Unternehmenstauglich

Python ist in den letzten Jahren in den Olymp der beliebtesten Programmiersprachen aufgestiegen. Schnelles Prototyping, einfache Integration mit mächtigen Bibliotheken wie TensorFlow oder NumPy und die unkomplizierte Erlernbarkeit haben maßgeblich dazu beigetragen. Im Unternehmenseinsatz ist mit dem Prototyp in Python häufig aber auch Schluss. Dabei fehlt für den produktiven Einsatz meistens nicht viel. Wir zeigen, wie sich mit dem Django Webframework Python-Anwendungen mit wenig Aufwand für den Unternehmenseinsatz aufpeppen lassen.

Django blickt auf eine mittlerweile mehr als 15-jährige Entwicklungsgeschichte zurück, seit das Web Framework „for perfectionists with deadlines“ - so der eigene Slogan - erstmals veröffentlicht wurde. Tatsächlich nimmt Django Entwicklern eine Menge lästige Routinearbeit ab, indem es konsequent den „Don’t Repeat Yourself“ (DRY) Ansatz verfolgt. Wie das durch die Verwendung eines einmal deklarierten Models funktioniert, erklärt der Kasten in aller Kürze.

An dieser Stelle schauen wir auch weniger auf den grundsätzlichen Aufbau einer Django-Anwendung. Einführende Tutorials finden sich hierzu entweder online auf der Projektseite [1] oder entsprechenden Büchern, z. B. [2]. Vielmehr stellen wir einige hilfreiche Django-Packages vor, die typische Aufgaben im Umfeld einer Unternehmensanwendung lösen und so beliebige Anwendungen mit wenig Aufwand unternehmenstauglich machen. Als Beispiel dient uns eine einfache Aufgabenverwaltung, deren Quellcode unter [3] verfügbar ist.

Torwächter

Eine grundlegende Anforderung für Unternehmensanwendungen besteht in der Regel in der eindeutigen Authentifizierung und der Berechtigungsprüfung von Anwendern. Django bringt eine eigene vollständige Benutzer- und Berechtigungsverwaltung mit. Jedoch sind Anwender nur selten begeistert, wenn ein weiterer Benutzername und Passwort gemerkt und möglicherweise auch noch regelmäßig aktualisiert werden muss. Glücklicherweise bietet Django mit dem Package *django-python3-ldap* [4] eine einfache Möglichkeit, die Django-Anwendung an ein bestehendes Active Directory im Unternehmen anzubinden. Die erforderlichen Verbindungsdaten sind schnell entsprechend Online-Dokumentation konfiguriert. In der *settings.py* wird unter AUTHENTICATION\_BACKENDS das Package als „django\_python3\_ldap.auth.LDAPBackend“ hinzugefügt und schon gehen die Login-Daten an das Active Directory zur Bestätigung. Netter Nebeneffekt: Die Django-Benutzerverwaltung läuft weiter und kann zusätzlich zur Einrichtung lokaler Benutzer verwendet werden.

Berechtigungen werden zu jedem Benutzer im Admin-Bereich von Django gepflegt, das auf einem gestarteten Webserver im Pfad „/admin“ erreichbar ist, bei gestartetem Entwicklungsserver unter: <http://localhost:8000/admin>

Die Berechtigungen orientieren sich am Model und ermöglichen für jedes Objekt die Unterteilung nach dem CRUD-Schema: Create, Read, Update, Delete. Jedes definierte Model erscheint ohne weiteres Zutun im Admin-Bereich und kann dort in der Web-Oberfläche gepflegt werden. Da je nach Umfang des Projekts und des Datenmodells sehr viele Einzelberechtigungen möglich sind, ist die Verwendung von „Gruppen“ empfehlenswert. Die erforderlichen Berechtigungen für eine bestimmte Anwenderrolle können in einer Gruppe zusammengefasst werden, die anschließend den gewünschten Benutzern zugewiesen wird. Änderungen von Berechtigungen sind so nur einmal für die Gruppe erforderlich, aber nicht mehr bei jedem einzelnen Anwender.

Konfiguration im Admin-Bereich

Der Admin-Bereich kann nicht nur für die Benutzerpflege genutzt werden. Durch Hinzufügen einer Model-Klasse in der *admin.py* werden automatisch Anzeige-, Bearbeitungs- und Listendialoge erzeugt. Im Beispielprojekt ist dies für *Status* und *Liste* durch

*admin.site.register(Status)*

*admin.site.register(Liste)*

in der Datei *admin.py* erfolgt. Im Screenshot “Django\_admin” sind die gepflegten Einträge für *Status* zu sehen, die dort auch geändert, erweitert oder gelöscht werden können. Als Faustregel sollte gelten, dass der Admin-Bereich nichts für Endanwender ist, sondern für Konfigurationen und administrative Einstellungen reserviert ist. Für das Beispielprojekt der Aufgabenverwaltung nehmen wir dementsprechend an, dass die Auswahlmöglichkeiten für Status und Liste nicht durch Endanwender verändert werden können.

Unabhängig vom sonstigen, an Objekten orientierten Berechtigungsmodell wird der Zugang zum Admin-Bereich im Benutzerstamm über die Checkbox „Mitarbeiter“ gewährt. Die Bezeichnung ist im Unternehmenskontext unglücklich. „Key-User“ oder „Anwendungskoordinator“ wären wohl die treffenderen Rollenbezeichnungen.

Frontend Fummelei

Wenn die Web-Oberflächen nett aussehen und auch die User Experience nicht zu kurz kommen sollen, helfen die gängigen CSS/Javascript Frameworks. Insbesondere Bootstrap [5] ist im Django-Universum verbreitet und wird auch von vielen Addon-Packages verwendet. Während der Kreativität bei der Gestaltung nur wenige Grenzen gesetzt sind, reichen für Unternehmenszwecke in der Regel aufgeräumte Strukturen, die sich einheitlich durch die Anwendung durchziehen und möglichst intuitiv bedienbar sind. In unserem Beispielprojekt haben wir hierfür einen fixierten Header-Bereich mit Menü vorgesehen. Die einzelnen Dialoge sind im Wesentlichen mit flexiblen Containern –in Bootstrap auch „fluid“ genannt – gestaltet, die sich dynamisch der Fenstergröße anpassen. Die Anpassung der Darstellung für das jeweilige Endgerät, auch als Responsive Design bezeichnet, ist beispielhaft für die Überschriften der Dialoge umgesetzt. Beim Verkleinern des Browser-Fensters springt die Darstellung zwischen einer Lang- und einer Kurzform der Überschriften um. Das Ergebnis ist in Abbildung „crispy\_mit.png“ zu sehen.

Ist der grundsätzliche Rahmen in Form von Templates einmal gestaltet, kann Django wieder übernehmen. Formulare auf Webseiten können mit Bezug zu einem Model definiert werden. In unserem Beispielprojekt ist das Formular für unsere Aufgaben in *forms.py* definiert (siehe Listing 1). Passende Widgets für die Datentypen und die Überprüfung auf Pflichteingaben leitet Django automatisch aus dem referenzierten Model ab. Lediglich das Vorlagedatum wird abweichend zum Standard vorgegeben und soll den etwas hübscheren DatePicker aus dem Package *django-bootstrap-datepicker-plus* [6] nutzen. Mit nur wenigen Code-Zeilen kann anschließend der komplette View für das Anlegen und Ändern von Aufgaben in der *views.py* erstellt werden (Listing 2). Sowohl der *AufgabeCreateView* als auch der *AusgabeDetailView* erben von den Django Standardviews und erledigen ihre Aufgabe inklusive Datenabruf und -ablage in der Datenbank ohne zusätzliches Coding. Autorisierungs- und Berechtigungsprüfung erfolgen durch Vererbung von *LoginRequiredMixin* und *PermissionRequiredMixin*.

Views verbinden Formulare und Daten der Anwendung mit den Templates, die wiederum für die HTML-Darstellung und die Interaktion mit den Anwendern sorgen. Der Austausch erfolgt über den *Context*, ein Datenobjekt, das im Rahmen der Template Language in die HTML-Ausgabe eingebunden werden kann. Im Beispielprojekt klinken sich der *AufgabeCreateView* und der *AufgabeDetailView* durch eine Überladung der „get\_context\_data“ Methode ein, um die zwei Variablen TITLE\_KURZ und TITLE\_LANG in das Context-Objekt hineinzugeben. So kann das gleiche Template für das Anlegen und das Bearbeiten genutzt und die Überschrift durch den aufrufenden View angepasst werde (Listing 2).

Nicht ausgelassen werden sollte das Package *Crispy Forms* [7], dasdie Darstellung von Formularen aufpeppt, wahlweise ebenfalls mit Hilfe von Bootstrap. Die komplette HTML-Generierung nach Bootstrap-Manier für ein definiertes Formular wird im Template durch

{% crispy form %}

erreicht. Wer mehr Einfluss auf Darstellung und Anordnung der Formularfelder haben möchte, definiert in HTML eine individuelle Anordnung und greift dafür explizit auf einzelne Felder zu. Für das Feld *kurztext* unserer Aufgaben zum Beispiel durch:

{{ form.kurztext|as\_crispy\_field }}

Gefiltert

Eine weitere Aufgabe, die es in nahezu jeder Anwendung zu meistern gilt, ist die Abfrage des Datenmodells, meistens nach Vorgaben und Selektionskriterien, die der Anwender im Dialog vorgibt. Das Ergebnis der Abfrage wird in der Regel als Liste bzw. tabellarisch dargestellt. Auch hier helfen ergänzende Packages dem Entwickler und nehmen viel lästige Tipp-Arbeit ab. Gut aufeinander abgestimmt sind die Packages *django-filter* [8] und *django-tables2* [9], die in unserem Beispielprojekt für die Abfrage und tabellarische Auflistung unserer Aufgaben genutzt werden.

Bei der Festlegung von Filterkriterien hilft wiederum der Bezug zum Model, so dass in der *filters.py* in unserem Beispiel lediglich die zu berücksichtigenden Felder zu benennen sind (Listing 3). Zusätzlich ergänzt werden kann der Filtertyp: während „exact“ für Status und Liste die eindeutige Übereinstimmung vorgeben, führt „icontains“ bei Kurztext zu einer Durchsuchung des Felds nach dem eingegebenen Suchbegriff. Beim Vorlagedatum weicht unser Beispiel wieder vom Standard ab, indem es das bereits zuvor erwähnte DatePicker Widget festlegt. Unser Beispiel greift die gerade für Datumsfelder häufige Situation auf, dass nicht nach einem exakten Datum gesucht werden soll, sondern ein Datumsintervall gegeben ist („irgendwann zwischen 01.03.2020 und 15.03.2020“). Mit dem hier vorgestellten DatePicker lassen sich zwei Datumsfelder durch den Zusatz *start\_of()* und *end\_of()* verknüpfen, so dass bereits bei der Eingabe nur sinnvolle Auswahlen möglich sind.

Der so, mit nur wenigen Zeilen Quellcode, definierte Filter wird ähnlich wie ein Form in eine aus *FilterView* vererbte Klasse gesteckt und dort mit einem Template verknüpft. Die HTML-Generierung lässt sich ebenso wie bei den Forms mit *Crispy Forms* aufhübschen.

Für die tabellarische Aufbereitung der Ausgabe sorgt das Package *django-tables2*. Auch bei der Festlegung der Tabellendarstellung wird wieder der Bezug zum Model hergestellt. Seitenweises Blättern bei vielen Ergebnissen, Sortierreihenfolge sowie Darstellungsoptionen lassen sich mit wenigen Code-Zeilen hinzufügen. Eine auf- oder absteigende Sortierung der Tabelle ist ebenfalls standardmäßig durch Klick in die Spaltenübersicht möglich. Wer etwas Spezielles benötigt, kann Zelleninhalte aber auch nach eigenem Belieben umgestalten. In unserem Beispielprojekt in *tables.py* sorgen die Methoden „render\_kurztext“ und „render\_vorlage\_datum“ dafür, dass nicht der Standardzelleninhalt verwendet wird (Listing 4). Den Kurztext wandeln wir in einen Link um, der in den Dialog zur Bearbeitung der Aufgabe wechselt. Beim Vorlagedatum prüfen wir, ob dieses bereits in der Vergangenheit liegt und färben es in diesem Fall rot ein.

Das Ergebnis *AufgabeFilterView* in *views.py* besteht inklusive Login- und Berechtigungsprüfung, Verweis auf Template, Filter und Tabelle sowie das Aufgaben-Model aus gerade einmal sechs Zeilen Quellcode (Listing 5). Das Ergebnis in Abbildung „filter\_tabelle“ kann sich dafür durchaus sehen lassen. Aus der oben bereits erwähnten Filter-Festlegung „icontains“ für das Feld Kurztext wird als Feldbezeichnung automatisch „Kurztext enthält“ generiert. Die Dropdown-Boxen für Status und Liste sind ohne weiteres Zutun mit den möglichen Werten aus dem Datenmodell vorbelegt. Die Datumsfelder öffnen bei Klick auf das Kalendersymbol einen Datepicker, der durch die Verknüpfung der beiden Felder bereits darauf achtet, dass das „Von Datum“ vor oder gleich dem „Bis Datum“ ist. Auch hinter der HTML-Darstellung der Tabelle liegt ein Template, von denen das *django-tables2* Package mehrere zur Auswahl mitbringt. Standardmäßig wird auch hier ein Bootstrap-Template verwendet.

Schnell integriert

Während der Entwickler auf der Programmierebene mit erstaunlich wenigen Zeilen Quellcode in Django bereits respektable Ergebnisse hervorzaubern kann, ist für die HTML-Templates und das Frontend schon deutlich mehr Tipp-Arbeit erforderlich. Auf der anderen Seite gibt es gerade im HTML5 Bereich verschiedenste Frameworks wie Vue.js oder Angular, die sich dank intensivem JavaScript Einsatz noch deutlich mehr nach lokaler Anwendung als nach Webseite anfühlen. Vielleicht soll eine neue Python-Anwendung auch in eine bestehende Webservice Landschaft einzubinden, für die es schon Oberflächen, Benutzerverwaltung etc. gibt? Auch hier kann Django mit dem Package *Django-Rest-Framework* [10] helfen, das häufig nur als DRF abgekürzt wird. Wie der Name schon vermuten lässt, rüstet dieses Package bestehende Anwendungen mit REST-Schnittstellen auf, die von anderen Anwendungen einfach konsumiert werden können.

Nach der Installation und Einbindung des Package wird für die gewünschten Objekte ein Serializer angelegt, in unserem Beispielprojekt in der *serializers.py*. Auch hier kürzt sich die Tipp-Arbeit für den Entwickler ein, wenn der Bezug zu einem Model hergestellt wird. In unserem Fall zeigen wir das Vorgehen beispielhaft für das Aufgaben-Objekt und geben die im Datenmodell verknüpften Informationen zu Status und Liste als verschachteltes Objekt zurück. Daher legen wir auch Serializer für Status und Liste an, auf die im Aufgaben-Serializer verwiesen wird (Listing 6).

Nun werden nur noch Views benötigt, die die HTTP-Anfragen annehmen und verarbeiten. Im Beispielprojekt sind dies *AufgabeApiListe* und *AufgabeApiDetail*, die entweder eine Liste aller Aufgaben zurückgeben oder über das Feld Id eine bestimmte Aufgabe verfügbar machen. Beide nutzen dafür den *AufgabeSerializer*.

Welche Operationen auf der Liste und dem einzelnen Objekt über REST verfügbar sein sollen, wird über die Basisklasse festgelegt. *AufgabeApiListe* erbt von ListCreateAPIView und ermöglicht auch die Neuanlage einer Aufgabe per POST. *AufgabeApiDetail* erbt von *RetrieveUpdateDestroyAPIView*, so dass neben dem Abrufen einer Aufgabe auch das Bearbeiten und Löschen möglich ist. Für den Entwickler hilfreich ist, dass die REST-Schnittstellen direkt auch im Browser getestet werden können. Im Beispielprojekt ist die URL: <http://localhost:8000/aufgabe/api> definiert, die in Abbildung „rest\_api“ gezeigt ist.

Weiter unten und nicht mehr im Bild zu sehen werden außerdem Eingabefelder für die Neuanlage per POST angezeigt. Einzelne Aufgaben, zum Beispiel die Aufgabe mit der Id 1 können entsprechend der REST-Logik direkt adressiert werden: <http://localhost:8000/aufgabe/api/1> (siehe Abbildung „rest\_api2“). Wohlgemerkt bekommt der Entwickler auch diese Oberflächen frei Haus und ohne weiteres Handanlegen. Beide Views sind mit nur wenigen Quellcode-Zeilen erstellt (Listing 7) und regeln neben den möglichen REST-Operationen auch die Nutzer-Authentifizierung und die Berechtigungsprüfung. Wer zusätzlich eine Filtermöglichkeit benötigt, kann diese über das bereits vorgestellte *django-filter* Package einbauen, das gut mit dem REST-Framework zusammenspielt.

Unterm Strich

Django hat sich seinen guten Ruf in den letzten 15 Jahren nicht umsonst erworben. Durch den Modell-basierten Entwicklungsansatz, das Objekt-relationale Mapping und die konsequente Bezugnahme auf das Modell in der weiteren Anwendung, entstehen mit nur wenig Entwicklungsaufwand solide und gut wartbare Web-Anwendungen. Durch die hier vorgestellten Erweiterungspakete lassen sich die im Standard fehlenden Helferlein ergänzen, die neben einer Active Directory Anbindung auch im Unternehmensumfeld typische Aufgaben wie flexible Datenselektionen und tabellarische Ausgaben zur Verfügung stellen.

Hakeliger wird es im Bereich des HTML-Frontends, da das Django Standard-HTML nicht wirklich überzeugt. In unserem Beispielprojekt haben wir Bootstrap genutzt und sind ansonsten in der Django-Welt geblieben. Alternativ lassen sich aber auch recht einfach REST-Schnittstellen erstellen, die nicht nur als Schnittstelle zu weiteren Anwendungen dienen können, sondern auch zur Einbindung in andere Frontend-Frameworks. Hat man sich einmal im Django-Universum eingewöhnt, können Python-Programme mit vergleichsweise wenig Aufwand in eine zuverlässige unternehmenstaugliche Hülle gesteckt werden.

Quellen:

[1] Django Documentation: <https://docs.djangoproject.com/en/3.1/>

[2] Aratyn, Tom: “Building Django 2.0 Web Applications”, Packt Publishing, 2018

[3] Quellcode Beispielanwendung online bei heise

[4] Django-python3-ldap: <https://github.com/etianen/django-python3-ldap>

[5] Bootstrap Projektseite: <https://getbootstrap.com>

[6] Bootstrap DatePicker für Django: <https://github.com/monim67/django-bootstrap-datepicker-plus>

[7] Django Crispy Forms: <https://django-crispy-forms.readthedocs.io/en/latest/#>

[8] Django-Filter Package: <https://django-filter.readthedocs.io/en/stable/>

[9] Django-tables2 Package: <https://django-tables2.readthedocs.io/en/latest/>

[10] Django Rest Framework: <https://www.django-rest-framework.org>

Bildunterschriften:

„django\_admin.png“: Im Django Admin-Bereich können neben Benutzern und Berechtigungen auch beliebige andere Objekte zur Bearbeitung eingebunden werden.

“crispy\_mit.png”: Bootstrap und Crispy Forms sorgen für eine aufgeräumte Oberfläche

„filter\_tabelle.png“: Das django-filter und django-tables2 Package ergänzen sich ideal, um flexible Abfragen und tabellarische Ausgaben zu erzeugen.

„rest\_api.png“: REST-Schnittstellen sind nicht nur schnell angelegt, sondern können auch im Browser geprüft werden.

„rest\_api2.png“: Je nach genutzter Basisklasse sind auch Änderungen und Löschen möglich, inklusive Berechtigungsprüfung.

ixtract:

+ Ausgereiftes Web-Framework, das wenig Wünsche offen lässt.

+ Offene Architektur mit vielen frei verfügbaren Packages und eigenen Erweiterungsmöglichkeiten

+ Starke Community

- Einarbeitung in Django durchaus komplex

- Frontend-Gestaltung erfordert zusätzliches Framework wie z. B. Bootstrap

Listing 1

class AufgabeForm(ModelForm):

vorlage\_datum = DateField(input\_formats=["%d.%m.%Y"], label="Vorlage",

widget=DatePickerInput(

options={

"format": "DD.MM.YYYY",

"showClose": True,

"showClear": True,

"showTodayButton": True,

}

))

class Meta:

model = Aufgabe

fields = (

'id',

'kurztext',

'langtext',

'vorlage\_datum',

'status',

'liste',

)

Listing 2

class AufgabeCreateView(LoginRequiredMixin, PermissionRequiredMixin, CreateView):

permission\_required = 'Aufgaben.add\_aufgabe'

model = Aufgabe

template\_name = 'detail.html'

form\_class = AufgabeForm

success\_url = '/'

def get\_context\_data(self, \*\*kwargs):

context = super().get\_context\_data(\*\*kwargs)

context['TITLE\_KURZ'] = 'Anlegen'

context['TITLE\_LANG'] = 'Aufgabe anlegen'

return context

class AufgabeDetailView(LoginRequiredMixin, PermissionRequiredMixin, UpdateView):

permission\_required = 'Aufgaben.change\_aufgabe'

model = Aufgabe

template\_name = 'detail.html'

form\_class = AufgabeForm

success\_url = '/'

def get\_context\_data(self, \*\*kwargs):

context = super().get\_context\_data(\*\*kwargs)

context['TITLE\_KURZ'] = 'Bearbeiten'

context['TITLE\_LANG'] = 'Aufgabe bearbeiten'

return context

Listing 3

class AufgabeFilter(FilterSet):

vorlage\_datum\_\_gte = DateFilter(

field\_name='vorlage\_datum',

lookup\_expr='gte',

input\_formats=["%d.%m.%Y"],

label='Von Datum...',

widget=DatePickerInput(

options={"format": "DD.MM.YYYY",

"showClose": True,

"showClear": True,

"showTodayButton": True,

}

).start\_of('vorlage'))

vorlage\_datum\_\_lte = DateFilter(

field\_name='vorlage\_datum',

lookup\_expr='lte',

input\_formats=["%d.%m.%Y"],

label='...bis Datum',

widget=DatePickerInput(

options={"format": "DD.MM.YYYY",

"showClose": True,

"showClear": True,

"showTodayButton": True,

}

).end\_of('vorlage'))

class Meta:

model = Aufgabe

fields = {

'kurztext': ['icontains', ],

'status': ['exact', ],

'liste': ['exact', ],

}

Listing 4

class AufgabeTable(Table):

id = Column(visible=False)

langtext = Column(visible=False)

def render\_kurztext(self, record, value):

url = Aufgabe.objects.get(pk=record.id).get\_absolute\_url()

html = format\_html('<a href="{}">{}</a>', url, value)

return html

def render\_vorlage\_datum(self, record, value):

today = datetime.now().date()

value\_date = value.strftime("%d.%m.%Y")

if value < today:

html = format\_html('<span class="text-danger">{}</span>', value\_date)

else:

html = value\_date

return html

class Meta:

model = Aufgabe

per\_page = 20

order\_by = ('vorlage\_datum',)

attrs = {"class": "table table-striped",

"thead": {

"class": "thead-light"

}

}

Listing 5

class AufgabeFilterView(LoginRequiredMixin, PermissionRequiredMixin, SingleTableMixin, FilterView):

template\_name = 'liste.html'

permission\_required = 'Aufgaben.view\_schritt'

filterset\_class = AufgabeFilter

model = Aufgabe

table\_class = AufgabeTable

Listing 6

class AufgabeSerializer(serializers.ModelSerializer):

status = StatusSerializer(

many=False,

read\_only=True,

)

liste = ListeSerializer(

many=False,

read\_only=True,

)

class Meta:

model = Aufgabe

fields = ('id',

'kurztext',

'langtext',

'vorlage\_datum',

'liste',

'status',

)

Listing 7

class AufgabeApiDetail(generics.RetrieveUpdateDestroyAPIView):

authentication\_classes = (BasicAuthentication, SessionAuthentication,)

permission\_classes = (

permissions.DjangoModelPermissions,

)

queryset = Aufgabe.objects.all()

serializer\_class = AufgabeSerializer