
REMOVE BEFORE FLIGHT

SILVR

Machine à états finis pour vos modèles Django

26 mars 2024

SILVR

Supercharged Business Loans

We love Python
We love Django

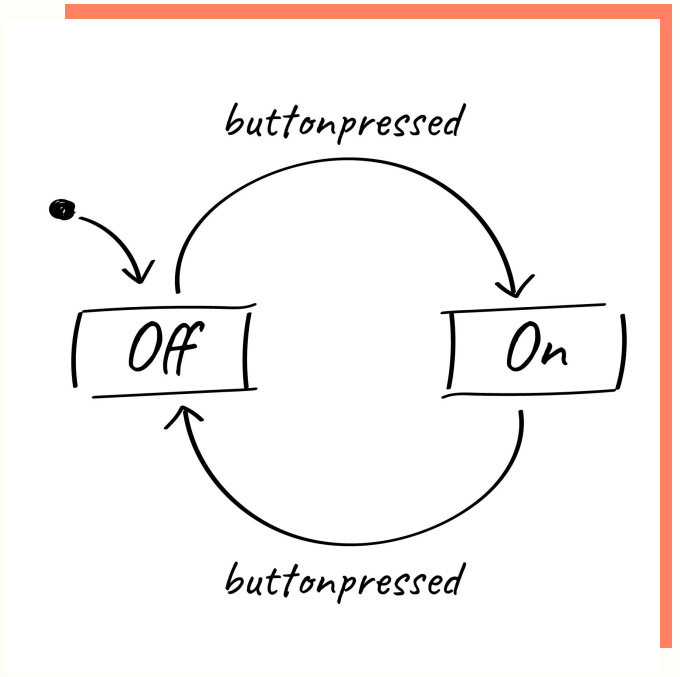
Pascal

Rémy

app.silvr.co

Qu'est-ce qu'une "State-Machine"?

STATE MACHINE =
ÉTATS FINIS +
TRANSITIONS



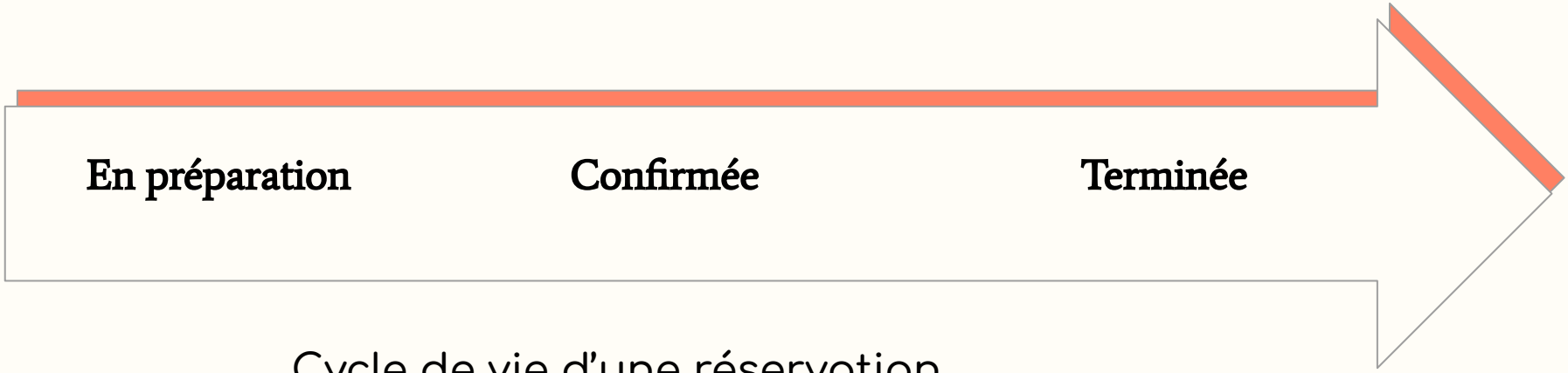
Avez vous déjà vu ?

```python
def publish_article_view(request, blog_article):
    ...

    if blog_article.status == ArticleStatus.DRAFT:
        blog_article.status = ArticleStatus.PUBLISHED
        blog_article.save()
        send_email_published_notification(blog_article.author.email)

        # Update RSS feed
        rss_feed.update()

    return ...
```

```python
def unpublish_article_view(request, blog_article):
    ...
    blog_article.status = "unpublished"
    blog_article.save()

    return ...
```

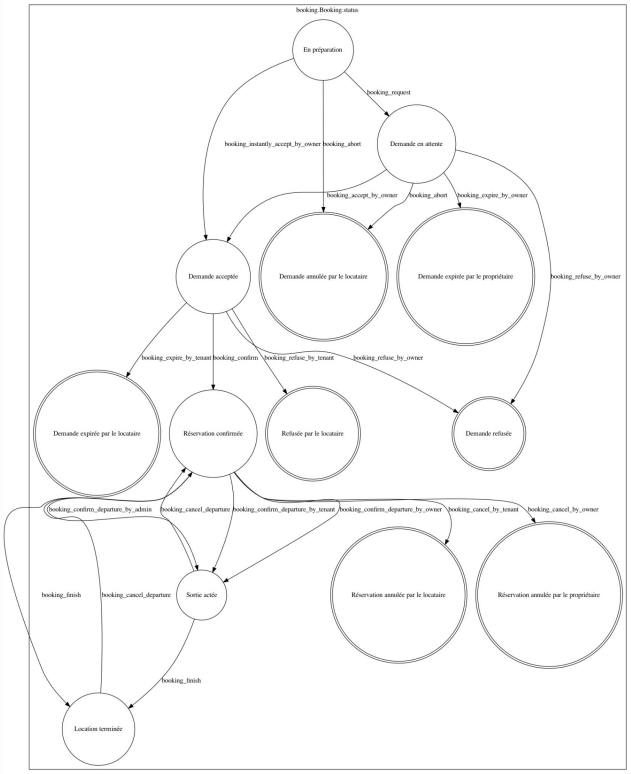
```python
def highlight_article_view(request, blog_article):
    ...
    blog_article.status = ArticleStatus.HIGHLIGHTED
    blog_article.save()

    return ...
```

Booking : Ça part toujours d’une bonne intention !



Cycle de vie d’une réservation

Booking : Mais quelques mois plus tard



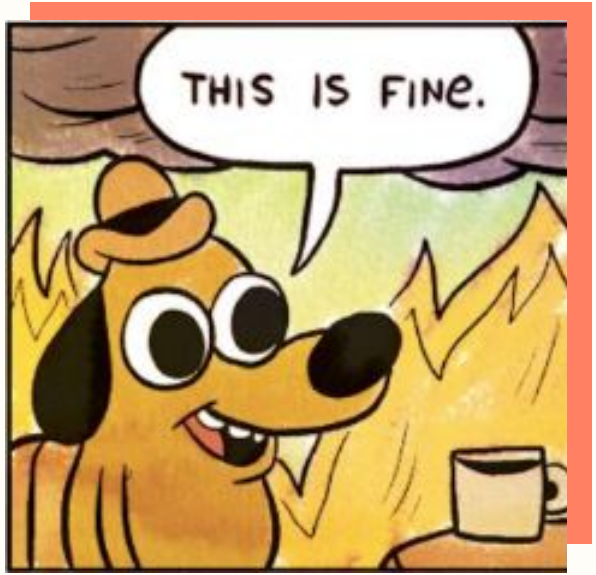
12 états
31 transitions

- **En préparation**
- Demande en attente
- **Réservation confirmée**
- Demande acceptée
- Refusée par le locataire
- Demande refusée
- Réservation annulée par le locataire
- Réservation annulée par le propriétaire
- Sortie actée
- **Location terminée**
- Demande annulée par le locataire
- Demande expirée par le locataire
- Demande expirée par le propriétaire

Booking : Mais quelques mois plus tard

12 états
31 transitions

Problématiques



- Savez vous encore **lister les transitions** métiers ?
- Pouvez vous les **représenter**
- Pouvez vous **vérifier la cohérence** ?
- Pouvez vous **ajouter un nouvel état** sans tout casser ?
- Êtes-vous sûr que les transitions **vérifient toutes les mêmes conditions** ?
- Êtes-vous sûr que les transitions **déclenchent les mêmes actions** ?

State machine FTW !

La state machine définie au niveau du modèle :

- **Liste exhaustive** des states et des transitions
- Vérification des **transitions possibles** depuis un état
- Les **sides effects définis** une fois pour toutes

Et tout ça accessible sur l’instance du modèle.

django-fsm, une implémentation pour les modèles de Django

```python
class ArticleStatus(models.TextChoices):
    DRAFT = "draft", _("Draft")
    PUBLISHED = "published", _("Published")
    UNPUBLISHED = "unpublished", _("Unpublished")
    HIGHLIGHTED = "highlighted", _("Highlighted")


class BlogArticle(models.Model):
    state = fsm.FSMField(
        choices=ArticleStatus.choices,
        default=ArticleStatus.DRAFT,
    )

    published_at = models.DateTimeField(...)

    @transition(
        source=ArticleStatus.DRAFT,
        target=ArticleStatus.PUBLISHED,
    )
    def publish(self):
        self.published_at = timezone.now()
        send_email_published_notification(self.author.email)
        # Update RSS feed
        rss_feed.update()
```

```python
def publish_article_view(request, blog_article):
    ...
    blog_article.publish()
    blog_article.save()

    return ...
```

Attention à ne pas oublier l'appel à save() après les transitions.

django-fsm, une implémentation pour les modèles de Django

```python
class ArticleStatus(models.TextChoices):
    DRAFT = "draft", _("Draft")
    PUBLISHED = "published", _("Published")
    UNPUBLISHED = "unpublished", _("Unpublished")
    HIGHLIGHTED = "highlighted", _("Highlighted")


class BlogArticle(models.Model):
    state = fsm.FSMField(
        choices=ArticleStatus.choices,
        default=ArticleStatus.DRAFT,
    )

    published_at = models.DateTimeField(...)

    @transition(
        source=ArticleStatus.DRAFT,
        target=ArticleStatus.PUBLISHED,
    )
    def publish(self):
        self.published_at = timezone.now()
        send_email_published_notification(self.author.email)
        # Update RSS feed
        rss_feed.update()
```

Liste exhaustive des states et des transitions

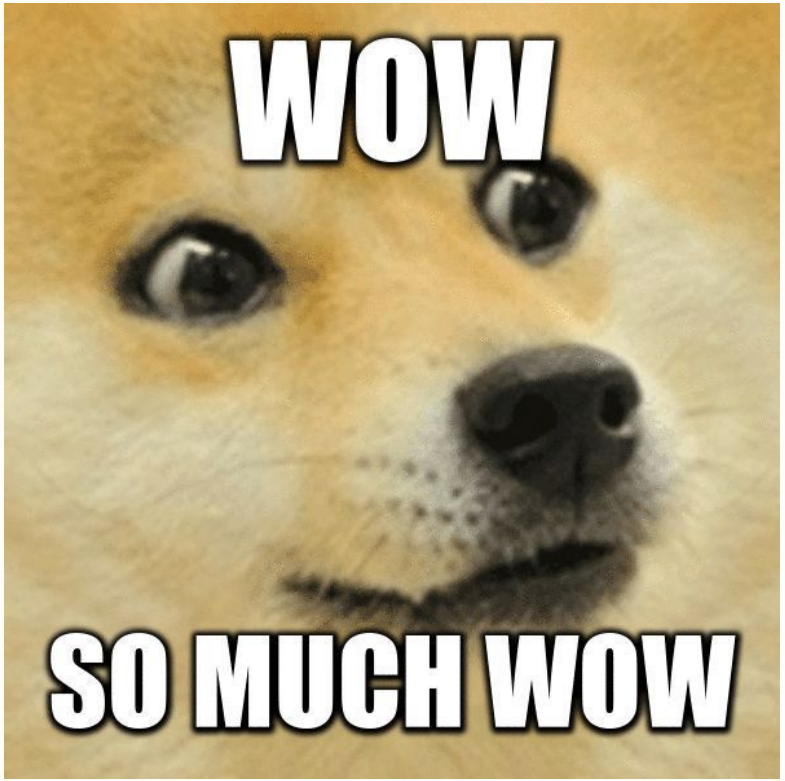
Vérification des **transitions possibles** depuis un état

Les **sides effects définis** une fois pour toutes

Et tout ça accessible sur l'instance du modèle.

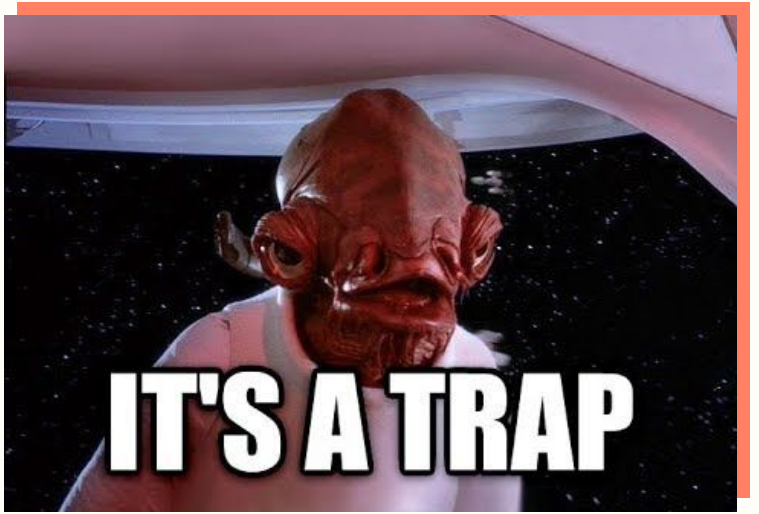
```python
def publish_article_view(request, blog_article):
    ...
    blog_article.publish()
    blog_article.save()

    return ...
```

Attention à ne pas oublier l'appel à save() après les transitions.


WOW
SO MUCH WOW

Traps

- Les cycles de vie métier ne sont jamais linéaires
- Une state machine est un pattern applicatif
- Attention le code de la transition s'exécute avant que la transition soit appliquée
- Une state-machine trop restrictive qui bloque certains cas business
- Plusieurs state-machine imbriquées / liées



Exemple avancé

```python
class BlogArticle(models.Model):
    state = fsm.FSMField(
        choices=ArticleStatus.choices,
        default=ArticleStatus.DRAFT,
    )

    published_at = models.DateTimeField(...)

    @transition(
        source=[ArticleStatus.DRAFT, ArticleStatus.UNPUBLISHED],
        target=RETURN_VALUE(ArticleStatus.PUBLISHED, ArticleStatus.HIGHLIGHTED),
        conditions=[is_valid_article],
        permission=[is_author],
        custom={"description": "Publish the article"},
    )
    def publish(self):
        self.published_at = timezone.now()
        transaction.on_commit(
            partial(
                send_email_published_notification,
                self.author.email
            )
        )
        # Update RSS feed
        transaction.on_commit(rss_feed.update)

        # if first time publishing, hihglight the article
        if self.author.is_first_class:
            return ArticleStatus.HIGHLIGHTED

        return ArticleStatus.PUBLISHED
```

Fonctions avancées

```
9     @transition(
10        source=[ArticleStatus.DRAFT, ArticleStatus.UNPUBLISHED],
11        target=RETURN_VALUE(ArticleStatus.PUBLISHED, ArticleStatus.HIGHLIGHTED),
12        conditions=[is_valid_article],
13        permission=[is_author],
14        custom={"description": "Publish the article"},
15    )
```

Source : Liste d'états autorisés pour exécuter la transition
Target : État de destination après l'exécution
Conditions : valident certaines valeurs pour autoriser la transition
Permissions : vérifient que l'utilisateur est autorisé à exécuter la transition
Custom : ajoute des propriétés à la transition (pour l'admin notamment)

Fonctions avancées

```
9     @transition(
10        source=[ArticleStatus.DRAFT, ArticleStatus.UNPUBLISHED],
11        target=RETURN_VALUE(ArticleStatus.PUBLISHED, ArticleStatus.HIGHLIGHTED),
12        conditions=[is_valid_article],
13        permission=[is_author],
14        custom={"description": "Publish the article"},
15    )
```

On peut utiliser RETURN_VALUE pour laisser la transition retourner l'état de destination
On peut utiliser GET_STATE pour décider de l'état en fonction des paramètres de la transition

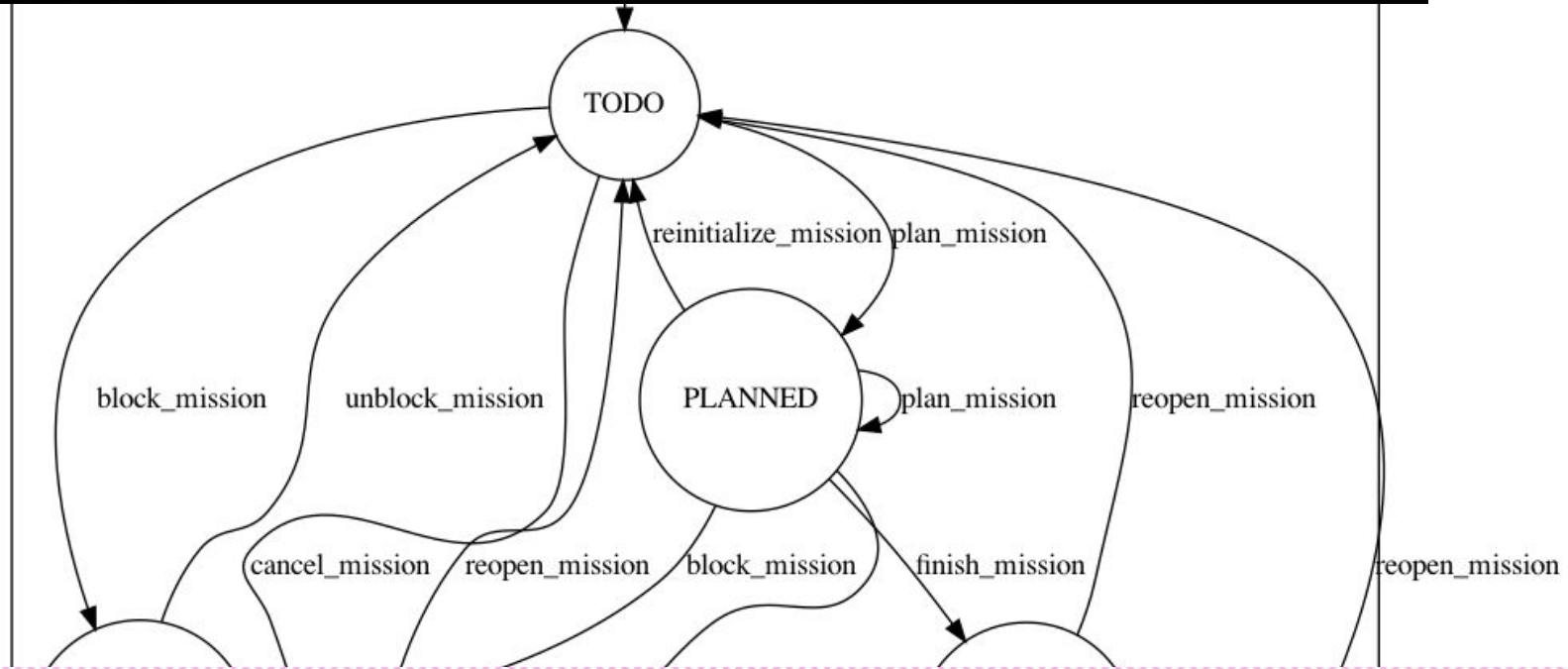
Fonctions avancées

La source “+” autorise les transitions depuis tous les autres state que la target.
La source “*” autorise les transitions depuis tous les states incluant la target.

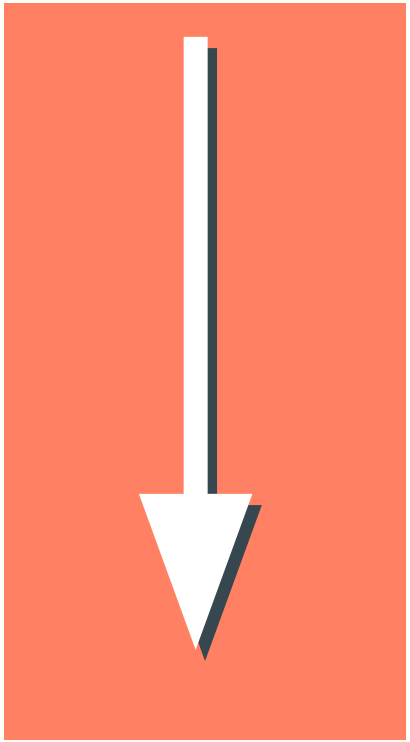
```python
@transition(
    field=status,
    source="+",
    target=PristineRequestStatus.APPROVED,
)
def approved(self) -> None:
    raw_message = f"{self.readable_id_name} financing request was approved."
    task_post_answer_to_slack(
        raw_message=raw_message,
        title=f":white_check_mark: {raw_message}",
        body="Backoffice <{}{}?status=approved|link>".format(
            settings.FQDN,
            reverse("backoffice:request-list"),
        ),
    )
```

🍒 le code c’est bien, un schéma c’est mieux !

```
./manage.py graph_transitions -o blog_transitions.png myapp.Blog
```

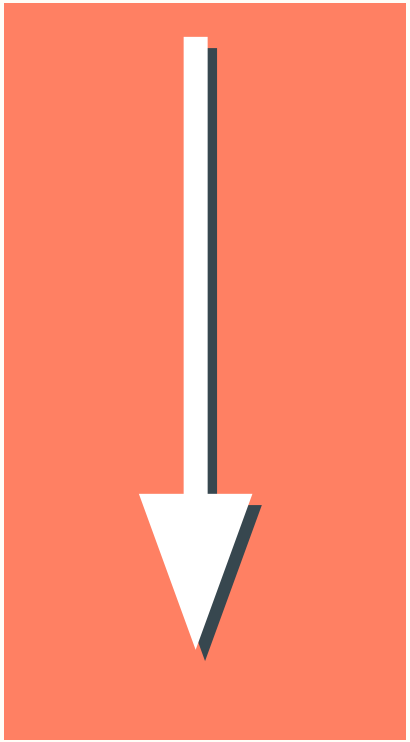


Plomberie interne de django-fsm



- Vérification de l'état Source
- Vérification des Conditions
- Vérification des Permissions
- Exécution de la fonction
- Changement d'état
- Save => commit de la transaction en DB
    - transaction.on_commit

Plomberie interne de django-fsm



- Vérification de l'état Source
- Vérification des Conditions
- Vérification des Permissions
- Exécution de la fonction
- Changement d'état
- Save => commit de la transaction en DB
  - transaction.on_commit

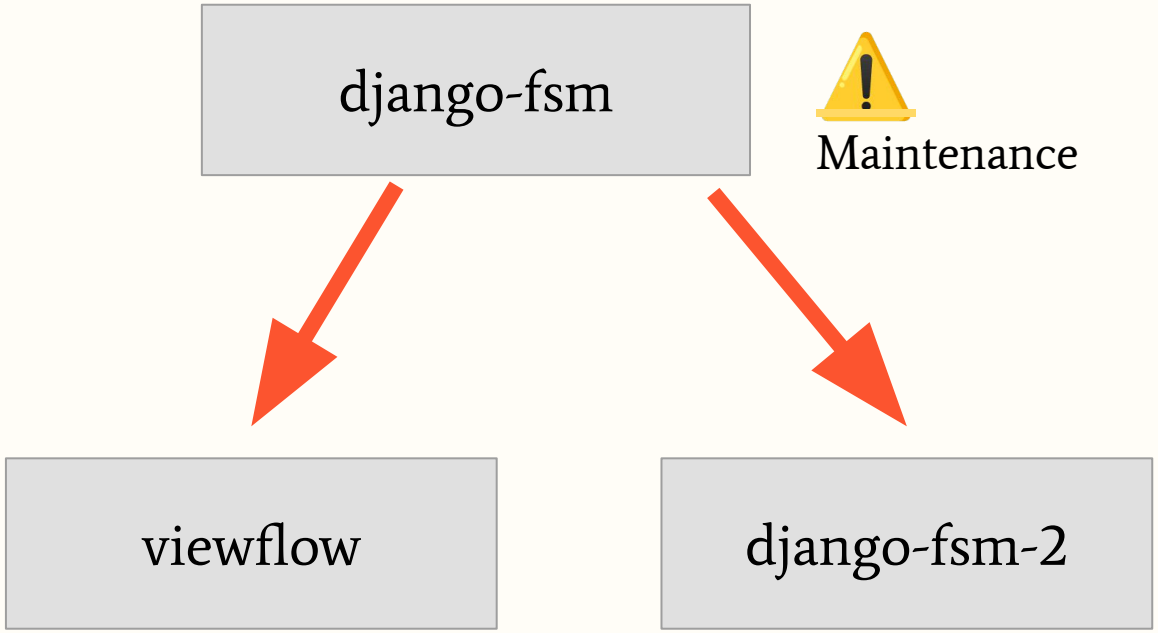
can_proceed()

Tester sa state-machine

- On peut simplement tester **l'appel de la transition**
- On peut **tester unitairement le code des transitions**
- On peut **vérifier la liste des transitions disponibles** pour chaque état

⇒ tests unitaires complets et exhaustifs.

django-fsm state



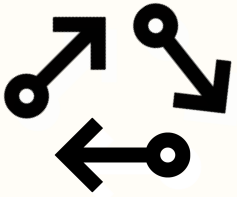
https://github.com/pfouque/django-fsm-2

Conclusion

Les state-machines offrent une approche **structurée et efficace** pour gérer la complexité des changements d'états métiers.

Leur utilisation permet d'améliorer la **clarté du code**, la g**estion des états et transitions**, la facilité de test et d'assurer une plus **grande sécurité et stabilité** aux changements.

En intégrant une state-machine dans votre projet, vous pouvez créer des logiciels **plus robustes, évolutifs et faciles à maintenir** en limitant les bugs lors des modifications.



SILVR

Merci pour votre écoute attentive,

Avez-vous des questions ?