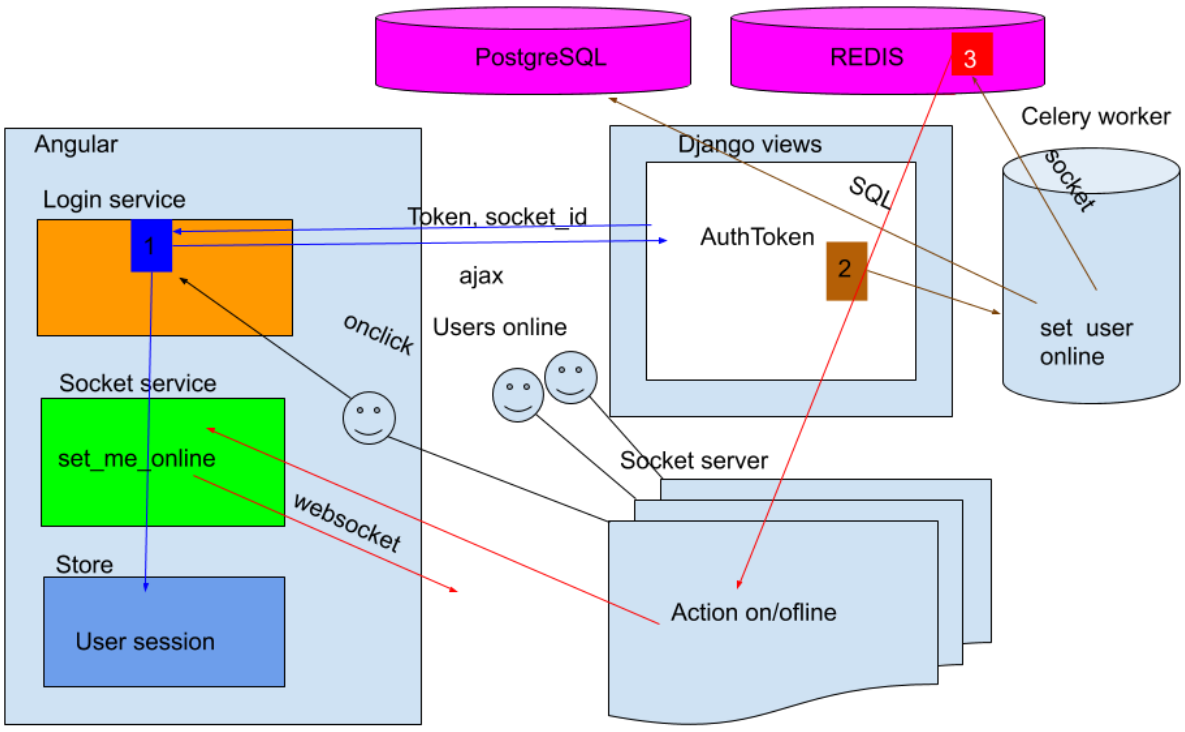
Реактивное программирование на Angular (ngrx) Django DRF Celery и socket.io. Система авторизации по token и отслеживание пользователей онлайн.

В этой статье автор затрагивает темы разработки интерактивной онлайн-системы, на базе одностраничного приложения (SPA), включающей в себя:

- авторизацию пользователей с использованием token-ов из библиотеки Django Rest Framework;
- двухсторонняя клиент-серверная коммуникация с использованием веб-сокет сервера Tornado и python-socketio и ng-socket-io на клиенте;
- учет пользователей онлайн с немедленным оповещением всех авторизованных пользователей;
- использование библиотеки ngrx (REDUX) для работы с хранилищем данных проекта Angular.
- вывод пользователей онлайн из хранилища на страницу с автоматической перерисовкой при авторизации или покиданием сайта участниками, включая закрытие окна браузера.

Блок-схема потоков передачи данных на примере события авторизации из html-формы изображена ниже.



Разберем последовательно каждый шаг этого процесса и приведем ключевые участки кода, описывающие логику нашего приложения.

1. Пользователь заполняет форму с логином и паролем и кликает на кнопку «Вход».

```
<form (submit)="login()">
```

2. Ангуляр-компонент вызывает событие сабмита формы и задействует сервис для отправки данных http-запросом на сервер в теле POST запроса.

```
login(){
  this.loginService.login({
   'username': this.user.username,
   'password': this.user.password});
})

…

this.http.post(`${this.app_config.APIurl}/api-token-auth/`,json_data).subscribe(...)
```

3. После успешной аутентификации сервер возвращает данные пользователя в следующем виде:

{
"user": {"id":255,"username":"admin","email":"admin@gmail.com",.....},
"Token":"1a710265ae00af7f5cdc4016faa89906d42238cb",
"agent":"Mozilla/5.0 (X11; Linux x86_64) Safari/537.36..."
}

При этом автоматически генерируется token, который в последствии сохраняется в localStorage и пробрасывается через заголовки всех последующих http-запросов интерсептором.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
   token: string;
     intercept(req: HttpRequest<any>,
          next: HttpHandler): Observable<HttpEvent<any>> {

      const idToken = localStorage.getItem("access_token");

      if (idToken) {
         const cloned = req.clone({
            headers: req.headers.set("Authorization",
               "Token " + idToken)
         });

         return next.handle(cloned);
      }
      else {
         return next.handle(req);
      }
   }
}
```

Этот класс объявляется в провайдерах главного модуля и цепляется к зарезервированному токену Ангуляра HTTP_INTERCEPTORS, клонируя объект запроса и возвращая новый с измененным хедером.

```
 {
   provide: HTTP_INTERCEPTORS,
   useClass: AuthInterceptor,
   multi: true
 }...
```

Использование Django Rest Framework позволяет прозрачно для разработчика извлекать авторизованного пользователя как и payload во вьюхе примерно так:

```
class AddRoomView(APIView):
   permission_classes = (IsAuthenticated,)
   def post(self, request):
      print(request.data)
      print(request.user)
```

Итак, токен мы получили и сохранили.

```
   this.http.post(`/api-token-auth/`,json_data).subscribe(
     (data: any) => {
           localStorage.setItem('access_token', data['token']);
           this.session_store.dispatch(new sessionActions.LogIn(data));
          }....
```

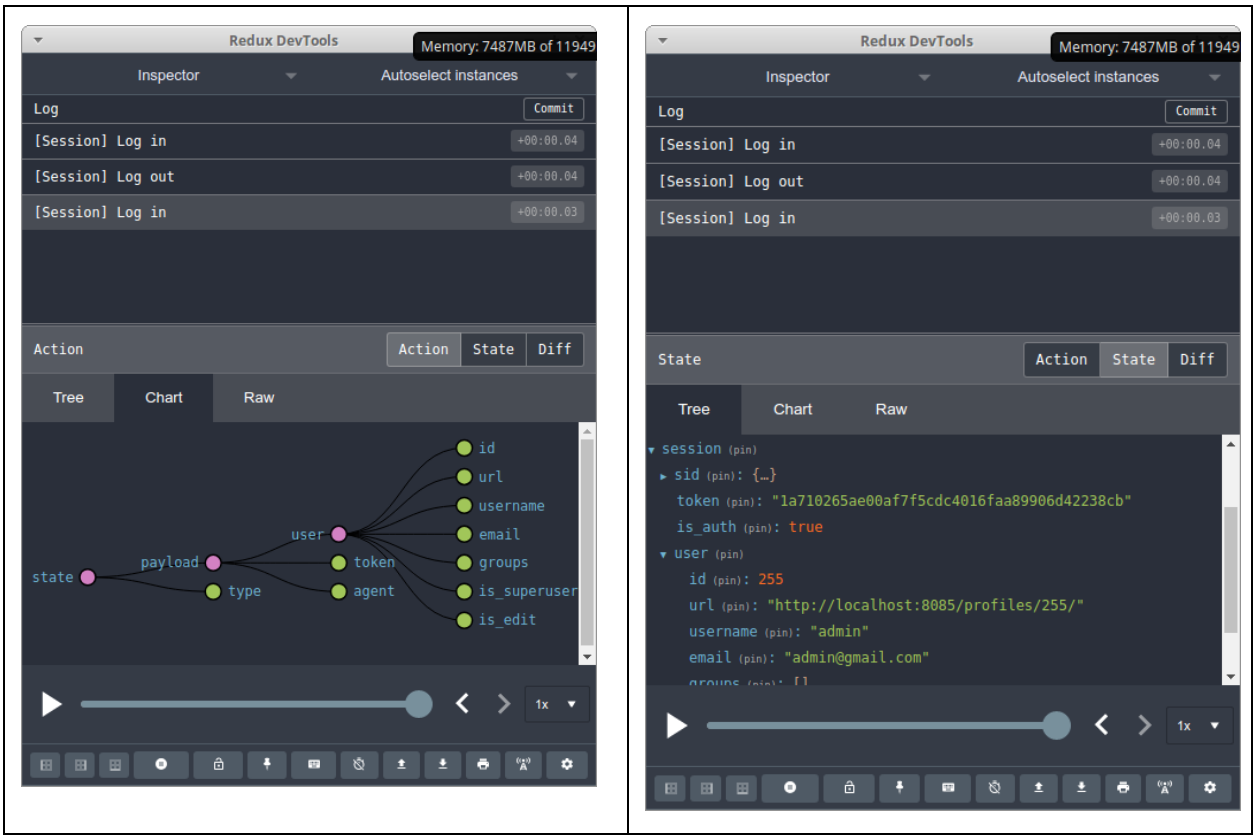
Далее мы диспачим в стору (глобальное хранилище данных всего приложения) ngrx наши данные, создавая действие action с нужным типом и вкладывая в него пейлоад, который поступил с сервера.

В редьюсере сторы мы сохраняем эти данные.

```
    case Actions.ActionTypes.LogIn:

       return {
         ...state,
         token: action.payload.token,
         is_auth: true,
         user: action.payload.user
       };
```

Вот как выглядит результат в дебагере ngrx-devtools.



4. Соединение с сокет-сервером. Инициирует соединение сервис, подключаемый глобально и настраиваемый в главном модуле так:

```
var SOCKET_CONFIG: SocketIoConfig = {
url: 'http://localhost:8888',
options: {path:'/websocket'} };

providers: [
    SocketIoModule.forRoot(SOCKET_CONFIG),
    ….
```

Сам сервис содержит наблюдаемые объекты на события, к которым можно подписаться с любого места, включая наше глобальное хранилище.

```
@Injectable()
export class SocketService {
  user_online$: Observable<any> =
this.socket.fromEvent<string>('server-action:update_user_online');
```

При создании такого объекта мы указываем конструктору тип события в виде строки, (в нашем случае **server-action:update_user_online**), которую будем указывать и на сервере, когда будем эмитить событие питоном примерно так:

```
    if data['task'] == 'user_offline' or data['task'] == 'user_online':
        print('Sending notification about updating of/online')
        await sio.emit('server-action:update_user_online')
```

Кроме предоставления подписных объектов, наш сервис способен подписываться на них сам и испускать события на сервер. Вот как может выглядеть процесс сбора данных о конкретном сокете.

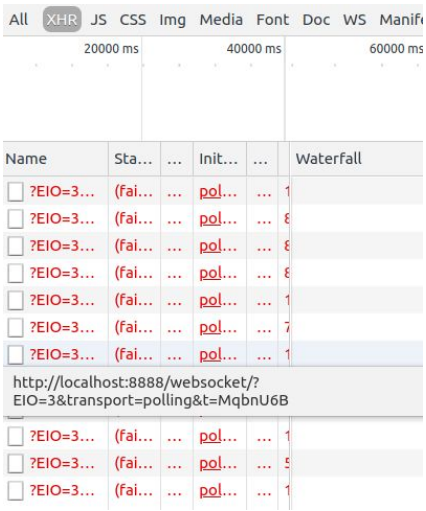
```
ping$: Observable<any> = this.socket.fromEvent<string>('server-action:ping');

  this.ping$.subscribe( (data) => {
    this.socket.emit('ng-action',
      {
        'action': 'pong',
        'socket_id': socket.ioSocket.id,
        'token': localStorage.getItem('access_token')});
  }
  );
```

Теперь если мы на сервере сделаем широковещательный эмит с типом **server-action:ping** например так:

await sio.emit('server-action:ping',{‘message’: ‘I wana know everything of you!’})
То моментально получим на сокет сервера данные со всех подключенных хостов со скоростью несколько тысяч запросов в сек. В вэб-сокетах все очень быстро передается.

После инжектирования сокет-сервиса в компонент или в другой сокет он начинает «ломиться» на сервер по порту и пытаться соединится, буквально зубами уцепиться за соединение и не отпускать. Если он его теряет, то возобновляет попытки, не прекращая раз в 5-6 сек.



Теперь поговорим о сокет-сервере и его роли, которых собственно три:

1. Принимать сообщения от REDIS сервера.
2. Принимать сообщения от Ангуляра (сокет соединения).
3. Реагировать на пришедшие сообщения эмитом событий.

REDIS сервер служит для хранения и передачи быстрых сообщений между разнородными системами. Его еще называют брокером сообщений. В нашем случае этими системами будут сокет-сервер, Django приложение и сервер отложенных задач Celery. Также эта база данных позволяет подписывать своих клиентов на каналы сообщений и передавать их по мере поступления, чем мы и воспользуемся, так как этот процесс просто до ужаса быстр.

Если нам необходимо в Django view запустить какие-либо долгие процессы, результаты которых не влияют на результат ответа, то чтобы не заставлять пользователя ждать, мы формируем задачу для Celery и запускаем ее отложено, например так:

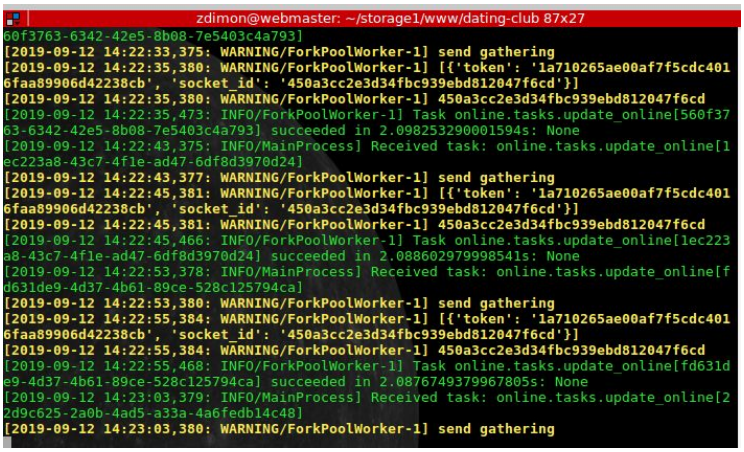
set_user_online.delay(user)

При этом пользователь сразу получает что хотел, а задача начинает выполняться в отдельном процессе (celery worker), который можно запустить такой командой в консоле:

celery -A backend worker -l info

После чего этот вокер подключается к серверу, где хранятся задачи (это может быть любая БД) и последовательно выполняет все, что туда приходит через вызов task_func.delay().

Вот как может выглядеть этот процесс:



Функцию, которую мы хотим передать celery, декорируем декоратором @task.

```
redis_client = redis.Redis(host='localhost', port=6379, db=4)
@task
def set_user_online(user):
    user.is_online = True
    user.save()
    redis_client.publish('notifications',json.dumps({'task': 'user_offline'}))
```

В этом примере мы отмечаем флаг пользователю и отправляем сообщение в редис-сервер в именной канал **notifications** базы данных с номером 4 (по умолчанию их 15).

Это сообщение можно автоматически получать всеми, кто подписан на данный канал. Подпишем на него наш сокет-сервер, чтобы отправить всем уведомление о том, что кто-то зашел на сайт.

```
async def consumer(channel):
    while await channel[0].wait_message():
        msg = await channel[0].get()
        data = json.loads(msg.decode("utf-8"))
        print('Message from redis %s' %  data['task'])
        if data['task'] == 'user_offline' or data['task'] == 'user_online':
            print('Sending notification about updating of/online')
            await sio.emit('server-action:update_user_online')

    ...

async def setup():
```

```
    connection = await aioredis.create_redis('redis://localhost')
    channel = await connection.subscribe('notifications')
    asyncio.ensure_future(consumer(channel))

    ...

if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    loop = IOLoop.current()
    loop.add_callback(setup)
    loop.start()
```

Наше сообщение имеет ключ task, по которому мы определяем что нужно делать.
Если на нужно отработать событие из Angular, то мы декорируем функцию так:

```
@sio.on('ng-action')
async def chat_message(sid, data):
    if data['action'] == 'pong':
        sio.current_connections.append({'socket_id':data['socket_id'],'token': data['token']})
        redis_client.set('socket_connections',json.dumps(sio.current_connections))
```

В данном примере мы отрабатываем событие **ng-action**, вызванное в сокет-сервисе ангуляра (см. выше)

```
    this.socket.emit('ng-action',
      …..
```

Определяем по ключу action, тип действия и принимаем переданные данные. Сохраняем их в объекте текущего сокет-соединения. Их создается по одному на каждую страницу браузера, на которой это соединение происходит. Осталось подписать компонент Ангуляр на хранилище и вывести все в шаблоне.

Для того, чтобы из ngrx-хранилища выбрать участок, используется селектор. Так как наше хранилище имеет корневые узлы и дочерние (корневые можно сравнить с таблицами базы данных) то и классов селекторов два, один для корневых – createFeatureSelector. Второй для дочерних – createSelector.

Пример двух селекторов:

```
export const getOnlineStateSelector = createFeatureSelector<OnlineState>('online');

 export const selectUsersList = createSelector(
```

```
  getOnlineStateSelector,
  (state: OnlineState) => state.users
 );
```

Интерфейс хранилища OnlineState:

```
import { User } from '../users/users.store';

export interface OnlineState {
 users_ids: number[];
 users:  {[id: number]: User};
}
```

Выборка по селектору в компоненте очень проста:

```
this.online = this.online_store.select(selectUsersList);
```

Естественно, не забываем включить наше хранилище в конструкторе компонента:

```
 constructor(
  private online_store: Store<OnlineState>,    ) {}
```

После чего на него можно подписаться потоком async в шаблоне:

```
<div class="card"
        *ngFor="let user of online | async"
        [user]="user"
        (call)="selectUser($event)"
        app-user-online-item>
</div>
```

Выводы

В статье кратко рассмотрены принципы построения реактивных приложений на базе Angular. Применение паттерна REDUX при проектировании хранилища состояний приложения. Коммуникация компонентов системы при помощи веб-сокетов. Организация системы отложенных задач celery.