to create this form programmatically? Well it looks something like this:



```python
CreateRoomForm = type(
    'CreateRoomForm',
    (forms.Form,),
    dict(
        name=forms.CharField(),
        description = forms.CharField(widget=forms.Textarea),
    )

form = CreateRoomForm()
```

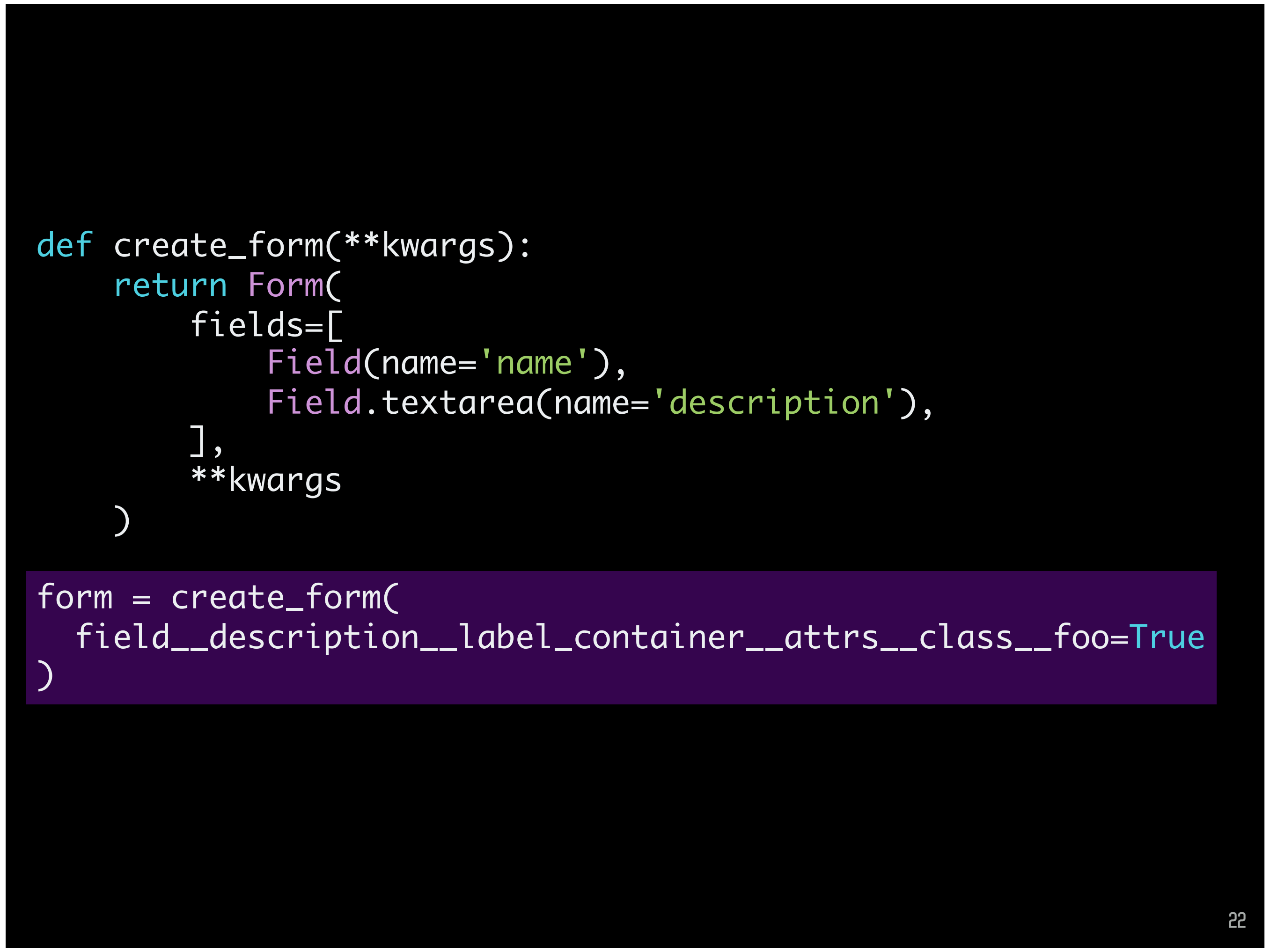
We started out doing this but it very soon became untenable. Why are we even creating a class here when it's only going to be used once? In tri.form the declarative style is actually syntactic sugar and not something you are forced to use. We can just pass the fields as a list. It looks like this:



```
form = Form(
    fields=[
        Field(name='name'),
        Field.textarea(name='description'),
    ]
)
```

The Form constructor uses both inherited members and the fields argument to build the list of fields for the form.

We also don't lose the ability to customize further here, as an example we can wrap the above in a reusable function like this:



```
def create_form(**kwargs):
    return Form(
        fields=[
            Field(name='name'),
            Field.textarea(name='description'),
        ],
        **kwargs
    )

form = create_form(
  field__description__label_container__attrs__class__foo=True
)
```

so if we want that form, but we'd like a CSS class set on the label container of the description field we can navigate the tree of parameters using double underscore syntax.

This avoids the common pattern of having an exploding number of boolean parameters to factory functions. It also enabled us to use this form library as a foundation to build more powerful libraries on top.

This brings me to tri.table and tri.query. I'll start with tri.query:

tri.query

github.com/trioptima/tri.query

tri.query is a library for querying, or filtering, a Django database. It works on top of Django QuerySets. It can render a form for ease of use.

It has an API that looks very familiar to you if you've used Django forms or tri.form:

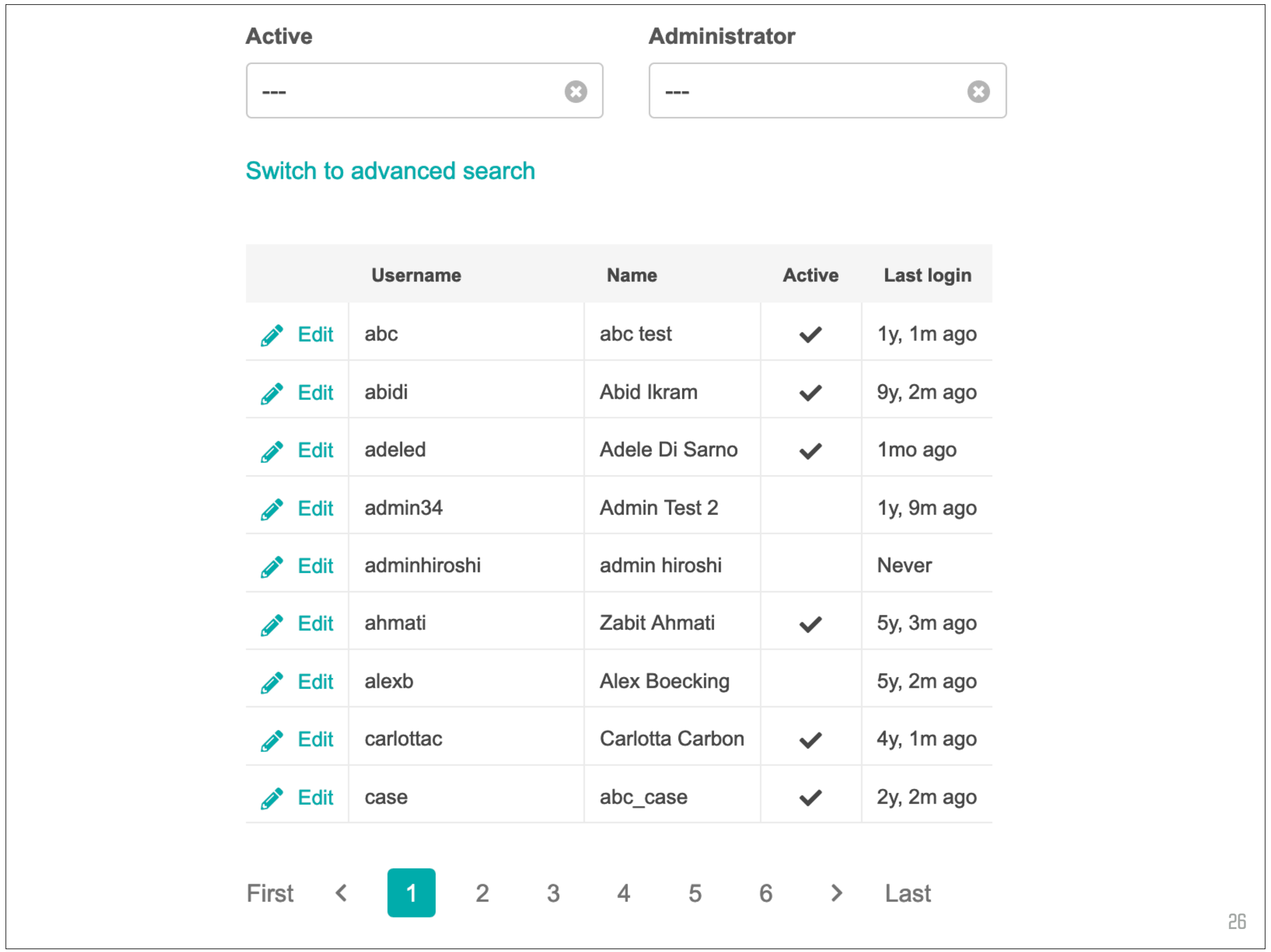
```
class MyQuery(Query):
    name = Variable(gui__show=True)
    description = Variable()
```

You can render this in your Django template and you'll get a search field for name and a button to switch to advanced mode where you can perform full boolean logic on name and description, with operators like contains, startswith, etc. We mostly only use this library indirectly via tri.table, so I'll just move on directly:

tri.table

github.com/trioptima/tri.table

tri.table is a library to create full featured tables in Django apps. Primarily based on QuerySets but you can also use them on any iterable.



Active

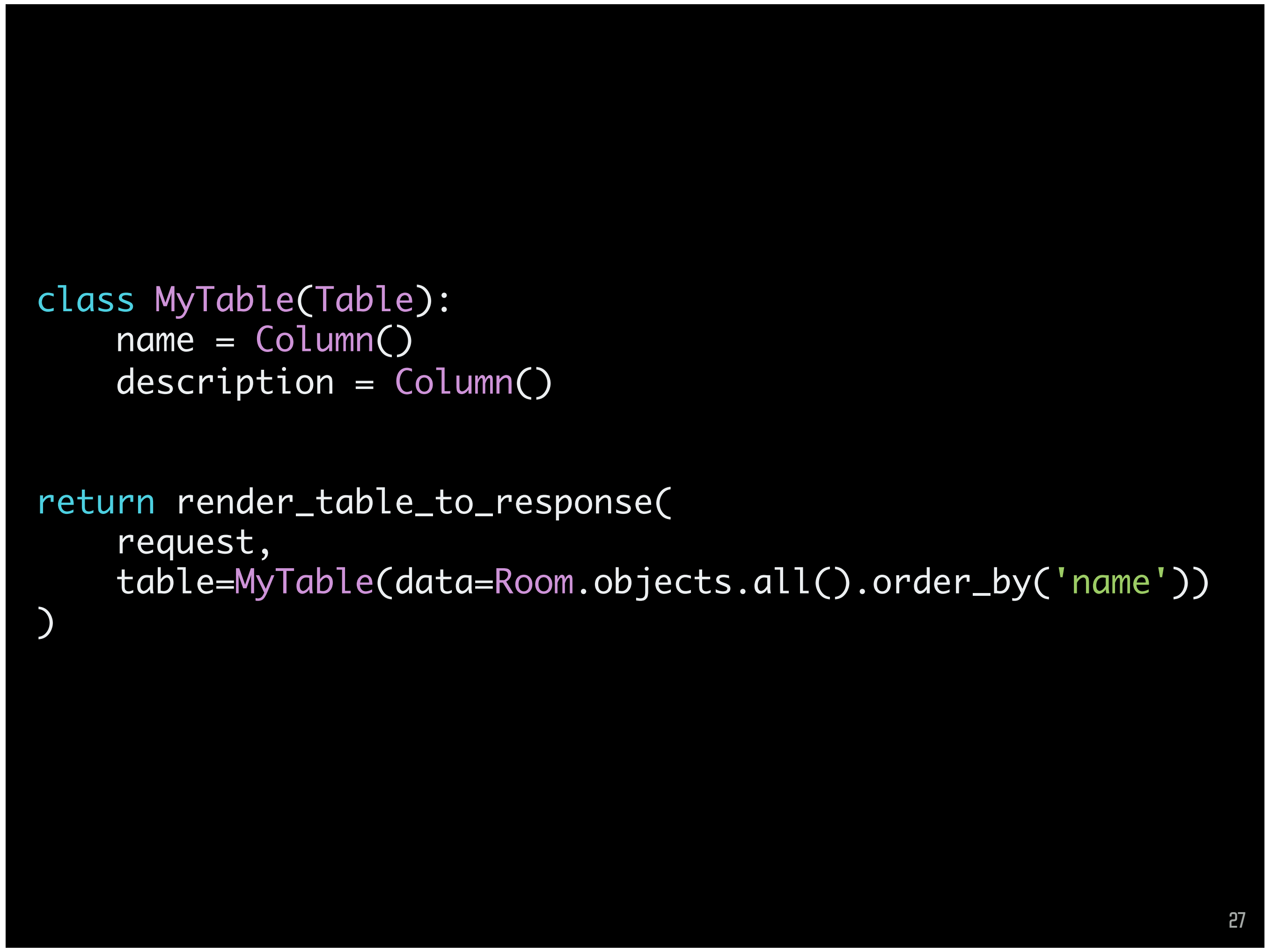
Administrator

Switch to advanced search

	Username	Name	Active	Last login
Edit	abc	abc test	✓	1y, 1m ago
Edit	abidi	Abid Ikram	✓	9y, 2m ago
Edit	adeled	Adele Di Sarno	✓	1mo ago
Edit	admin34	Admin Test 2		1y, 9m ago
Edit	adminhiroshi	admin hiroshi		Never
Edit	ahmati	Zabit Ahmati	✓	5y, 3m ago
Edit	alexb	Alex Boecking		5y, 2m ago
Edit	carlottac	Carlotta Carbon	✓	4y, 1m ago
Edit	case	abc_case	✓	2y, 2m ago

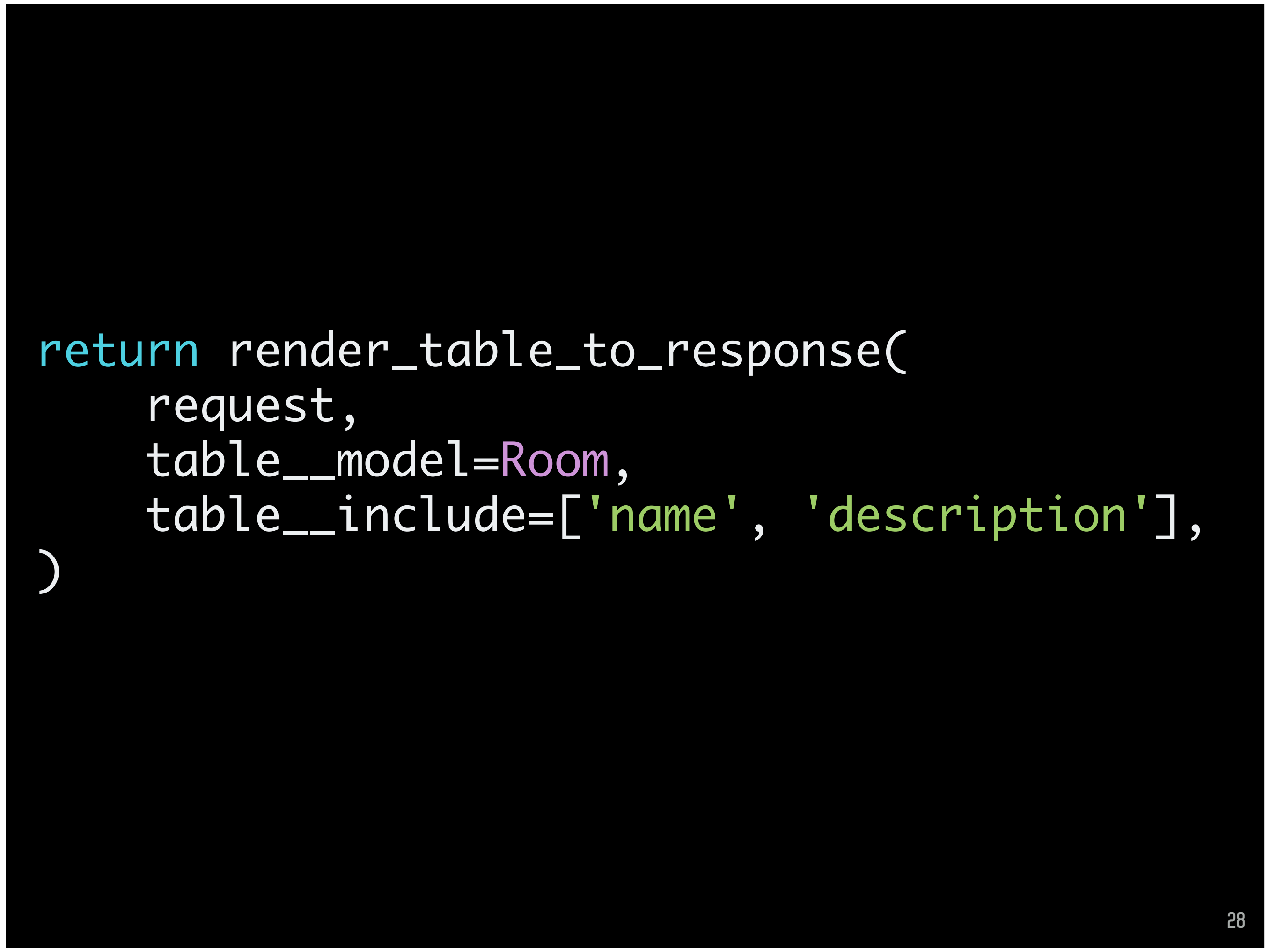
Some features you get for free are:
- sorting
- searching (via tri.query)
- pagination
- bulk editing

You can probably guess what the API looks like by now:



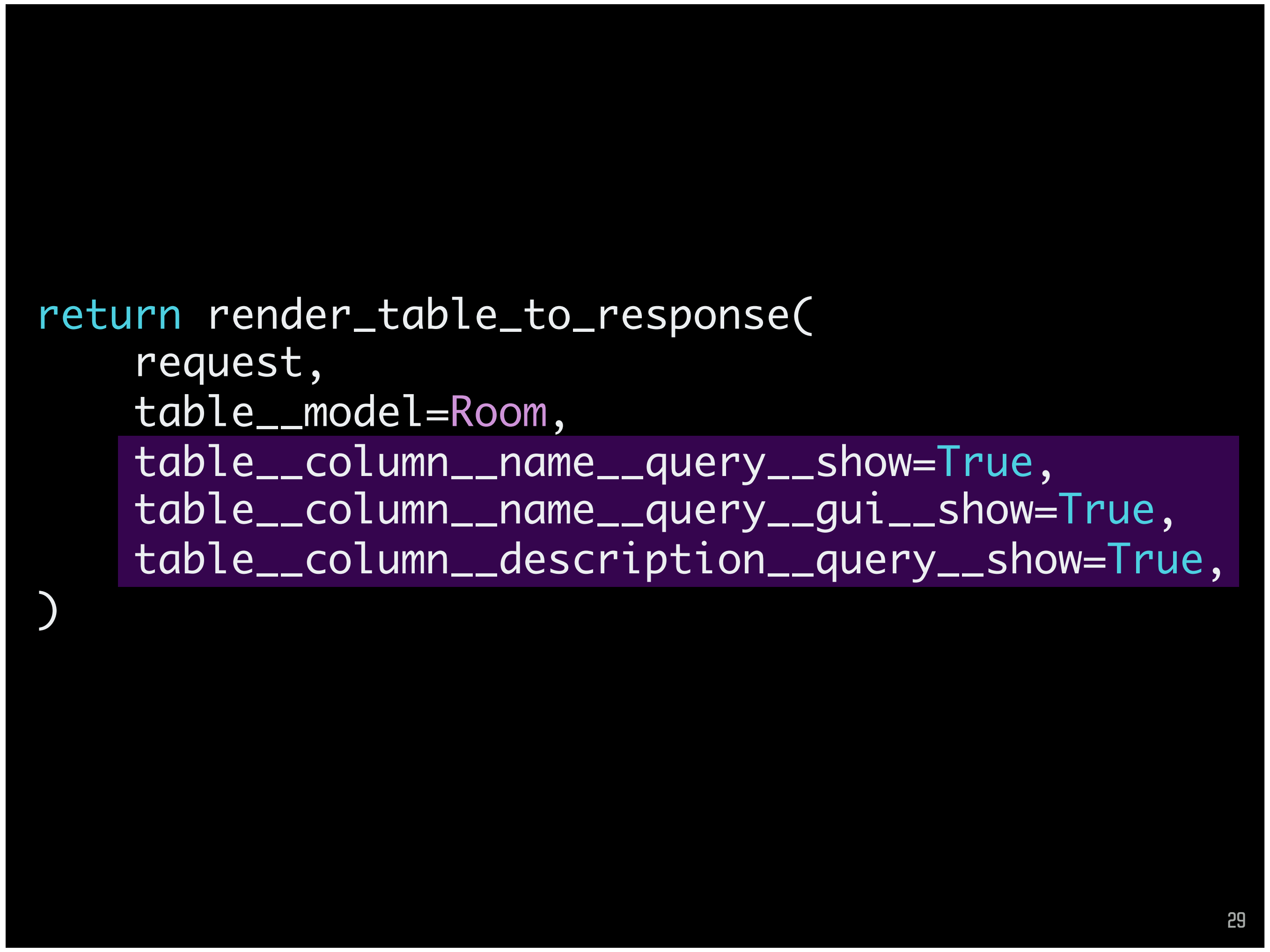
```
class MyTable(Table):
    name = Column()
    description = Column()


return render_table_to_response(
    request,
    table=MyTable(data=Room.objects.all().order_by('name'))
)
```

This gives you a table of all the rooms in your database with pagination and sortable columns. Now this is where we really start to see the power of all the stuff I've talked about so far. We have reflection code so the above can be simplified to not even use the declarative syntax:



```
return render_table_to_response(
    request,
    table__model=Room,
    table__include=['name', 'description'],
)
```

If you don't specify the columns with `include` here you'll get them all, dynamically inspected from the model class. tri.table builds on top of tri.query and tri.form so we can turn on filtering for specific fields by passing arguments down to tri.query and tri.form:



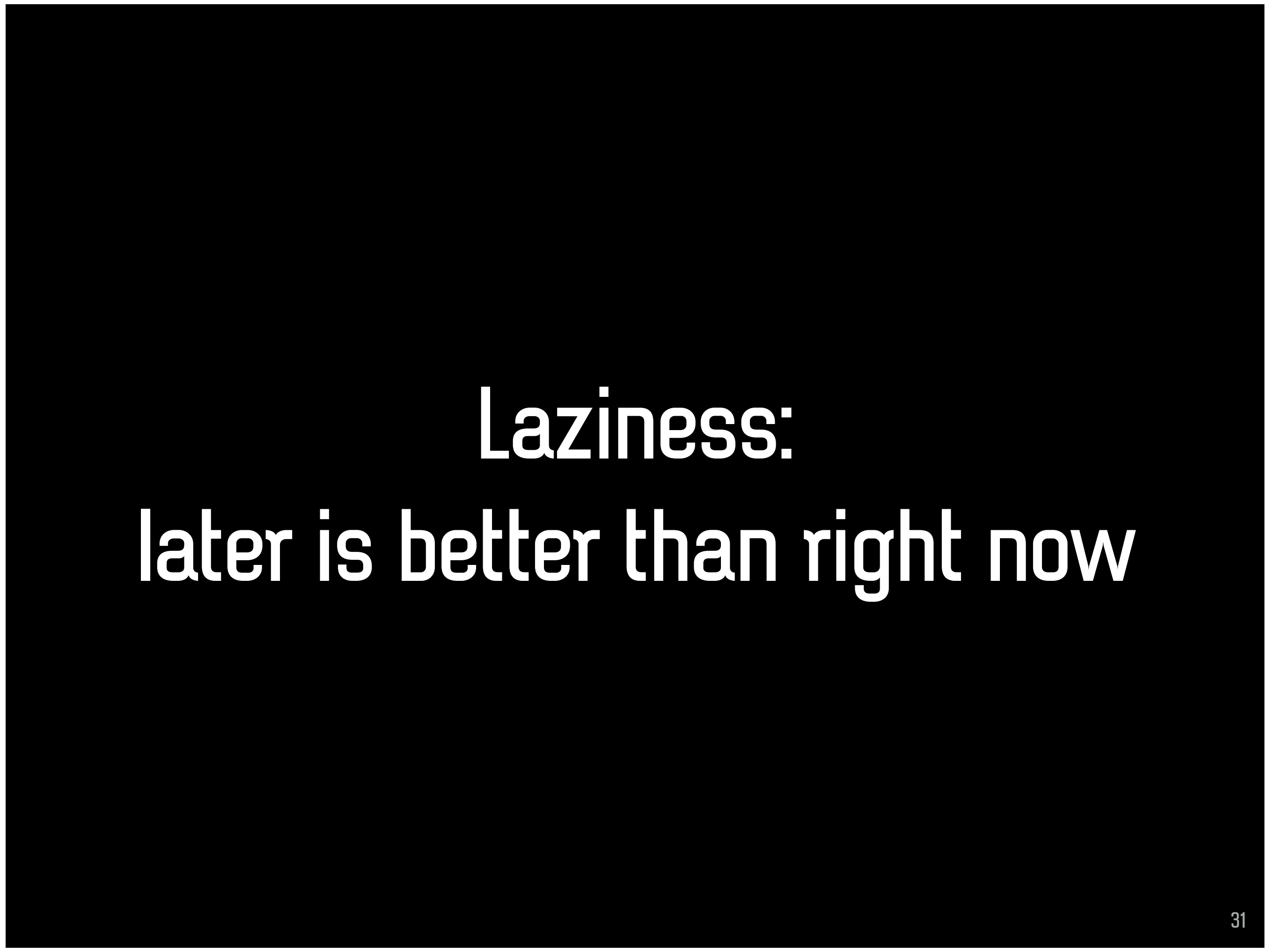
```
return render_table_to_response(
    request,
    table__model=Room,
    table__column__name__query__show=True,
    table__column__name__query__gui__show=True,
    table__column__description__query__show=True,
)
```

The "name" column will automatically get a tri.query Variable instance of the correct type, and there will be a form created from the query as a whole. As you can see even if there are many layers here, we can address all of them with all of their parameters using the double underscore syntax.

We can also enable the built in bulk edit functionality by passing an argument in the same way:

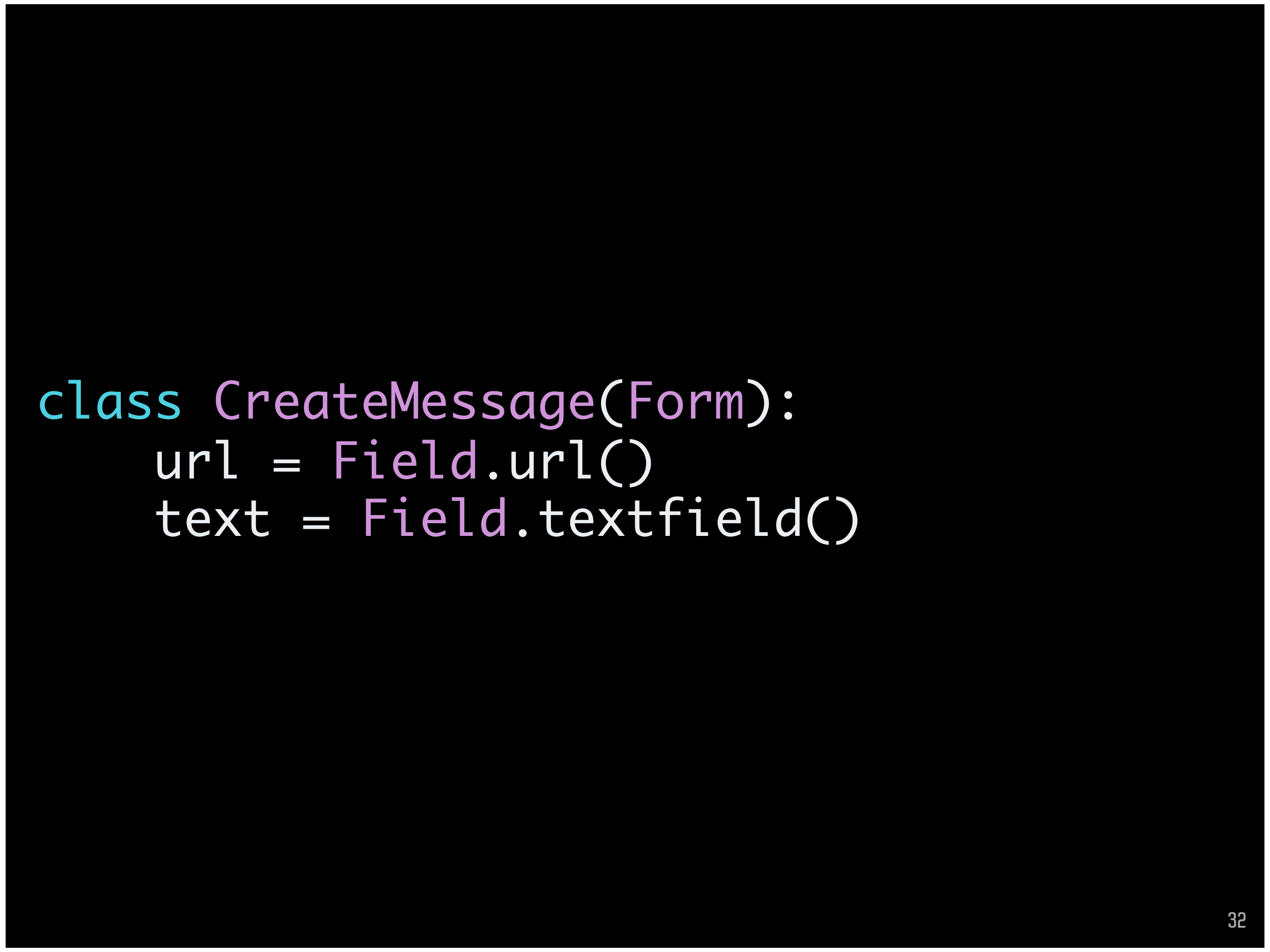
```
table__column__description__bulk__show=True
```

Bulk editing means you can multi-select a bunch of rows, or all of the rows, and set some fields to some specified values. We use this to publish reports to customers, requeue files for processing, comment on many things at once, etc.



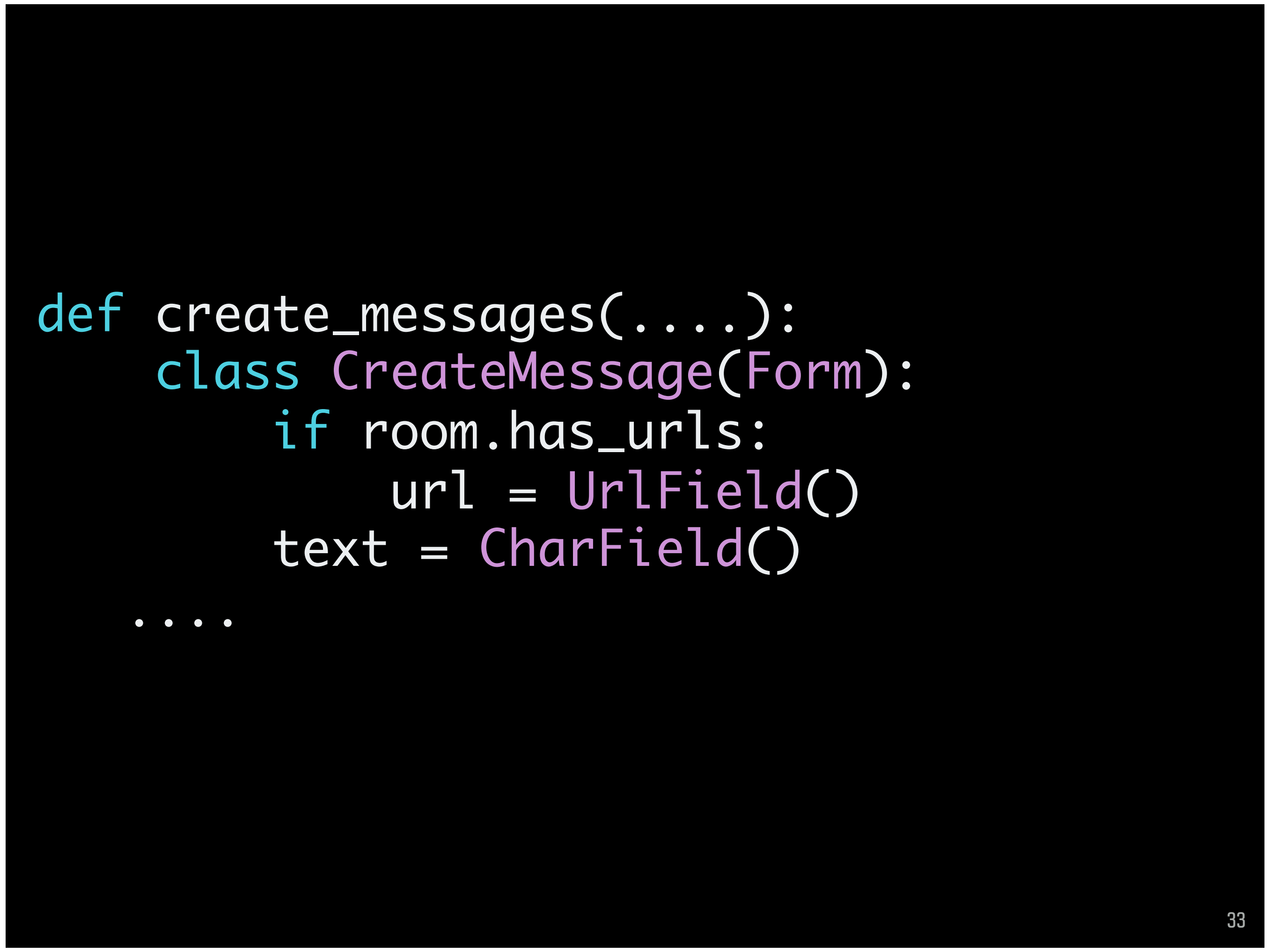
Laziness:
later is better than right now

So far I've talked about customization on different levels, from very small to big and how tri.form and tri.table gives you all the options to go to the level you need right now. But another aspect of customization is laziness and dynamism. We might want to not provide a parameter right now because the data will come later. For example in tri.table we might want to go over each row and do some processing after pagination but before the rendering, or in tri.form we'd like to base some parameter on the contents of the request GET parameters. Let's look at a simple example:



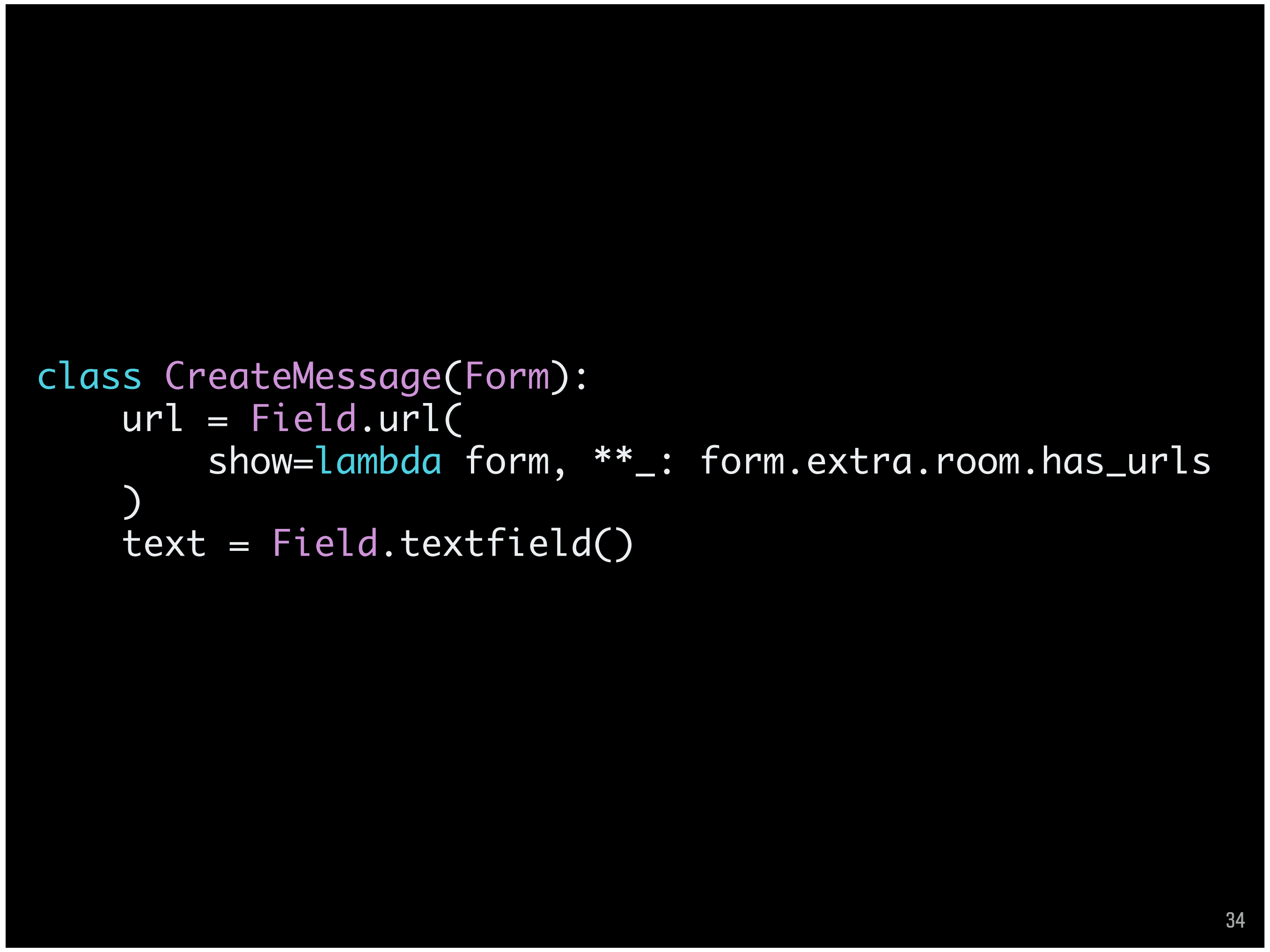
```
class CreateMessage(Form):
    url = Field.url()
    text = Field.textfield()
```

Let's say there are rooms where you can post messages with URLs. In Django forms we'd probably move the form into the view:



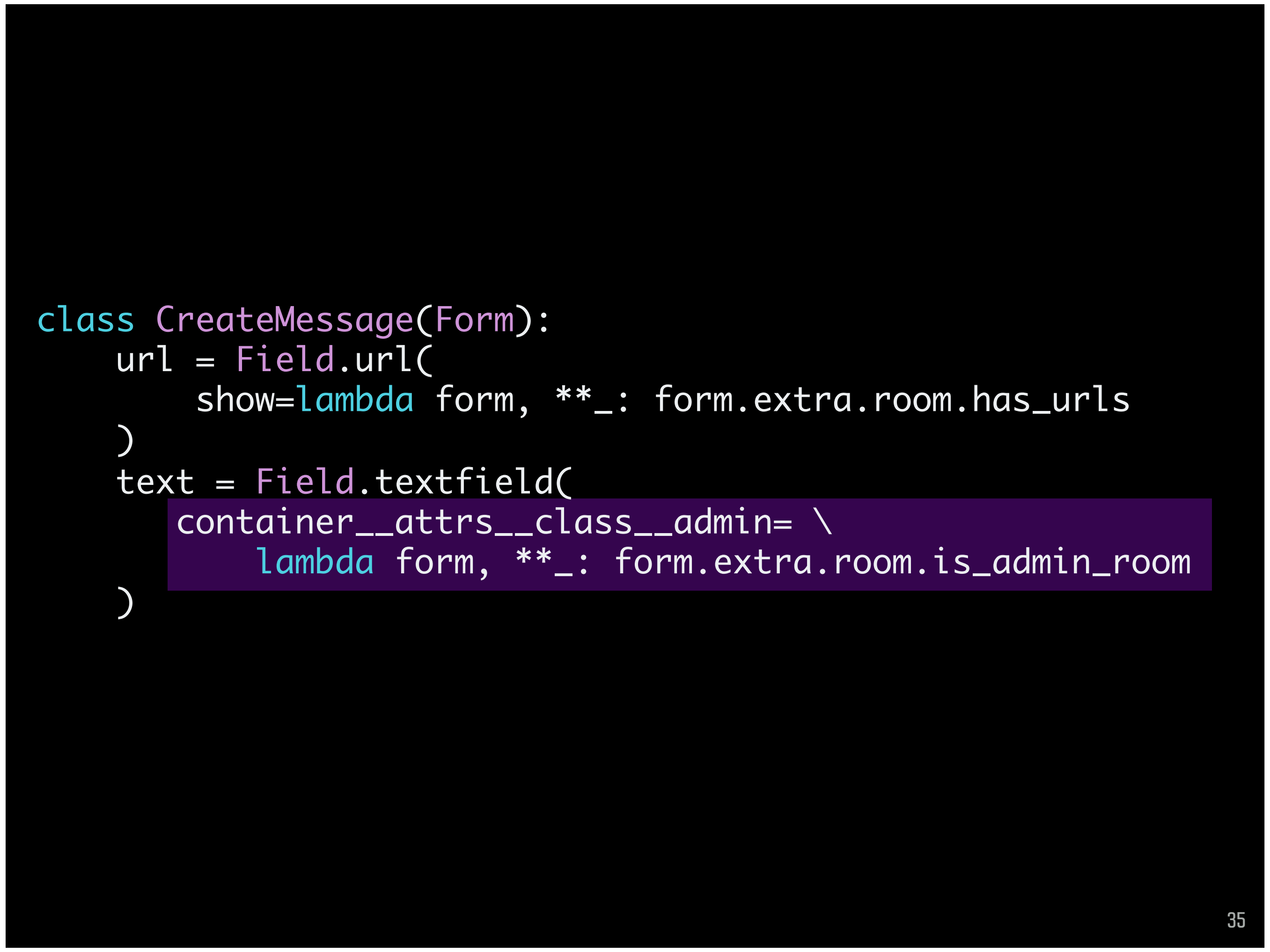
```
def create_messages(....):
    class CreateMessage(Form):
        if room.has_urls:
            url = UrlField()
        text = CharField()
   ....
```

In tri.form we can keep the form at the top level and modify it like this:



```
class CreateMessage(Form):
    url = Field.url(
        show=lambda form, **_: form.extra.room.has_urls
    )
    text = Field.textfield()
```

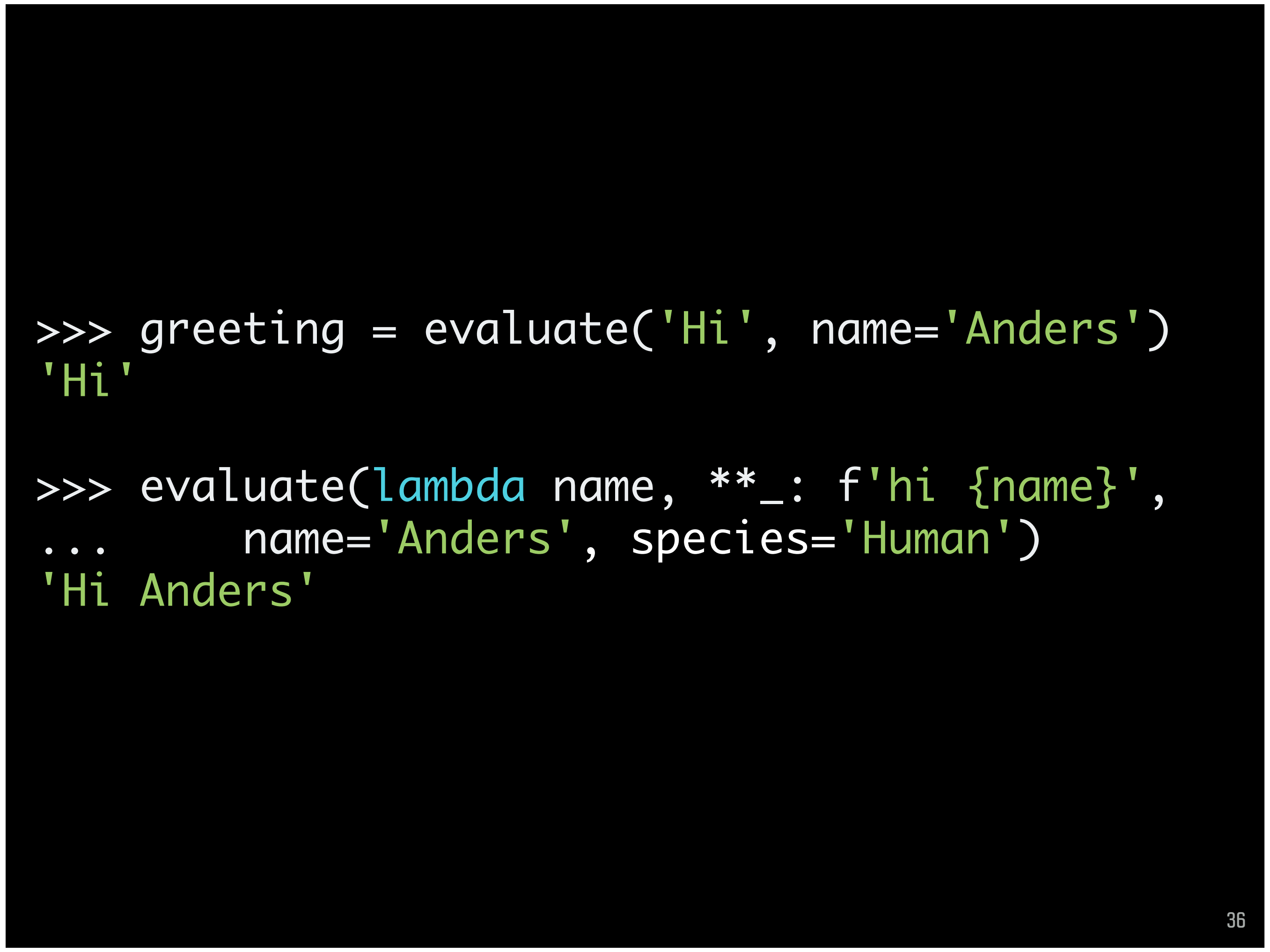
we can customize pretty much anything like this. Maybe we have special rooms for admins? Let's style the input field for them differently:



```
class CreateMessage(Form):
    url = Field.url(
        show=lambda form, **_: form.extra.room.has_urls
    )
    text = Field.textfield(
       container__attrs__class__admin= \
           lambda form, **_: form.extra.room.is_admin_room
    )
```

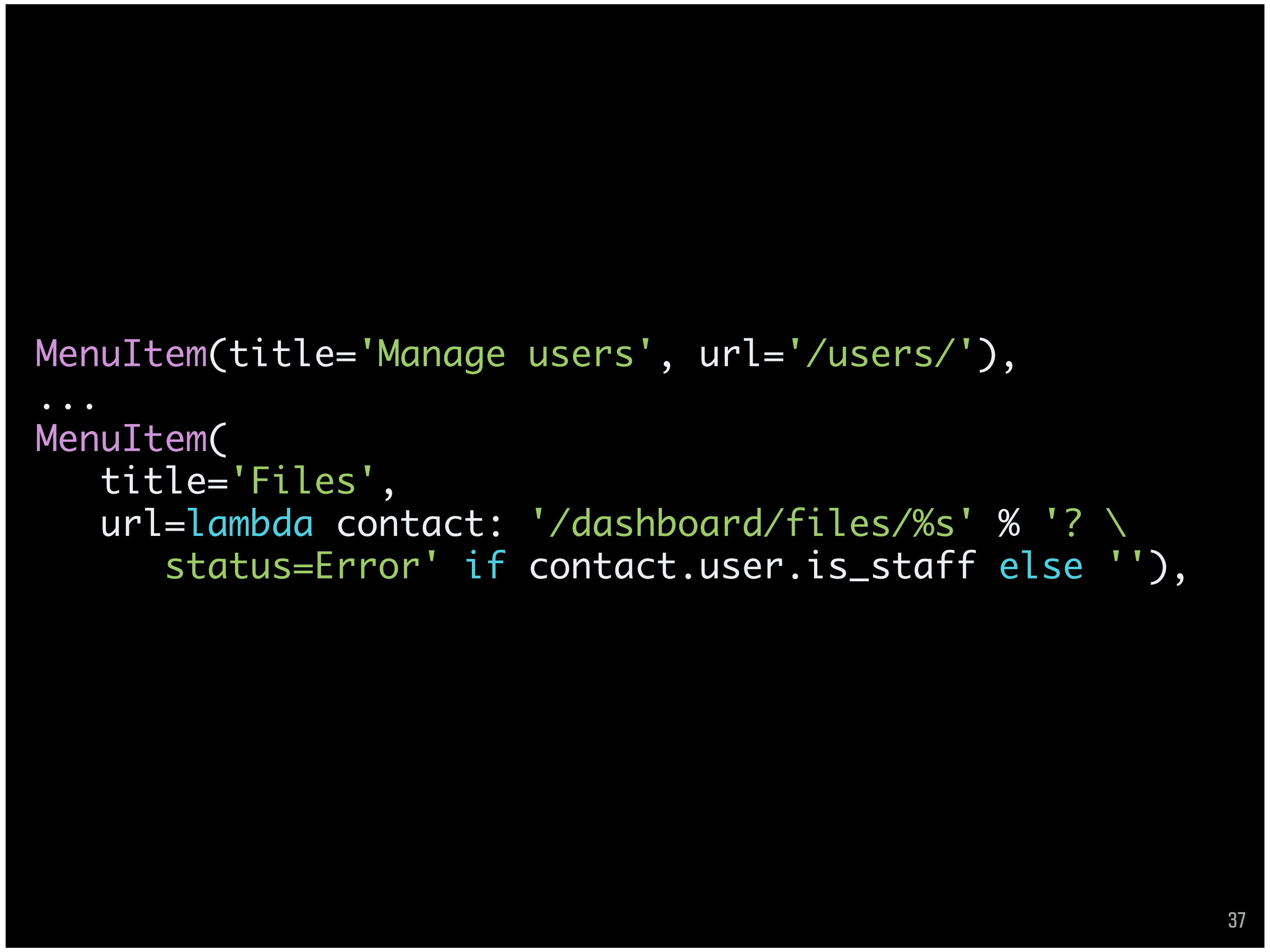
you can even supply templates for different items like this, choosing very late what template you need based on runtime data just before you need it.

The general pattern is that where you can put a hardcoded constant, you can also put any callable that matches the signature we define for these callbacks. This little magic is implemented with the evaluate() function in tri.declarative. It looks like this:



```
>>> greeting = evaluate('Hi', name='Anders')
'Hi'

>>> evaluate(lambda name, **_: f'hi {name}',
...     name='Anders', species='Human')
'Hi Anders'
```

We require you to match at least one argument, otherwise we'll get cases where something is called when it shouldn't be, and we highly recommend catching the rest and throwing them away with **_ which means you're future proof if new parameters are added. This little function by itself is pretty useful to make code simple for the simple case but still being able to handle the complex case. We use it to get a nice declarative definition of our menu system for example (this is actual code):



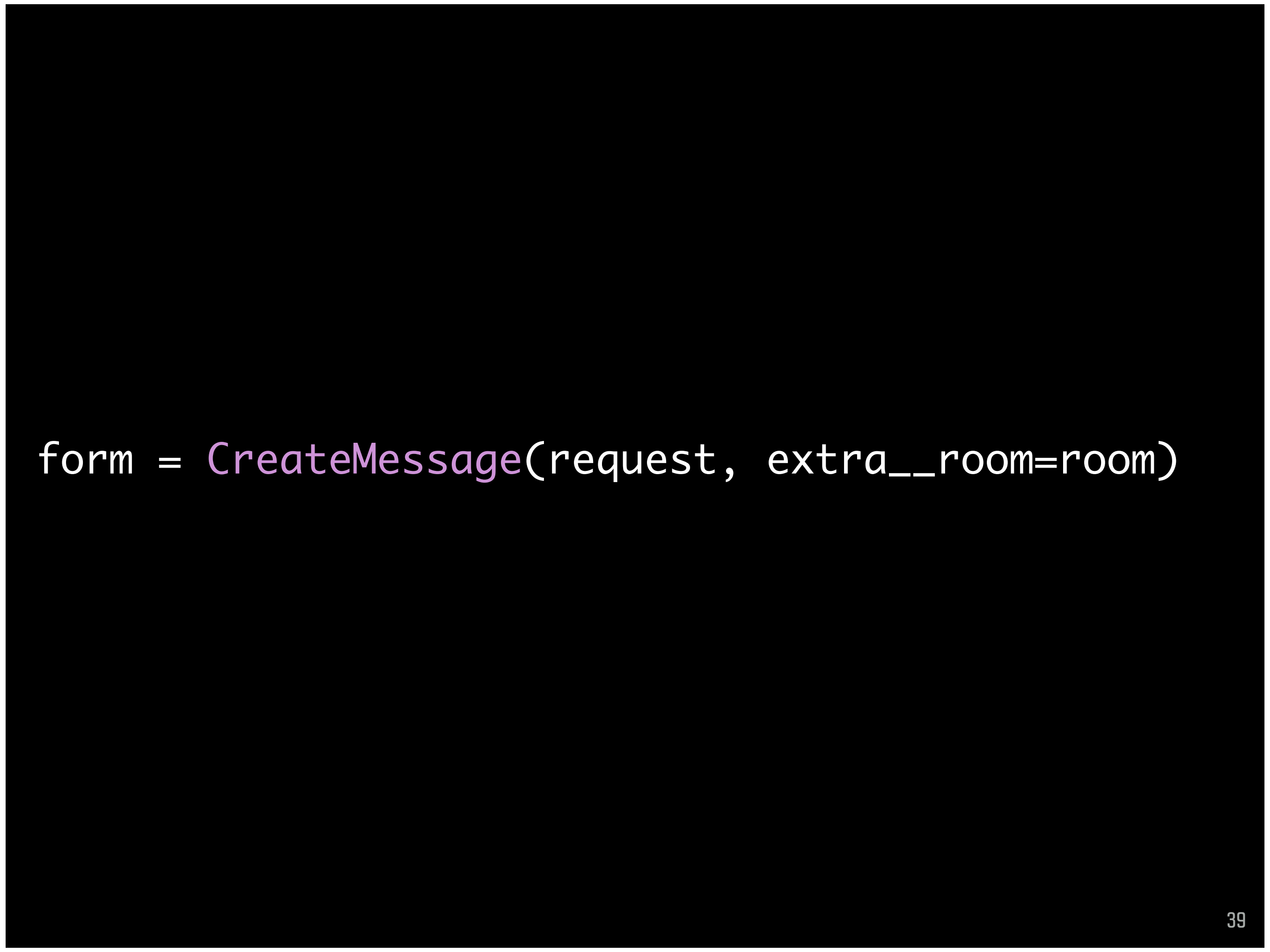
```
MenuItem(title='Manage users', url='/users/'),
...
MenuItem(
   title='Files',
   url=lambda contact: '/dashboard/files/%s' % '? \
      status=Error' if contact.user.is_staff else ''),
```

this way we don't need to put a lambda everywhere just because one or two places need it, greatly improving the readability of the code.

It's important to evaluate in layers too. First we evaluate the show parameter, and throw away things that shouldn't be shown. This means a lot of logic becomes much simpler because it's only called if the item is actually used.

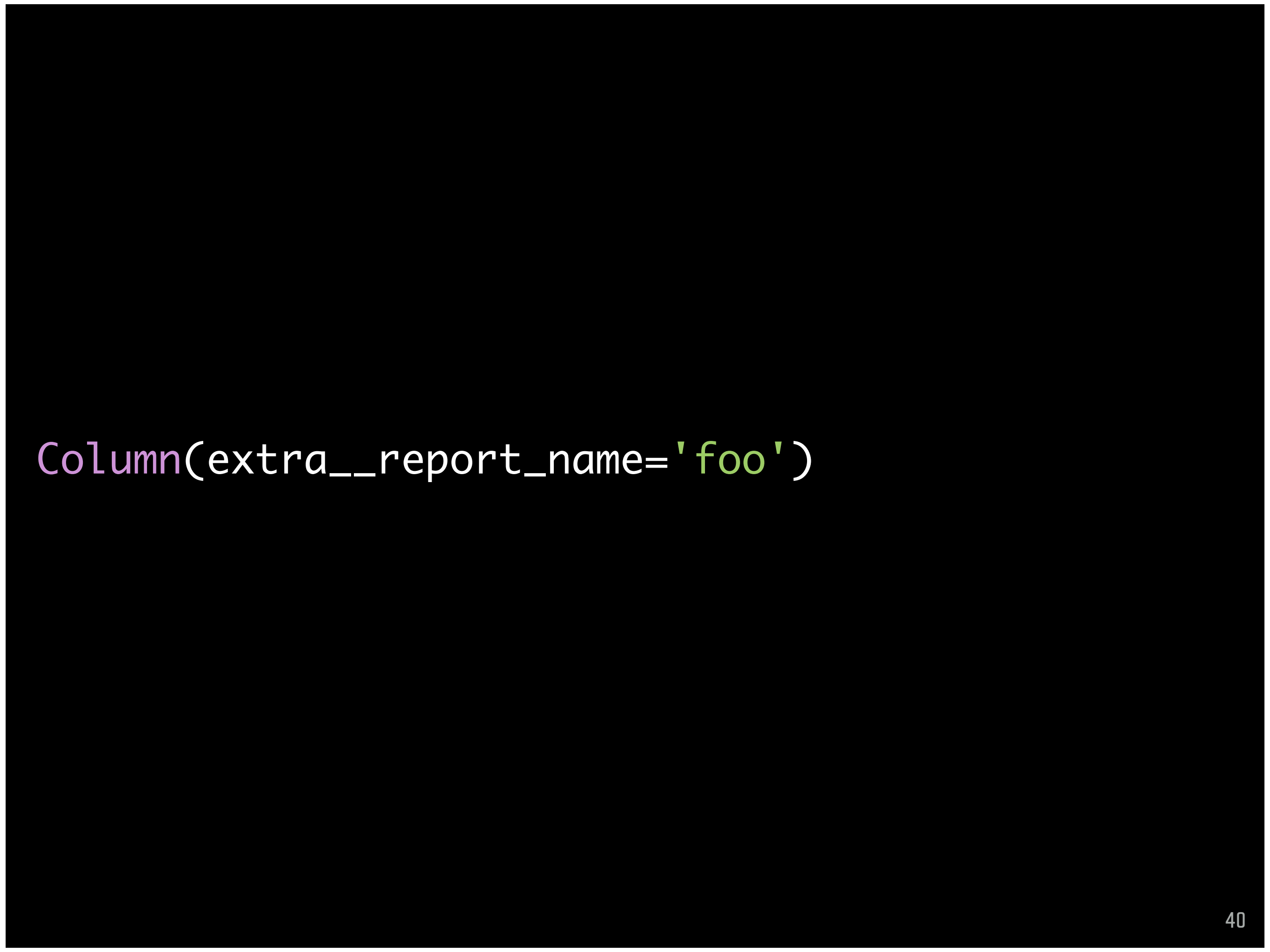
Escape hatch: extra

I've shown another feature on the previous slides that I glossed over. For all our classes in tri.table/query/form there is a member called "extra" that is a struct (an object-like dict) that we don't validate at all. This is an escape hatch if you need to signal something to yourself later and the lib itself doesn't supply that signal path. So when creating a message we already know which room we should post in, but we don't have that in the Form itself. We just put it in extra like so:



```
form = CreateMessage(request, extra__room=room)
```

This extra member is on both the Form and on the Fields, and the corresponding classes in tri.table and tri.query have the same thing. This is analogous to UserData if you ever did Win32 programming like I started my career with. This little escape hatch lets you bolt on lots of extra functionality without having to subclass. This is especially nice for simple one-off cases, which are the majority of the cases where we need this type of signal path.



```python
Column(extra__report_name='foo')
```

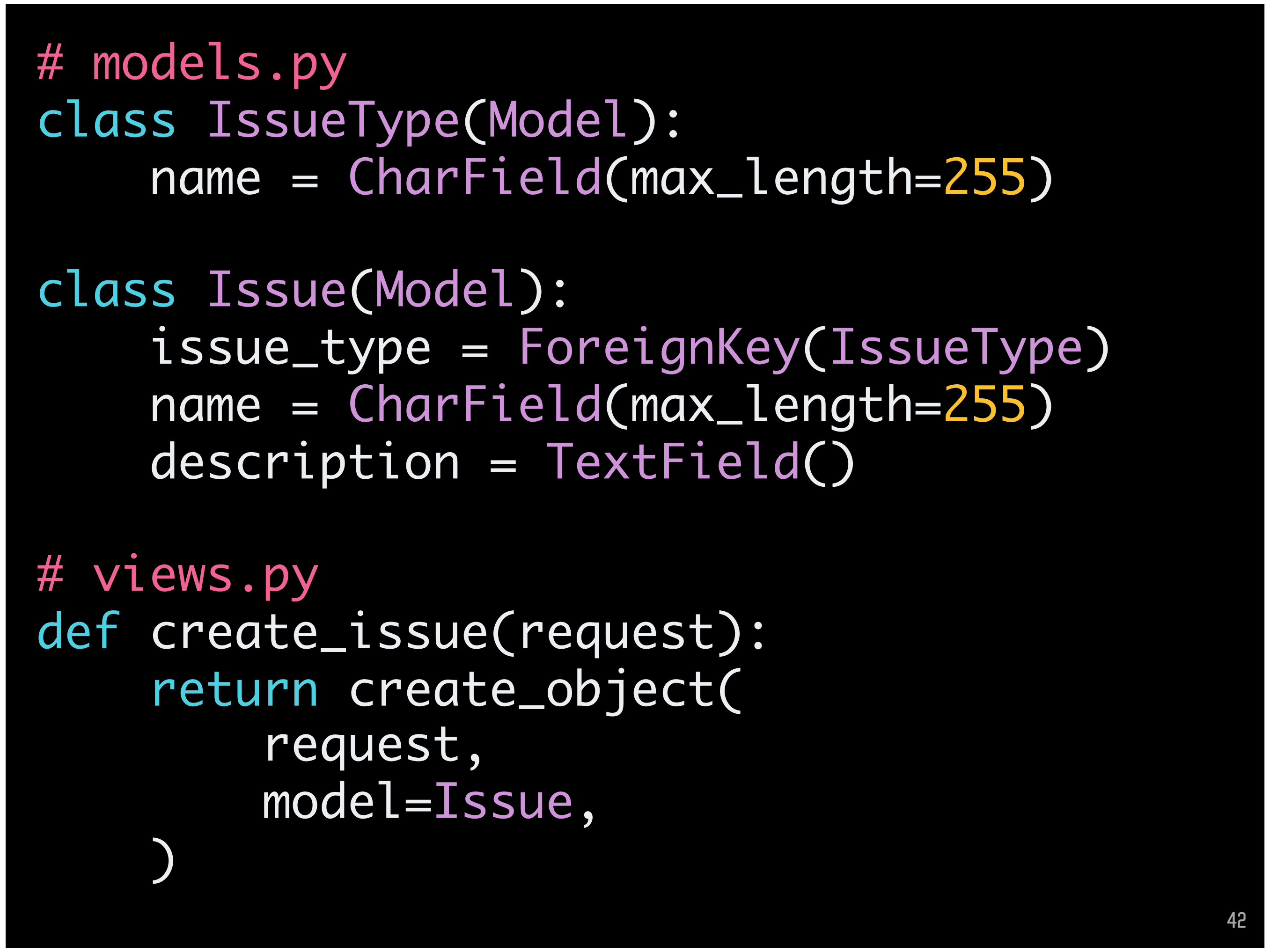
The `extra` escape hatch is something we use extensively to build extra functionality.

For example, with `extra` plus that we have programmatic access to the full definition of tables, we've built a feature to export tables to Excel, CSV and PDF and we can schedule this to run on the back end.

Everything has a name

Another thing we've discovered with these libraries is that it's great if everything has a name. It's very easy to just have a list of un-named things, but that approach locks you out of the possibility to navigate to those objects. Accessing by index is super fragile but by name is pretty robust.

Also, because we have names for everything we can implement some really cool features like automatically produced AJAX endpoints for autocomplete. An example can look like this simple issue tracker:



```
# models.py
class IssueType(Model):
    name = CharField(max_length=255)

class Issue(Model):
    issue_type = ForeignKey(IssueType)
    name = CharField(max_length=255)
    description = TextField()

# views.py
def create_issue(request):
    return create_object(
        request,
        model=Issue,
    )
```

and that's it. The create_object function handles an ajax endpoint for the issue type.

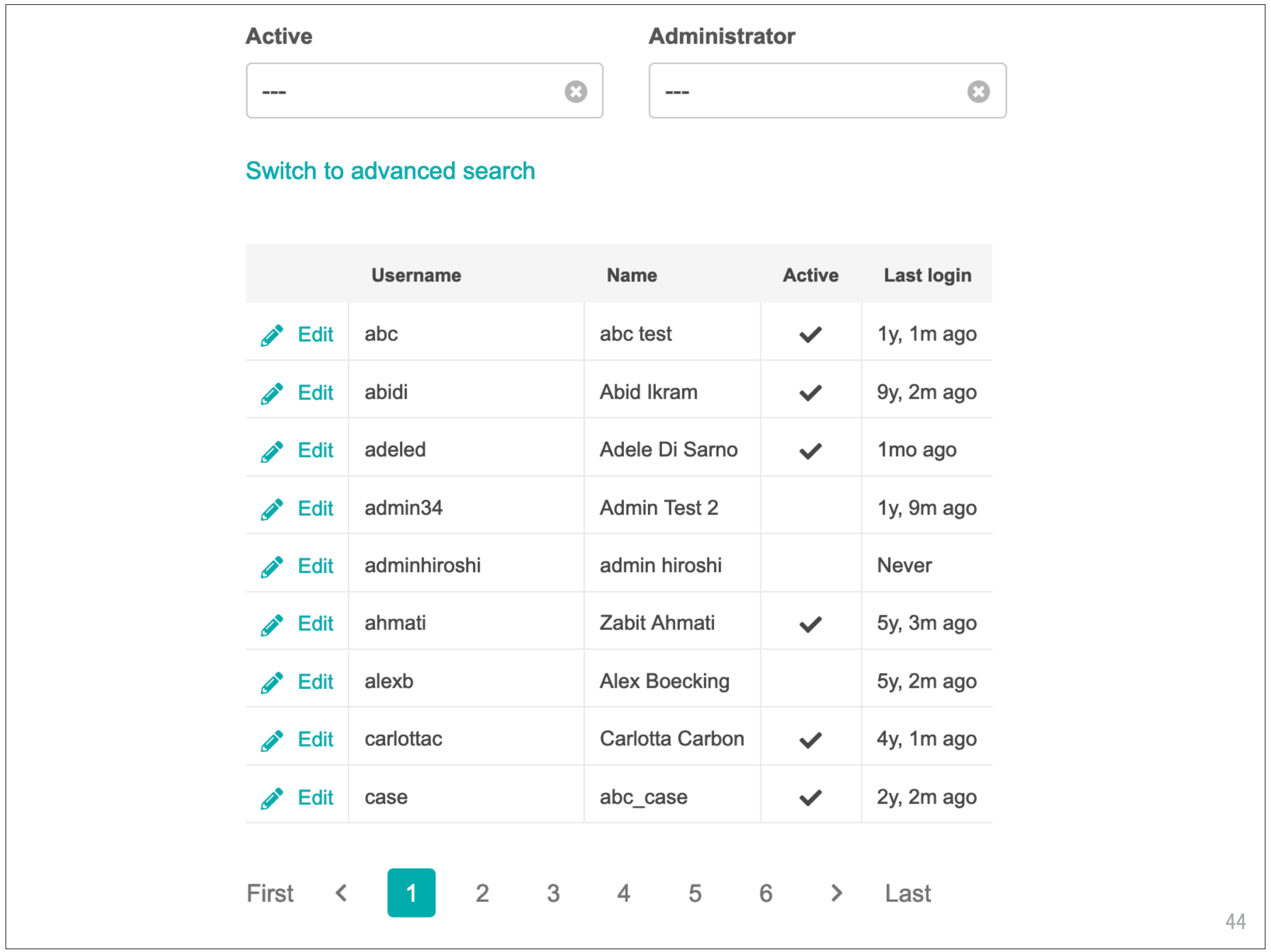
What we really like about this approach is that we get one view function for the entire view, including ajax. But even better is that when we have permission checks the ajax endpoint has to pass through the exact same permissions code, making it much easier to audit.

Time to breathe!

Now that was a pretty crazy whirlwind tour of our libraries tri.form, tri.query and tri.table. I've glossed over lots and lots of features and niceties obviously. Now you might be asking how do we actually do all that stuff? Can I also get these multi layered APIs with full flexibility? Of course!

First I'd like to give this philosophy a name: Transparent APIs. Normally you have opaque APIs where if you have a function A and it calls some other function B, now A has hidden all parameters of B except those it explicitly exposes via its own parameters. But what we're talking about here is a way to layer functions and still expose all those things. That's why we call them transparent APIs, because you can see "through" them.

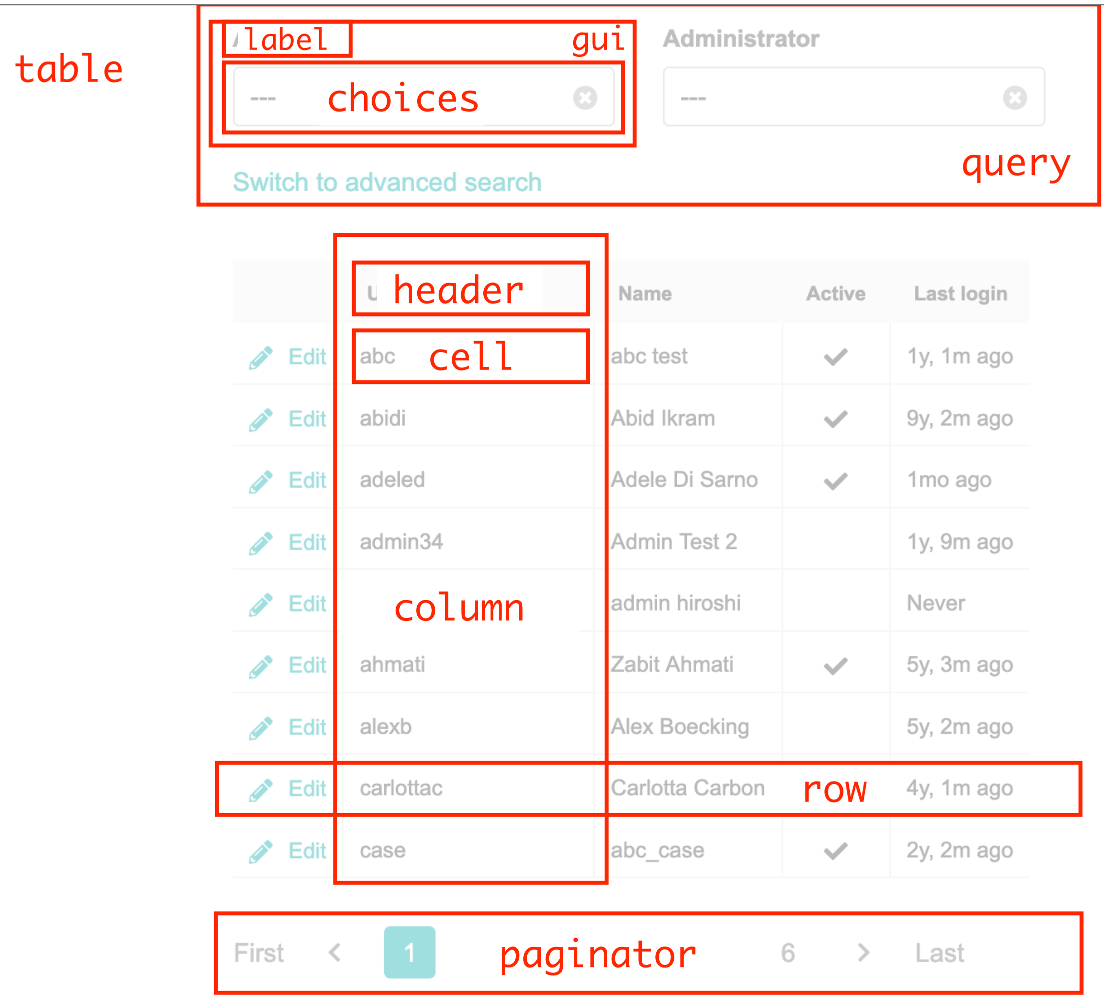
First we'll talk about nested namespaces.


Active

Administrator

Switch to advanced search

	Username	Name	Active	Last login
Edit	abc	abc test	✓	1y, 1m ago
Edit	abidi	Abid Ikram	✓	9y, 2m ago
Edit	adeled	Adele Di Sarno	✓	1mo ago
Edit	admin34	Admin Test 2		1y, 9m ago
Edit	adminhiroshi	admin hiroshi		Never
Edit	ahmati	Zabit Ahmati	✓	5y, 3m ago
Edit	alexb	Alex Boecking		5y, 2m ago
Edit	carlottac	Carlotta Carbon	✓	4y, 1m ago
Edit	case	abc_case	✓	2y, 2m ago

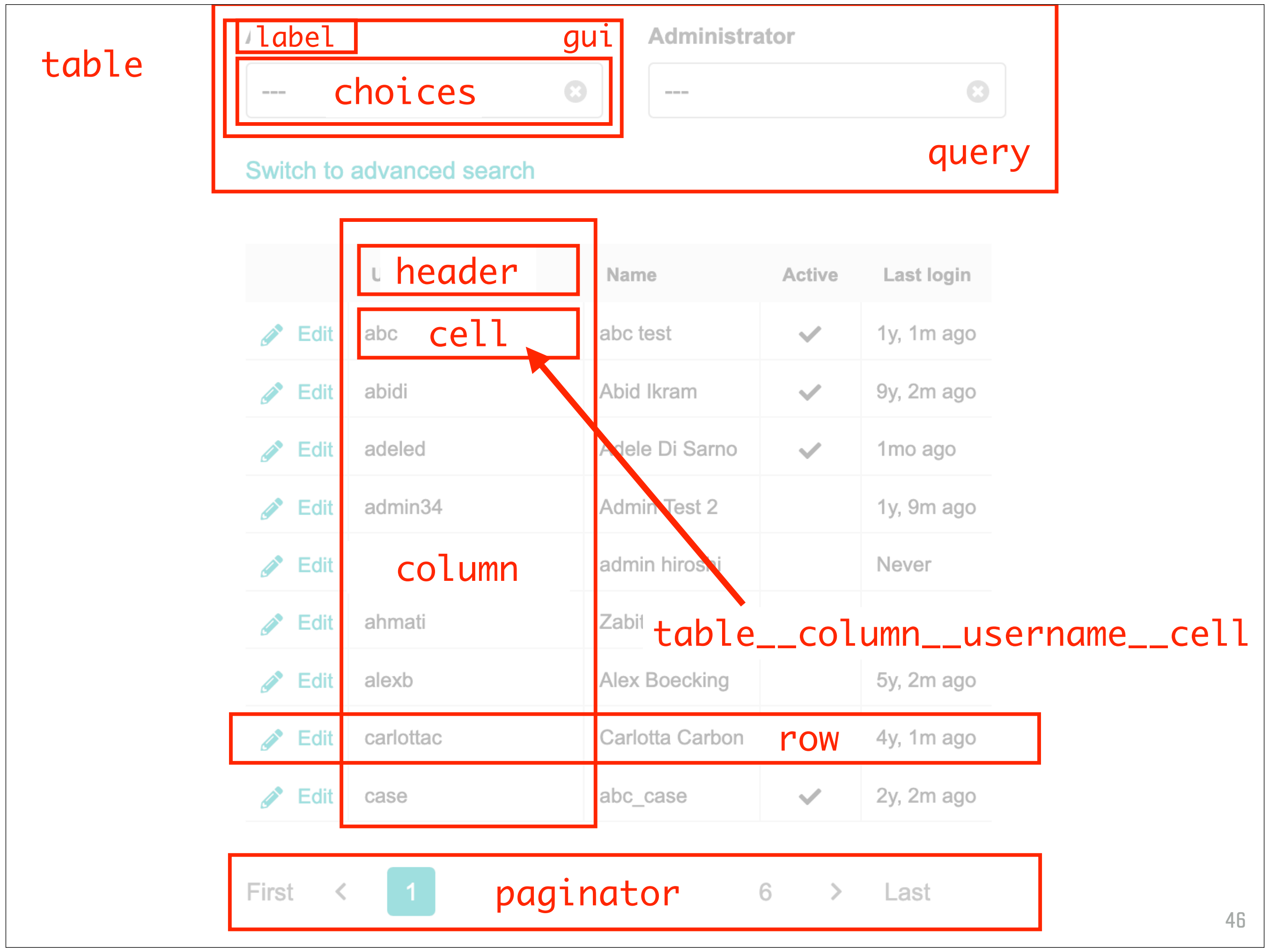


Administrator

Switch to advanced search

		Name	Active	Last login
Edit	abc	abc test	✓	1y, 1m ago
Edit	abidi	Abid Ikram	✓	9y, 2m ago
Edit	adeled	Adele Di Sarno	✓	1mo ago
Edit	admin34	Admin Test 2		1y, 9m ago
Edit		admin hiroshi		Never
Edit	ahmati	Zabit Ahmati	✓	5y, 3m ago
Edit	alexb	Alex Boecking		5y, 2m ago
Edit	carlottac	Carlotta Carbon		4y, 1m ago
Edit	case	abc_case	✓	2y, 2m ago

First < 1 6 > Last



		Name	Active	Last login
Edit	abc	abc test	✓	1y, 1m ago
Edit	abidi	Abid Ikram	✓	9y, 2m ago
Edit	adeled	Adele Di Sarno	✓	1mo ago
Edit	admin34	Admin Test 2		1y, 9m ago
Edit		admin hiroshi		Never
Edit	ahmati	Zabit		
Edit	alexb	Alex Boecking		5y, 2m ago
Edit	carlottac	Carlotta Carbon		4y, 1m ago
Edit	case	abc_case	✓	2y, 2m ago

Now I'll explain the secret sauce that powers all this:

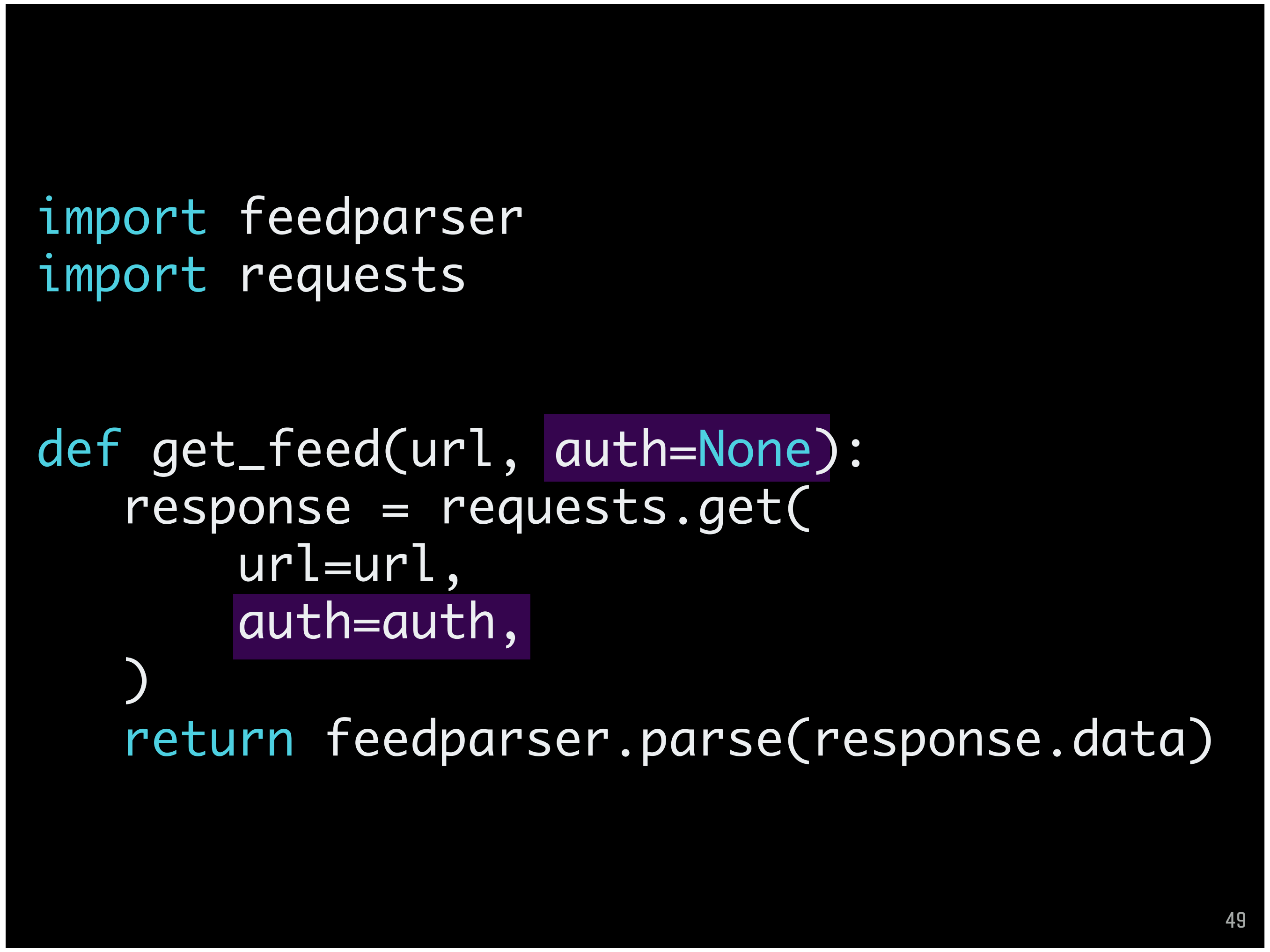
@dispatch and Namespace

I'm going to go through a simple example where we might want to use the @dispatch decorator to get a transparent API. First we'll start with a traditional implementation and then see how it can be improved, and why we'd do this.

So let's say we want to write some RSS aggregator. So we'd start out with something simple that uses requests to get the data and feedparser to parse it:

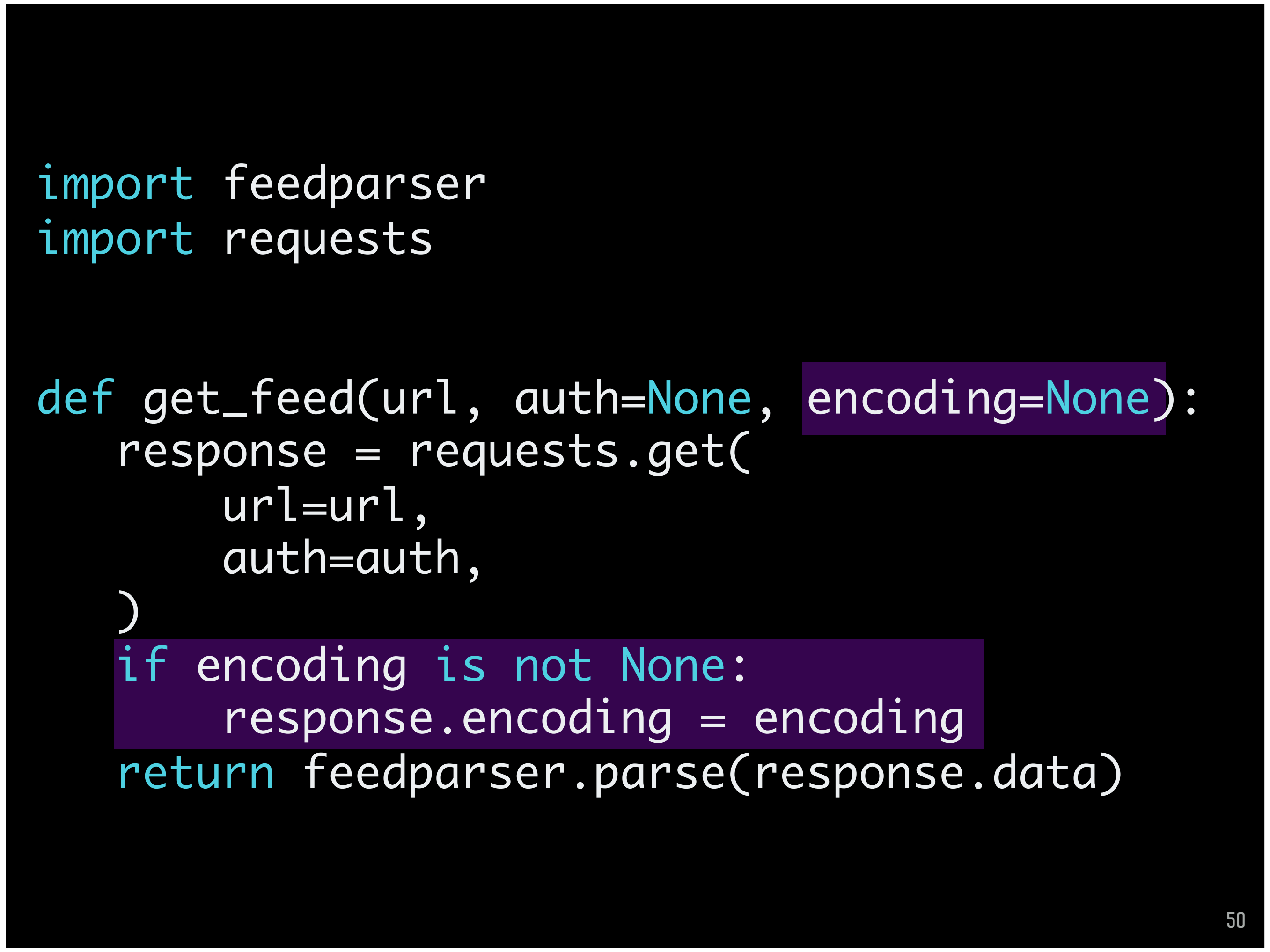
```python
import feedparser
import requests


def get_feed(url):
   response = requests.get(url=url)
   return feedparser.parse(response.data)
```

Pretty trivial. Ok, but how to we handle edge cases? Maybe the first thing we'd hit is password protected RSS feeds. We could solve this by adding an auth parameter that we send to requests:



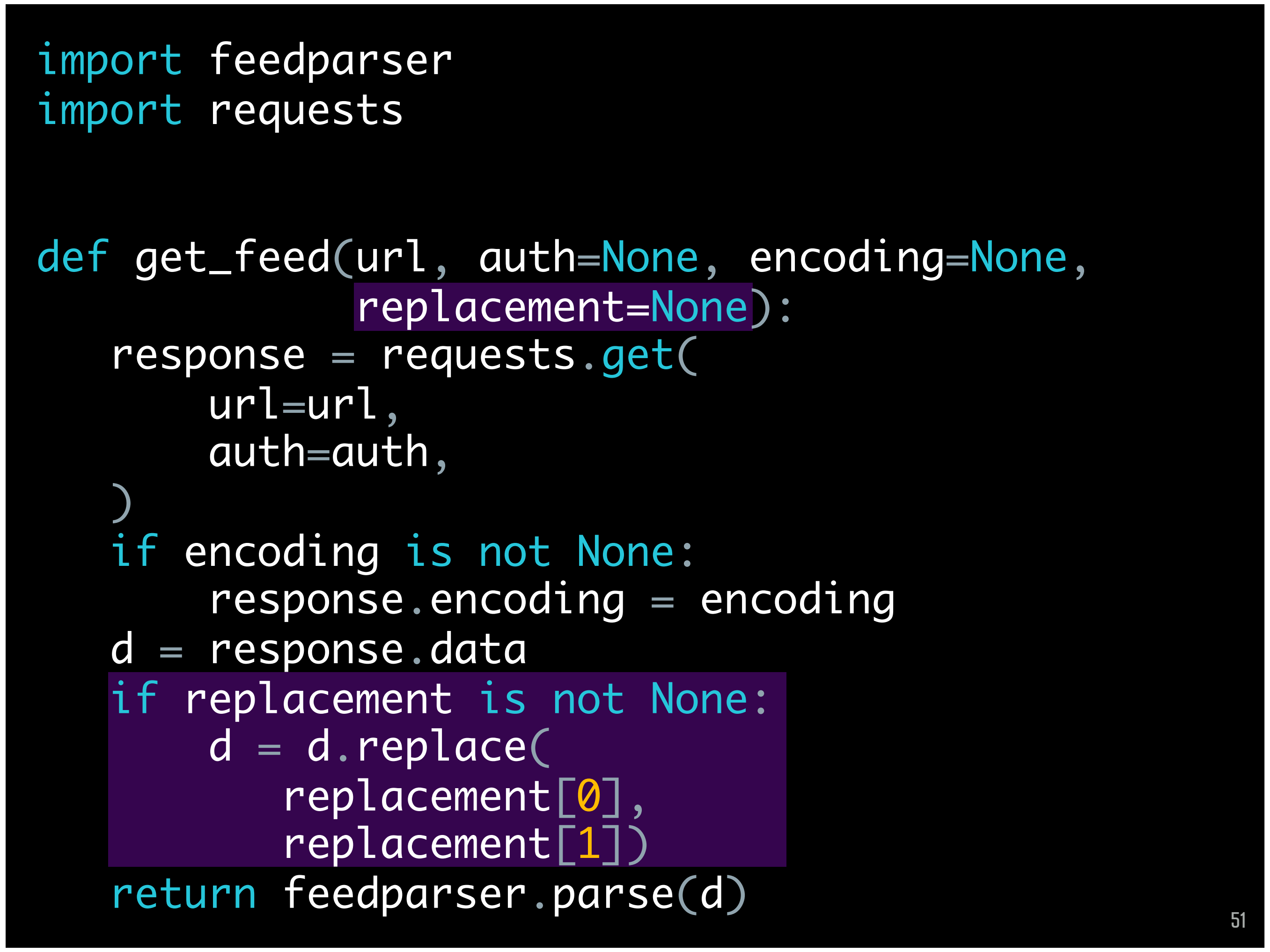
```python
import feedparser
import requests


def get_feed(url, auth=None):
   response = requests.get(
       url=url,
       auth=auth,
   )
   return feedparser.parse(response.data)
```

Seems pretty ok. Now we hit an RSS feed that generates the data in latin-1 encoding but the webserver is misconfigured so it says it's utf8. Ok, so we could handle this by another parameter 'encoding':



```python
import feedparser
import requests


def get_feed(url, auth=None, encoding=None):
    response = requests.get(
        url=url,
        auth=auth,
    )
    if encoding is not None:
        response.encoding = encoding
    return feedparser.parse(response.data)
```

Then there's a feed that generates corrupt XML that feedparser chokes on. So we'll have to add something for that:

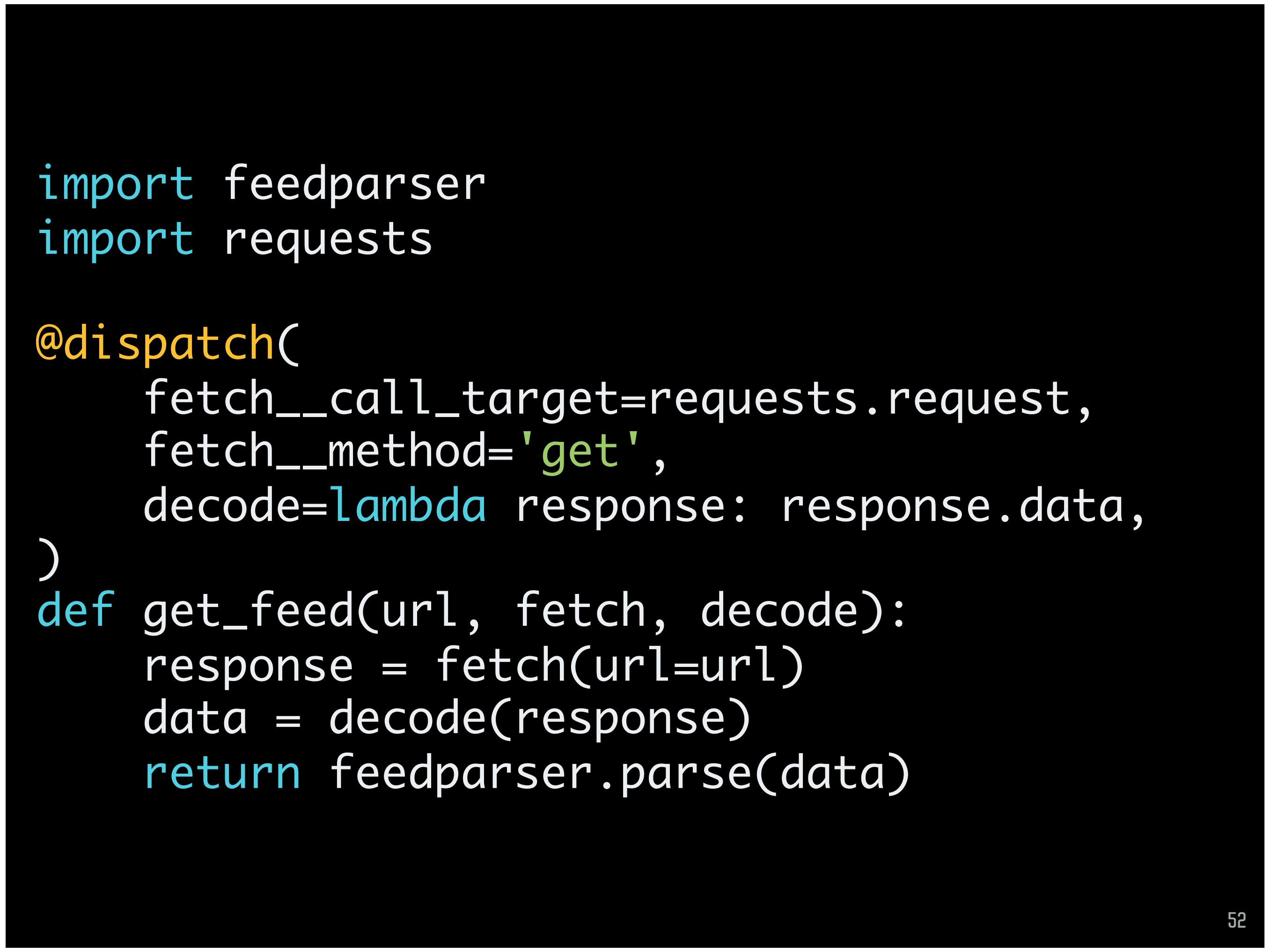


```
import feedparser
import requests


def get_feed(url, auth=None, encoding=None,
             replacement=None):
    response = requests.get(
        url=url,
        auth=auth,
    )
    if encoding is not None:
        response.encoding = encoding
    d = response.data
    if replacement is not None:
        d = d.replace(
            replacement[0],
            replacement[1])
    return feedparser.parse(d)
```

Then we find a feed that has TWO errors, so replacement has to be changed to replacementS and a for loop introduced. Then we find some other feed where we need to give it a special user agent string and we introduce an argument for that...

I think you see where I'm going with this... every little small edge cases cause lots of code to be added in the implementation of get_feed. We'd prefer the edge cases to be isolated to the CALLS of get_feed and not pollute get_feed with their single use special cases.

Let's look at how we'd solve this with transparent APIs.

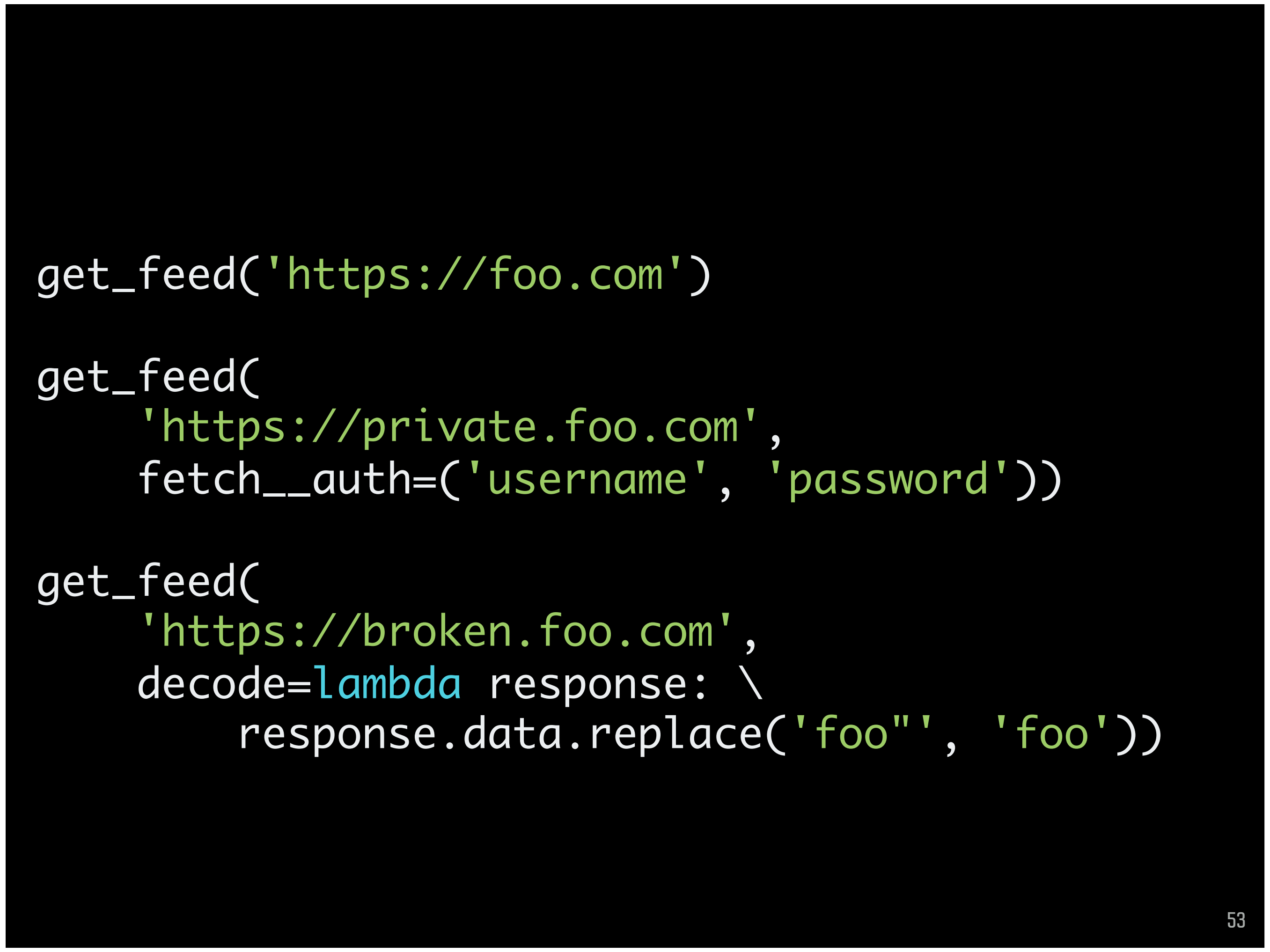


```
import feedparser
import requests

@dispatch(
    fetch__call_target=requests.request,
    fetch__method='get',
    decode=lambda response: response.data,
)
def get_feed(url, fetch, decode):
    response = fetch(url=url)
    data = decode(response)
    return feedparser.parse(data)
```

The @dispatch decorator here handles the double underscore syntax I've talked before, and it also takes a list of defaults. What will happen is that the get_feed function will get a Namespace object passed as the `fetch` parameter. This acts pretty similar to a partial evaluation, as you can see we call `fetch` as if it's a function. Really the @dispatch decorator here is mostly just a thin wrapper around the Namespace constructor.

We differ here from a normal partial application like functools.partials by explicitly naming the target function `call_target`, and we can override it. Think of it more as a default value for the partial target than us specifying the target.

Now if we look at some calls of get_feed:



```
get_feed('https://foo.com')

get_feed(
    'https://private.foo.com',
    fetch__auth=('username', 'password'))

get_feed(
    'https://broken.foo.com',
    decode=lambda response: \
        response.data.replace('foo"', 'foo'))
```

You can see that the simple case is just as simple as before, but the complex edge cases are very easy to handle and they don't pollute the code of get_feed.

This would be even cleaner if the request function of requests had been written with transparent APIs in mind and we could pass it the decode parameter as a lambda.

Summary

I hope I've convinced you that we can use the transparent API style to create highly flexible APIs that can scale to handle edge cases while still keeping the code simple.

I also hope I've piqued your interest in using tri.table, tri.query, tri.form and tri.declarative for use in your own products. For us it has been a huge boon to our development speed. We even write full html tables with sorting and bulk editing for single use pages that are only valid for a few days or even hours.

Thanks for listening!

Find our libs at:
github.com/TriOptima

Anders Hovmöller
github.com/boxed