Django演習
(関数ベースビュー)

掲示板サイトを作成する

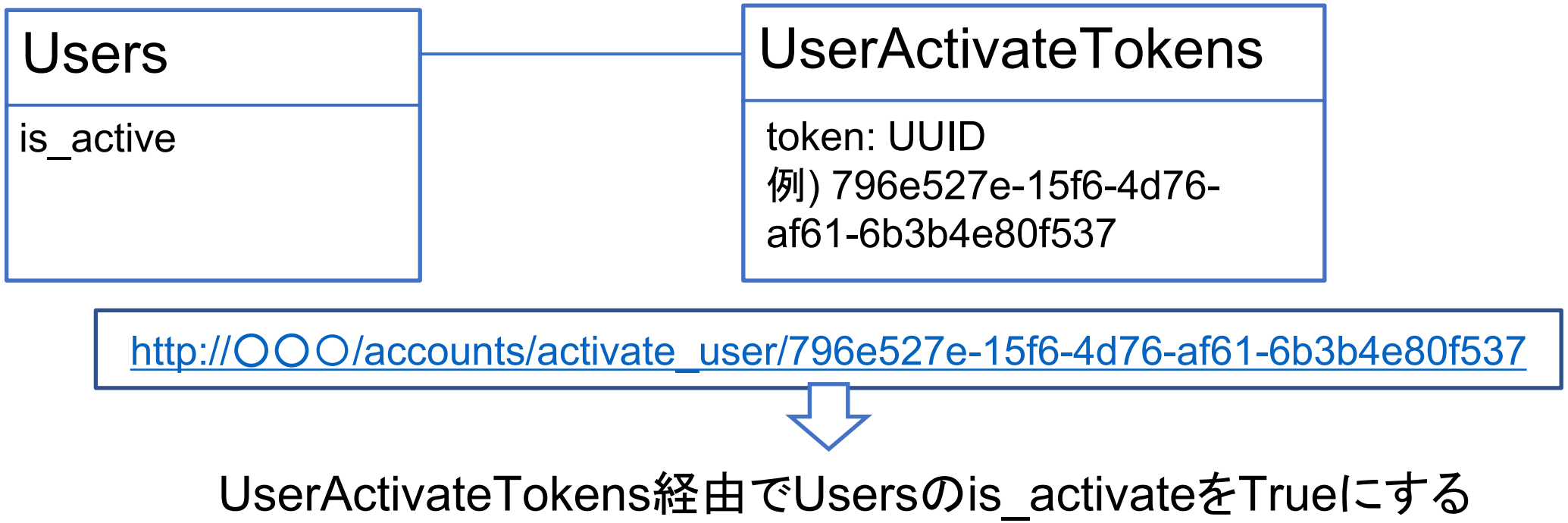
以下の機能を持った掲示板サイトを作成しましょう

1. ユーザ登録、ログイン処理を行う。この際に、ユーザ登録時には、UUIDを発行してメールアドレスにユーザを本登録するURLを送信するような仕組みにする（ただし、メール送信機能は実装しない）
2. 自分のプロフィール（名前、年齢、写真）を変える画面がある。
3. 掲示板ページにスレッドを立てることができる。(テーマを決める)
4. スレッドに対して、各ユーザが書き込みを行える
5. 一時保存として、書き込みをキャッシュに入れることができる。
6. 一時保存は、AJAXで行う。
7. エラーハンドリングとして、404のエラー画面を作成する。

掲示板サイトを作成する

以下の機能を持った掲示板サイトを作成しましょう

1. ユーザ登録、ログイン処理を行う。この際に、ユーザ登録時には、UUIDを発行してメールアドレスに**ユーザを本登録するURLを送信する**ような仕組みにする（ただし、メール送信機能は実装しない）



Djangoの応用機能（シグナル）

シグナルは特にモデルでよく実装されるが、ある特定の処理を実行した際に、自動的に呼び出される処理を定義したい場合に用いる。
post_saveで、特定のモデルのデータを保存した後に、実行する関数を定義する。
例)

```
from django.contrib.auth.models import User
from django.db.models.signals import post_save

def save_profile(sender, instance, **kwargs):
    instance.profile.save()

post_save.connect(save_profile, sender=User)
```

このようにすると、Userクラスで新たにオブジェクトが追加されるたびに、save_profile関数が呼び出される。

Djangoの応用機能（シグナル）

特定のモデルのデータを保存した後に、実行する関数を指定する方法はもう一つあり、@receiverを関数の前に付与すればよい

例)

```
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=User)
def save_profile(sender, instance, **kwargs):
    instance.profile.save()
```

同様、Userクラスで新たにオブジェクトが追加されるたびに、save_profile関数が呼び出される。
また、Signalにはpost_save以外にも方法があり

pre_save: 保存処理の前に実行
pre_delete: 削除処理の前に実行
post_delete: 削除処理の後に実行

Djangoの応用機能（Modelマネージャー）

モデルを用いるときは、テーブルの定義を記述するクラスとテーブルのデータ挿入、取り出しをするクラスとを分けることもある。

テーブルのデータ挿入、取り出しをするクラスは、models.Managerを継承して作成する。

```
class UserManager(models.Manager): # Managerの作成
    def counts(self):
        pass

class User(models.Model):
    name = models.CharField(max_length=200)
    :
    objects = UserManager() # Managerの指定
```

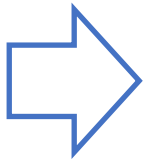
UserManagerのcountsメソッドを呼び出す際には、User.objects.count()とする。

Djangoの応用機能（ログイン、ログアウト、メッセージ）

ログインを実行するには、カスタマイズしたUserにdjango.contrib.auth.models.UserManagerか、UserManagerを継承したクラスを指定したobjectsを定義する必要がある

遷移前に、メッセージを格納して遷移先の画面上にメッセージを表示するには、
django.contrib.messages
を用いる。
(https://docs.djangoproject.com/ja/3.1/ref/contrib/messages/)
debug, info, success, warning, errorにメッセージを入れる

```
messages.debug(request, メッセージ')
messages.info(request, メッセージ')
messages.success(request, メッセージ')
messages.warning(request, メッセージ')
messages.error(request, メッセージ')
```



messages変数に格納されて、以下のように表示できる

```
{% for message in messages %}
<div>
    {{ message.message }}
</div>
{% endfor %}
```

```
from Django.contrib.messages import constants as message_constants
MESSAGE_LEVEL = message_constants.INFO # メッセージの表示レベルの変更(settings.py)
```
デフォルトはINFO

Djangoの応用機能（ModelFormでデータ、パスワードの更新）

ModelFormでデータの更新をするには、Formの作成時にinstance=として、更新したいレコードを指定する。
forms.ModelForm(request.POST or None, instance=model)

パスワードを変更する場合の注意点

ログインをするとログインユーザとシステムの間でセッションが張られるが、パスワードを変更する場合には、ユーザのセッションを更新することが必要である。

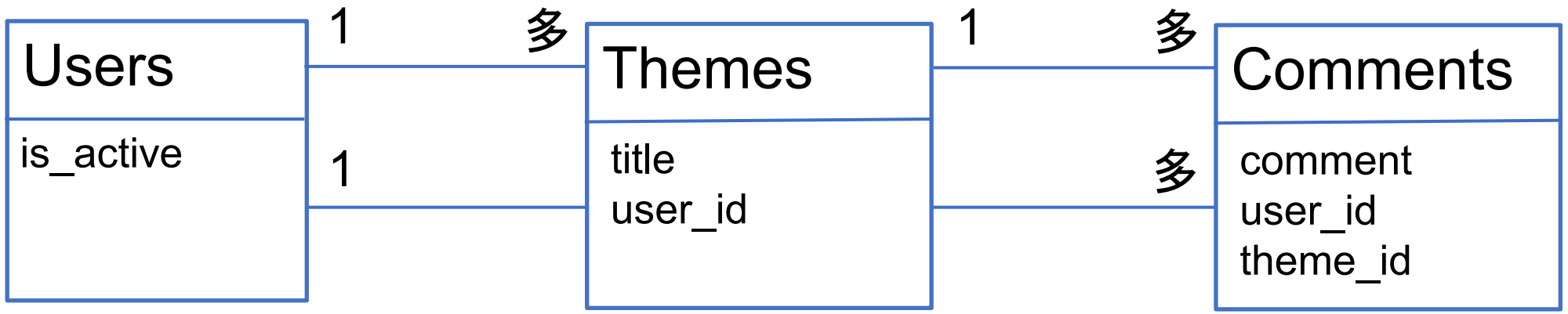
このセッションの更新をするには、
django.contrib.auth.update_session_auth_hash
を利用する。

update_session_auth_user(request, instance)として、リクエストとインスタンスを指定する

request.user: **ログインしているユーザインスタンスを取得することができる。**

Djangoの応用機能（掲示板画面を作成する）

掲示板画面を作成するには、以下のテーブルを作成すると良い
各掲示板のテーマを決めるthemesテーブルとコメントを行うcommentsテーブルを作成する。



そして、各テーブルに対して、中にデータを挿入する処理を書いて行く

Djangoの応用機能（AJAX）

Ajaxは非同期通信を行って、インタフェース構築する技術で、画面遷移なくサーバとの情報のやり取りを行う
Ajaxを用いることで、アプリケーションのUIを改善することができる。
クライアント側では、jQueryを用いて、以下のような処理を作成して、リクエストをサーバ側に投げる。

```
$.ajax({
      url : “create_post/”, // 実行するURLを指定
      type : “POST”, // HTTPメソッド
      data : { the_post : $(‘#post-text’).val() }, // 送信するデータ

      success : function(json) { // 実行が成功した場合の処理
      },
      error : function(xhr,errmsg,err) { //  実行が失敗した場合の処理
     }
  });
```

Djangoの応用機能（AJAX）

サーバ側では、以下のようにjsonを返す処理を定義する。

```
 from django.http import JsonResponse
from django.core import serializers



if request.is_ajax:
    # 処理
    json_instance = serializers.serialize('json', [ instance, ]) # jsonに変換する
    return JsonResponse({"instance": json_instance}, status=200) # レスポンスを返す

    または
    return HttpResponse(
            json.dumps(response_data),
            content_type="application/json"
        )
```

Djangoの応用機能（CACHE）

キャッシュとは、アプリケーションで必要な情報を一時的にメモリ上に保存して利用する機能
(ドキュメント: https://docs.djangoproject.com/ja/3.1/topics/cache)
Djangoで利用するキャッシュは、以下の通り

1. **Memcached**: メモリー上のキャッシュで、複数のサーバで共有もする。（aws elasticacheなど）

2. **Database**: データベースに保存するキャッシュ。取り出す速度は遅いが、複雑なデータを大容量で格納できる

3. **File system**: ファイル上に分割して保存するキャッシュ。速度は遅いが、管理はしやすい

4. **Local memory**: ローカルPCのメモリー上に保存するキャッシュ。デフォルトのキャッシュだが、サーバ間で共有することはできない。

5. **Dummy**: 実際にはキャッシュを行わないキャッシュのインタフェースを提供するダミーのキャッシュ。開発環境でのテストで用いる。

Djangoの応用機能（CACHE）

キャッシュの設定は、settings.pyに以下のような内容で記述する。

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11212‘,
        ]
    }
}
```

キャッシュの操作(django.core.cache.cache)

cache.set(‘my_key’, ‘hello world!’) **# my_keyにhello worldをキャッシュする**

cache.get(‘my_key’, ‘default’) **# my_keyに該当する値をキャッシュから取り出す。存在しない場合は、defaultを返す**

cache.clear() **# キャッシュを全て削除する**

cache.delete(‘a’) **# キャッシュからaに該当するものを削除する**