

Реализовать аутентификацию пользователя в приложении Django & React с помощью Knox

Добавьте аутентификацию на основе токенов с помощью Django-rest-knox в приложение, созданное с помощью Django и React / Redux



Коджи Мочизуки

27 июля 2019 г. · 8 минут чтения ★



Фото Джона Мура на Unsplash

В последнем уроке мы создали приложение CRUD Todo с React, Redux и Django (каркас REST). На этот раз мы реализуем аутентификацию пользователя, такую как Логин, для приложения. Опять же, мы используем Semantic UI для дизайна пользовательского интерфейса.

Username

Email

Password

Confirm Password

Register

Already have an account? [Login](#)

регистр

TodoCRUD

Home

Sign Up

Login

Username

Password

Login

Don't have an account? [Register](#)

Авторизоваться

TodoCRUD

Home

John ▾

Logout

Task

Add

📅

[John's Task 1](#)
07/15/2019 09:05:25

Delete

📅

[John's Task 2](#)
07/15/2019 09:05:39

Delete

Выйти

Содержание

- Добавление владельца Todo
- Создание нового приложения для аутентификации пользователя
- Тестирование аутентификации с использованием клиента REST
- Реализация загрузки пользователя и входа в систему
- Создание формы входа
- Создание PrivateRoute
- Реализация выхода
- Добавление элементов в панель навигации
- Реализация Реестра
- Создание формы регистрации
- Обновление URLconf

. . .

ПОДГОТОВКА

Первое, что нам нужно сделать, это подготовить базовое приложение. Если вы хотите создать базовое приложение с нуля, начните с учебника ниже:

Создайте приложение CRUD Todo с помощью Django и React / Redux

В этом уроке мы узнаем, как создать приложение CRUD Todo с каркасом REST Django для серверной части, React...

medium.com

Вы также можете скачать базовое приложение из моего репозитория на GitHub .

Если вы клонировали репозиторий на своем компьютере, инициализируйте свой локальный репозиторий:

```
$ rm -rf .git
$ git init
```

Установите пакеты:

```
$ pipenv install
$ npm install
```

Запустите миграцию и запустите сервер dev:

```
$ pipenv shell
$ python manage.py migrate
$ python manage.py runserver
```

Откройте другой терминал и запустите скрипт:

```
$ npm run dev
```

Посетите <http://127.0.0.1:8000/> с вашим браузером. Если отображается форма создания Todo, базовое приложение готово.

Добавление владельца Todo

Мы добавим `owner` поле к `Todo` модели и свяжем это поле с `User` моделью, которую предоставляет Django:

```
1  # todos / models.py
2
3  из Django . db импорт моделей
4  из Django . вно . авт . добавлен импорт моделей # Пользователь
5
6
7  Класс Todo ( модели . Модель ):
8      задача = модели . CharField ( max_length = 255 )
9      владелец = модели . ForeignKey (
10         Пользователь , related_name = "todos" , on_delete = models . CASCADE , null = 1
```

```
13     def __str__ ( самостоятельно ) :
14         вернуть себя . задача
```

models.py с ❤️ на GitHub

[просмотреть raw](#)

Примените изменения к нашим базам данных, выполнив следующие команды:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Далее мы будем устанавливать разрешения на `TodoViewSet` . И определите метод, чтобы получить только задачи, созданные каждым владельцем. Откройте `todos/api/views.py` файл:

```
1  # todos / api / views.py
2
3  из rest_framework импортировать наборы представлений , разрешения # добавленные ра
4
5  от . импорт сериализаторов TodoSerializer
6  # из todos.models import Todo # удалить
7
8
9  Класс TodoViewSet ( viewsets . ModelViewSet ) :
10     # queryset = Todo.objects.all () # удалить
11     serializer_class = TodoSerializer
12     access_classes = [ права доступа . IsAuthenticated ] # добавлено
13
14     def get_queryset ( self ) : # добавлено
15         вернуть себя . запрос . пользователь . Todos . все ()
16
17     def execute_create ( self , serializer ) : # добавлено
18         сериализатор . сохранить ( владелец = сам . запрос . пользователь )
```

views.py с ❤️ на GitHub

[просмотреть raw](#)

Наш сериализатор теперь будет иметь 'owner' поле, переопределяя `perform_create()` метод, как указано выше.

Создание нового приложения для аутентификации пользователя

Мы добавим новое приложение для аутентификации пользователя, выполнив команду ниже:

```
Аккаунты $ python manage.py startapp
```

Мы установим новый пакет для использования Token Authentication:

```
$ pipenv install django-rest-knox
```

Откройте `settings.py` файл и включите их:

```
1  # todocrud / settings.py
2
3  INSTALLED_APPS = [
4      'accounts.apps.AccountsConfig' ,    # добавлено
5      'frontend.apps.FrontendConfig' ,
6      'todos.apps.TodosConfig' ,
7      'rest_framework' ,
8      'knox' ,    # добавлено
9      # ...
10 ]
11
12 REST_FRAMEWORK = {
13     # 'DEFAULT_PERMISSION_CLASSES': [# удалить
14     # 'rest_framework.permissions.AllowAny'
15     #],
16     'DEFAULT_AUTHENTICATION_CLASSES' : (    # добавлено
17         'knox.auth.TokenAuthentication' ,
18     ),
19     «DATETIME_FORMAT» : «% m /% d /% Y% H:% M:% S» ,
20 }
```

`settings.py`, размещенный на  на GitHub

[просмотреть raw](#)

Мы можем устанавливать разрешения отдельно для каждого просмотра, поэтому удалите политику разрешений по умолчанию, установленную глобально. Мы установим `DEFAULT_AUTHENTICATION_CLASSES` настройку вместо этого.

Теперь, когда **Knox** добавлен, давайте запустим миграцию:

```
$ python manage.py migrate
```

Создание API

Сначала мы создадим новую папку с именем `api` и создадим в ней четыре файла:

```
account /
api /
    __init__.py
    serializers.py
urls.py
views.py
```

Давайте создадим каждый файл.

сериализаторы

```
1  # account / api / serializers.py
2
3  из Django . вно . авторизация импорт Аутентифицировать
4  из Django . вно . авт . Пользователь импорта моделей
5
6  из rest_framework импортировать сериализаторы
7
8  Пользователь . _мета . get_field ( 'электронная почта' ). _unique = True
9
10
11  Класс UserSerializer ( сериализаторы . ModelSerializer ):
12      класс Meta :
13          модель = пользователь
14          fields = ( 'id' , 'username' , 'email' )
15
16
17  Класс RegisterSerializer ( сериализаторы . ModelSerializer ):
18      класс Meta :
19          модель = пользователь
20          fields = ( 'id' , 'username' , 'email' , 'password' )
21          extra_kwargs = { 'password' : { 'write_only' : True }}
22
23      def create ( self , validated_data ):
24          пользователь = пользователь . объекты . create user (
```

```

27         validated_data [ 'пароль' ]
28     )
29     возвратный пользователь
30
31
32 Класс LoginSerializer ( serializers . Сериализатор ):
33     имя пользователя = serializers . CharField ()
34     пароль = serializers . CharField ()
35
36     def validate ( self , data ):
37         пользователь = аутентификация ( ** данные )
38         если пользователь и пользователь . is_active :
39             возвратный пользователь
40         поднять serializers . ValidationError ( «Неверные учетные данные» )

```

serializers.py размещенного с помощью ❤️ GitHub

[просмотреть сырой](#)

Я хотел бы установить, `User._meta.get_field('email')._unique = True` чтобы дублирующие адреса электронной почты не могли быть зарегистрированы.

APIViews

```

1  # account / api / views.py
2
3  из rest_framework импортировать дженерики , разрешения
4  из rest_framework . ответ импорта Ответ
5
6  из knox . импорт моделей AuthToken
7
8  от . импорт сериализаторов UserSerializer , RegisterSerializer , LoginSerializer
9
10
11 Класс UserAPIView ( обобщения . RetrieveAPIView ):
12     access_classes = [
13         разрешения . IsAuthenticated ,
14     ]
15     serializer_class = UserSerializer
16
17     def get_object ( self ):
18         вернуть себя . запрос . пользователь
19
20
21 Класс RegisterAPIView ( generics . GenericAPIView ):
22     serializer_class = RegisterSerializer
23
24

```



```

25     сериализатор = сам . get_serializer ( data = request . data ,
26     сериализатор . is_valid ( повышение_эксклюзии = True )
27     пользователь = сериализатор . сохранить ()
28     обратный ответ ({
29         «пользователь» : UserSerializer ( пользователь , context = self . get_serializer
30         "token" : AuthToken . объекты . создать ( пользователь ) [ 1 ]
31     })
32
33
34     Класс LoginAPIView ( дженерики . GenericAPIView ):
35         serializer_class = LoginSerializer
36
37     пост def ( self , request , * args , ** kwargs ):
38         сериализатор = сам . get_serializer ( data = request . data )
39         сериализатор . is_valid ( повышение_эксклюзии = True )
40         пользователь = сериализатор . validated_data
41         обратный ответ ({
42             «пользователь» : UserSerializer ( пользователь , context = self . get_serializer
43             "token" : AuthToken . объекты . создать ( пользователь ) [ 1 ]
44         })

```

"token": AuthToken.objects.create(user)[1] означает, что AuthToken.objects.create() возвращает кортеж (экземпляр, токен) . И так, добавьте [1] и укажите вторую позицию.

URL-адрес

```

1  # account / api / urls.py
2
3  из Django . URL пути импорта , включают
4
5  из knox . просмотр импорта LogoutView
6
7  от . импорт просмотров UserAPIView , Регистрация APIView , Логин APIView
8
9  urlpatterns = [
10     путь ( ' ' , include ( 'knox.urls' ) ),
11     путь ( «пользователь» , UserAPIView . as_view () ),
12     путь ( 'регистрация' , RegisterAPIView . as_view () ),
13     путь ( 'login' , LoginAPIView . as_view () ),
14     путь ( 'logout' , LogoutView . as_view () , name = 'knox_logout' )
15 ]

```

Чтобы выйти из системы, используйте представление, предоставленное **Knox** .

URL проекта

Наконец, мы включим URL-адреса учетных записей в URLconf проекта:

```
1  # todocrud / urls.py
2
3  из Django . вно импорт админ
4  из Django . URL пути импорта , включают
5
6  urlpatterns = [
7     путь ( '' , include ( 'frontend.urls' )),
8     путь ( 'api /' , include ( 'todos.api.urls' )),
9     путь ( 'api / auth /' , include ( 'accounts.api.urls' )), # добавлено
10    путь ( 'admin /' , admin . site . urls ),
11 ]
```

urls.py с with на GitHub

[просмотреть raw](#)

Теперь, когда мы внесли изменения в модель User, запустите миграцию:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Тестирование аутентификации с использованием клиента REST

Что такое REST Client ?

REST Client позволяет отправлять HTTP-запрос и просматривать ответ непосредственно в коде Visual Studio.

Если вы не используете **VS Code** , вы можете использовать Почтальон .

В этом руководстве мы будем использовать **REST-клиент** для проверки подлинности.

Прежде всего, мы создадим новую папку с именем `requests` в корневом каталоге проекта и создадим файл с именем `api.http` в ней:

запросы /
api.http

регистр

Давайте сначала создадим пользователя. Запишите в api.http файле следующее. Затем «Отправить запрос» будет отображаться над методом HTTP. Нажмите здесь, чтобы посмотреть ответ:

```
1  # запросы / api.http
2
3  # Послать запрос
4  POST http: // 127.0 . 0.1 : 8000 / api / auth / register
5  Тип контента: приложение / JSON
6
7  {
8      " имя пользователя " : " Джон " ,
9      " электронная почта " : " john@example.com " ,
10     " пароль " : " 1234 "
11 }
```

api.http, размещенного на ❤️ в GitHub

[просмотр raw](#)

```
{
  "user": {
    "id": 1,
    "username": "John",
    "email": "john@example.com"
  },
  "token":
  "5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04"
}
```

Если ответ выглядит как выше, это сработало.

Загрузить пользователя

Чтобы написать несколько запросов в одном файле, поместите три или более подряд в # качестве разделителей между запросами. Напишите следующим образом и отправьте запрос GET с токеном:

```
1  # запросы / api.http
```

```
4  ПОЛУЧИТЕ http: // 127.0 . 0.1 : 8000 / api / auth / user
5  Авторизация: токен 5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04
6
7  ###
8
9  # Послать запрос
10 POST http: // 127.0 . 0.1 : 8000 / api / auth / register
11 Тип контента: приложение / JSON
12
13 {
14     " имя пользователя " : " Джон " ,
15     " электронная почта " : " john@example.com " ,
16     " пароль " : " 1234 "
17 }
```

api.http, размещенного на  в GitHub

[просмотр raw](#)

```
{
  "id": 1,
  "username": "John",
  "email": "john@example.com"
}
```

Данные пользователя должны быть возвращены.

Выйти

```
1  # запросы / api.http
2
3  # Послать запрос
4  POST http: // 127.0 . 0.1 : 8000 / api / auth / logout
5  Авторизация: токен 5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04
```

api.http, размещенного на  в GitHub

[просмотр raw](#)

Ответа нет, конечно. Если запрос проходит, токен, используемый для аутентификации, удаляется из системы и больше не может использоваться.

Попробуйте отправить запрос, чтобы снова получить данные пользователя. Ошибка должна быть возвращена следующим образом:

```
{
  "detail": "Неверный токен."
}
```

Войти с неверными данными

Давайте проверим аутентификацию входа в систему. Сначала мы отправим неверные данные. Ошибка должна быть возвращена:

```
1  # запросы / api.http
2
3  # Послать запрос
4  POST http: // 127.0 . 0.1 : 8000 / api / auth / login
5  Тип контента: приложение / JSON
6
7  {
8      " username " : " Mary " ,
9      " пароль " : " 1234 "
10 }
```

api.http, размещенного на ❤️ в GitHub

[просмотр raw](#)

```
{
  "non_field_errors": [
    "Неверные учетные данные"
  ]
}
```

Войти с правильными данными

Далее мы вышлем правильные данные:

```
1  # запросы / api.http
2
3  # Послать запрос
4  POST http: // 127.0 . 0.1 : 8000 / api / auth / login
5  Тип контента: приложение / JSON
6
7  {
8      " имя пользователя " : " Джон " ,
9      " пароль " : " 1234 "
10 }
```

```
{
  "user": {
    "id": 1,
    "username": "John",
    "email": "john@example.com"
  },
  "token":
  "055eb2ea1bf931039433dbd2030f6b9075e067c81e0eda57ca383880244a7015"
}
```

Ответ должен быть возвращен, как указано выше.

Наконец, попробуйте отправить запрос, чтобы получить пользовательские данные с вновь созданным токеном.

Реализация загрузки пользователя и входа в систему

Давайте сначала добавим все типы действий, необходимые для аутентификации пользователя. Откройте `types.js` файл и добавьте их:

```
1 // внешний интерфейс / src / actions / types.js
2
3 export const USER_LOADING = 'USER_LOADING' ; // добавлено
4 export const USER_LOADED = 'USER_LOADED' ; // добавлено
5 export const AUTH_ERROR = 'AUTH_ERROR' ; // добавлено
6 export const REGISTER_SUCCESS = 'REGISTER_SUCCESS' ; // добавлено
7 export const REGISTER_FAIL = 'REGISTER_FAIL' ; // добавлено
8 экспорт const LOGIN_SUCCESS = 'LOGIN_SUCCESS' ; // добавлено
9 экспорт const LOGIN_FAIL = 'LOGIN_FAIL' ; // добавлено
10 экспорт const LOGOUT_SUCCESS = 'LOGOUT_SUCCESS' ; // добавлено
```

`types.js`, размещенный на ❤️ на GitHub

[просмотреть raw](#)

Создатели действий

Мы создадим новый файл с именем `auth.js` в `actions` каталоге и определим двух создателей действий:

```
1 // внешний интерфейс / src / actions / auth.js
2
```

```

4   import { useState, useContext } from 'react';
5
6   import {
7     USER_LOADING,
8     USER_LOADED,
9     AUTH_ERROR,
10    LOGIN_SUCCESS,
11    НЕВЕРНЫЙ ЛОГИН
12  } из './types';
13
14  // ЗАГРУЗИТЬ ПОЛЬЗОВАТЕЛЯ
15  export const loadUser = ( ) => async ( dispatch, getState ) => {
16    рассылка ( { тип : USER_LOADING } );
17
18    попытаться {
19      const res = ждать axios . get ( '/ api / auth / user', tokenConfig ( getState ) );
20      отправка ( {
21        тип : USER_LOADED,
22        полезная нагрузка : рез . данные
23      } ) ;
24    } catch ( err ) {
25      отправка ( {
26        тип : AUTH_ERROR
27      } ) ;
28    }
29  } ;
30
31  // LOGIN USER
32  export const login = ( { имя пользователя , пароль } ) => асинхронная отправка
33    // Заголовки
34    const config = {
35      заголовки : {
36        'Content-Type' : 'application / json'
37      }
38    } ;
39
40    // Тело запроса
41    const body = JSON . stringify ( { имя пользователя , пароль } ) ;
42
43    попытаться {
44      const res = ждать axios . post ( '/ api / auth / login', body , config ) ;
45      отправка ( {
46        тип : LOGIN_SUCCESS,
47        полезная нагрузка : рез . данные
48      } ) ;
49    } catch ( err ) {

```

```

52     } ) ;
53     отправка ( stopSubmit ( 'LoginForm' , заблуждается . ответ . данные ) ) ;
54 }
55 } ;
56
57 // вспомогательная функция
58 export const tokenConfig = getState => {
59     // Получить токен
60     const token = getState ( ) . авт . жетон ;
61
62     // Заголовки
63     const config = {
64         заголовки : {
65             'Content-Type' : 'application / json'
66         }
67     } ;
68
69     if ( token ) {
70         конфиг . headers [ 'Authorization' ] = `Token $ { token } ` ;
71     }
72
73     возврат конфигурации ;
74 } ;

```

Создайте функцию с именем `tokenConfig` вспомогательной функции, которая получает и устанавливает токены. Эта функция также используется для создателей действий `todo`.

Мы можем использовать `stopSubmit()` для передачи ошибок на стороне сервера в наши поля формы `Redux`. `loginForm` Будет создан позже. Не волнуйтесь об этом сейчас.

Нам также нужно передать токен в действия `todo`. Откройте `actions/todos.js` файл и обновите каждого создателя действия:

```

1 // фронтенд / src / actions / todos.js
2
3 import { tokenConfig } из './auth' ; // добавлено
4
5 // ПОЛУЧИТЬ ТОДОС
6 export const getTodos = ( ) => async ( dispatch , getState ) => {
7     const res = ждать axios . get ( '/ api / todos /' , tokenConfig ( getState ) ) ;

```



```

10
11 // ПОЛУЧИТЬ ТОДО
12 экспорт const getTodo = id => async ( dispatch , getState ) => {
13   const res = ждать axios . get ( `/ api / todos / $ { id } /` , tokenConfig ( get
14   // ...
15 } ;
16
17 // ДОБАВИТЬ ТОДО
18 экспорт const addTodo = formValues => async ( dispatch , getState ) => {
19   const res = ждать axios . пост (
20     '/ api / todos /' ,
21     { ... formValues } ,
22     tokenConfig ( getState )
23   ) ;
24   // ...
25 } ;
26
27 // УДАЛИТЬ ТОДО
28 экспорт const deleteTodo = id => async ( dispatch , getState ) => {
29   жду аксиос . delete ( `/ api / todos / $ { id } /` , tokenConfig ( getState ) ) ;
30   // ...
31 } ;
32
33 // РЕДАКТИРОВАТЬ ТОДО
34 экспорт const editTodo = ( id , formValues ) => async ( dispatch , getState )
35   const res = ждать axios . патч (
36     `/ api / todos / $ { id } /` ,
37     formValues ,
38     tokenConfig ( getState )
39   ) ;
40   // ...
41 } ;

```

Переходники

Далее мы создадим новый файл с именем `auth.js` в `reducers` каталоге и напишем редуктор аутентификации:

```

1 // frontend / src / redurs / auth.js
2
3 import {
4   USER_LOADING ,
5   USER_LOADED ,
6   AUTH_ERROR

```

```

8     ПЕРЕДПОВИДЬ ЛОГІНІ
9 } из '../actions/types' ;
10
11 const initialState = {
12     isLoading : false ,
13     isAuthenticated : null ,
14     пользователь : null ,
15     токен : localStorage . getItem ( 'токен' )
16 } ;
17
18 функция экспорта по умолчанию ( state = initialState , action ) {
19     switch ( action . type ) {
20         case USER_LOADING :
21             возврат {
22                 ... состояние ,
23                 isLoading : правда
24             } ;
25         case USER_LOADED :
26             возврат {
27                 ... состояние ,
28                 isLoading : false ,
29                 isAuthenticated : правда ,
30                 пользователь : действие . полезная нагрузка
31             } ;
32         case LOGIN_SUCCESS :
33             localStorage . setItem ( 'token' , action . payload . token ) ;
34             возврат {
35                 ... состояние ,
36                 isLoading : false ,
37                 isAuthenticated : правда ,
38                 ... действие . полезная нагрузка
39             } ;
40         case AUTH_ERROR :
41         case LOGIN_FAIL :
42             localStorage . removeItem ( 'токен' ) ;
43             возврат {
44                 ... состояние ,
45                 isLoading : false ,
46                 isAuthenticated : ложь ,
47                 пользователь : null ,
48                 токен : ноль
49             } ;
50         по умолчанию :
51             возвратное состояние ;
52     }
53 }

```

Токены хранятся в веб-браузере с использованием `localStorage` свойства.

Узнайте больше о `localStorage` .

Затем мы добавим аутентификатор `auth` к родительскому редуктору. Откройте `reducers/index.js` файл и обновите его следующим образом:

```
1 // frontend / src / redurs / index.js
2
3 импортировать аутентификацию из './auth' ; // добавлено
4
5 экспорт по умолчанию combReducers ( {
6   form : formReducer ,
7   Тодо ,
8   auth // добавлено
9 } ) ;
```

`index.js`, размещенный на ❤️ на GitHub

[просмотреть raw](#)

Создание формы входа

Давайте создадим новый компонент для входа, используя **Redux Form** .

Создайте новую папку с именем `auth` в `components` каталоге и добавьте `LoginForm.js` в нее новый файл :

```
1 // frontend / src / components / auth / LoginForm.js
2
3 импортировать React , { Component } из 'response' ;
4 импорт { Ссылка , Перенаправление } от «реагировать-маршрутизатор-дом» ;
5 импорта { подключения } от 'реагируют-Redux' ;
6 import { Field , reduxForm } из 'redux-формы' ;
7 import { login } из '../actions/auth' ;
8
9 Класс LoginForm расширяет Компонент {
10   renderField = ( { input , label , type , meta : { touch , error } } ) => {
11     возврат (
12       < DIV имя класса = { `поле $ { прикоснулся && ошибку ? 'error' : '' } ` } >
13         < label > { label } < / label >
14         < input { ... input } type = { type } / >
15         { тронут && ошибка && (
16           < span className = 'интерфейс, указывающий на красную базовую метку' > { err
17         ) }
18       < / div >
```

```

21
22 hiddenField = ( { тип , мета : { ошибка } } ) => {
23     возврат (
24         < div className = 'field' >
25             < input type = { type } / >
26             { error && < div className = 'ui red message' > { error } < / div > }
27         < / div >
28     ) ;
29 } ;
30
31 onSubmit = formValues => {
32     это . реквизит . логин ( formValues ) ;
33 } ;
34
35 render ( ) {
36     если ( это . реквизит . isAuthenticated ) {
37         return < Redirect to = '/' / > ;
38     }
39     возврат (
40         < div className = 'ui container' >
41             < div className = 'UI сегмент' >
42                 < форма
43                     onSubmit = { это . реквизит . handleSubmit ( this . onSubmit ) }
44                     className = 'UI Form '
45                 >
46                     < Поле
47                         name = 'username'
48                         type = 'text'
49                         компонент = { это . renderField }
50                         label = 'Имя пользователя'
51                     / >
52                     < Поле
53                         имя = «пароль»
54                         type = 'password'
55                         компонент = { это . renderField }
56                         label = 'Password'
57                     / >
58                     < Поле
59                         name = 'non_field_errors'
60                         type = 'hidden'
61                         компонент = { это . hiddenField }
62                     / >
63                     < button className = 'основная кнопка пользовательского интерфейса' > Войти
64                 < / form >
65                 < p style = { { marginTop : '1rem' } } >
66                     У вас нет аккаунта? < link to = '/' registerView > Зарегистрироваться < / link >

```

```

68         < / div >
69     < / div >
70 ) ;
71 }
72 }
73
74 const mapStateToProps = state => ( {
75     isAuthenticated : состояние . авт . isAuthenticated
76 } ) ;
77
78 LoginForm = подключить (
79     mapStateToProps ,
80     { логин }
81 ) ( LoginForm ) ;
82
83 экспорт по умолчанию reduxForm ( {
84     форма : 'loginForm'
85 } ) ( LoginForm ) ;

```

Если username и password не совпадают с информацией в базе данных, Django возвращает **ошибки, не связанные с полем**. Чтобы отобразить эту ошибку, нам нужно иметь поле с именем 'non_field_errors'.

'loginForm' это название этой формы. Вы можете назвать каждую форму, как вам нравится.

Создание PrivateRoute

Рекомендуется перенаправлять пользователей, которые не вошли на страницу входа, когда они посещают домашнюю страницу. Создайте новую папку с именем common в components каталоге и добавьте PrivateRoute.js в нее новый файл. А затем напишите код следующим образом:

```

1 // frontend / src / components / common / PrivateRoute.js
2
3 импорт React из 'реакции' ;
4 импорта { маршрута , перенаправление } от 'реагируют-маршрутизатор-DOM' ;
5 импорта { подключения } от 'реагируют-Redux' ;
6
7 const PrivateRoute = ( { component : Component , auth , ... rest } ) => (
8     < Маршрут
9     { ... отдых }

```

```

12     return < div > Загрузка ... < / div > ;
13   } else if ( ! auth . isAuthenticated ) {
14     return < Redirect to = '/ login' / > ;
15   } еще {
16     return < Component { ... props } / > ;
17   }
18 } }
19 / >
20 ) ;
21
22 const mapStateToProps = state => ( {
23   auth : гос . авт
24 } ) ;
25
26 экспортировать соединение по умолчанию ( mapStateToProps ) ( PrivateRoute ) ;

```

PrivateRoute.js, размещенного на with GitHub

[просмотр raw](#)

Для получения информации о render функции поддержки см. Следующий URL:
<https://reacttraining.com/react-router/web/api/Route/render-func>

Давайте откроем App.js файл и внесем некоторые изменения:

```

1  // внешний интерфейс / src / components / App.js
2
3  импортировать LoginForm из './auth/LoginForm' ; // добавлено
4  импортировать PrivateRoute из './common/PrivateRoute' ; // добавлено
5
6  import { loadUser } из '../actions/auth' ; // добавлено
7
8  Класс App расширяет Компонент {
9    // добавлено
10   componentDidMount ( ) {
11     магазин . рассылка ( loadUser ( ) ) ;
12   }
13
14   render ( ) {
15     возврат (
16       < Provider store = { store } >
17       < История маршрутизатора = { история } >
18         < Заголовок / >
19         < Switch >
20           < PrivateRoute точный путь = '/' component = { Dashboard } / > // обновл
21           < Route точный путь = '/ delete /: id' component = { TodoDelete } / >
22           < Route точный путь = '/ edit /: id' component = { TodoEdit } / >

```

```

25         < / Router >
26     < / Provider >
27   ) ;
28 }
29 }

```

App is connected with us GitHub

PROMOTOTE FOR

Теперь мы можем войти в систему. Посетите <http://127.0.0.1:8000/> с помощью своего браузера:

The screenshot shows the application interface on the left and the Redux DevTools state on the right. The application has a header with 'TodoCRUD' and 'Home' links. The main form has 'Username' and 'Password' input fields, a 'Login' button, and a link to 'Register'. The Redux DevTools state shows the following structure:

```

{
  form (pin): { loginForm: { ... } },
  todos (pin): { },
  auth (pin): {
    isLoading (pin): false,
    isAuthenticated (pin): false,
    user (pin): null,
    token (pin): null
  }
}

```

Форма входа

Вы должны быть перенаправлены на страницу входа. Это сработало?

The screenshot shows the application interface on the left and the Redux DevTools state on the right. The application has a header with 'TodoCRUD' and 'Home' links. The main form has 'Username' and 'Password' input fields, a 'Login' button, and a link to 'Register'. The 'Username' field contains the text 'Mary'. The 'Password' field is masked with dots. A red error message 'Incorrect Credentials' is displayed below the password field. The Redux DevTools state shows the following structure:

```

{
  form (pin): { loginForm: { ... } },
  todos (pin): { },
  auth (pin): {
    isLoading (pin): false,
    isAuthenticated (pin): false,
    user (pin): null,
    token (pin): null
  }
}

```

Неполевые ошибки также отображаются правильно.

Наконец, попробуйте войти в систему, используя учетную запись, которую мы создали во время проверки подлинности. А затем попробуйте добавить пару задач.

Реализация выхода

Создатели действий

Откройте `actions/auth.js` файл и добавьте нового создателя действий для выхода из системы:

```
1  // внешний интерфейс / src / actions / auth.js
2
3  // добавлен LOGOUT_SUCCESS
4  import {
5    USER_LOADING ,
6    USER_LOADED ,
7    AUTH_ERROR ,
8    LOGIN_SUCCESS ,
9    LOGIN_FAIL ,
10   LOGOUT_SUCCESS
11 } из './types' ;
12
13 // LOGOUT USER
14 export const logout = ( ) => async ( dispatch , getState ) => {
15   жду axios . post ( '/ api / auth / logout' , null , tokenConfig ( getState ) ) ;
16   отправка ( {
17     тип : LOGOUT_SUCCESS
18   } ) ;
19 } ;
```

`auth.js`, размещенного на ❤️ в GitHub

[просмотр raw](#)

Переходники

Откройте `reducers/auth.js` файл и добавьте действие в редуктор:

```
1  // frontend / src / redurs / auth.js
2
```



```

5  USER_LOADING ,
6  USER_LOADED ,
7  AUTH_ERROR ,
8  LOGIN_SUCCESS ,
9  LOGIN_FAIL ,
10 LOGOUT_SUCCESS
11 } из '../actions/types' ;
12
13 функция экспорта по умолчанию ( state = initialState , action ) {
14   switch ( action . type ) {
15     // ...
16     case AUTH_ERROR :
17     case LOGIN_FAIL :
18     case LOGOUT_SUCCESS : // добавлено
19       localStorage . removeItem ( 'токен' ) ;
20       возврат {
21         ... состояние ,
22         isLoading : false ,
23         isAuthenticated : ложь ,
24         пользователь : null ,
25         токен : ноль
26       } ;
27       // ...
28     }
29   }

```

auth.js, размещенного на ❤️ в GitHub

[просмотр raw](#)

Теперь мы можем выйти, но есть одна проблема. Например, если вы входите и выходите из **учетной записи 1**, а затем входите в нее с **учетной записью 2**, объекты задач для **учетной записи 1** также отображаются в списке задач для **учетной записи 2**. Это потому, что редуктор todo не инициализирован.

Чтобы решить эту проблему, настройте корневой редуктор следующим образом:

```

1  // frontend / src / reducers / index.js
2
3  импортировать { LOGOUT_SUCCESS } из '../actions/types' ; // добавлено
4
5  // экспорт по умолчанию combineReducers ({
6  //   form: formReducer,
7  //   задачи,
8  //   аутентификация

```

```

11  const appReducer = combReducers ( {
12    form : formReducer ,
13    Тодо ,
14    авт
15  } ) ;
16
17  const rootReducer = ( состояние , действие ) => {
18    if ( action . type === LOGOUT_SUCCESS ) {
19      состояние = не определено ;
20    }
21    возврат appReducer ( состояние , действие ) ;
22  } ;
23
24  экспорт по умолчанию rootReducer ;

```

index.js, размещенный на ❤️ на GitHub

[просмотреть raw](#)

Теперь все редукторы будут инициализированы всякий раз, когда каждый пользователь выходит из системы.

Добавление элементов в панель навигации

Давайте добавим ссылку на панель навигации, чтобы мы могли выйти. Кроме того, измените отображение в зависимости от того, вошли вы в систему или нет. Откройте Headers.js файл и обновите его следующим образом:

```

1  // frontend / src / components / layout / Headers.js
2
3  импортировать React , { Component } из 'response' ;
4  импорт { Ссылка } от «реагировать-маршрутизатор-дом» ; // добавлено
5  импорта { подключения } от 'реагируют-Redux' ; // добавлено
6  import { logout } из '../actions/auth' ; // добавлено
7
8  Заголовок класса расширяет Компонент {
9    render ( ) {
10      const { пользователь , isAuthenticated } = это . реквизит . аутентификация ; //
11
12      // добавлено
13      const userLinks = (
14        < div className = 'правильное меню' >
15          < div className = 'UI простой выпадающий элемент' >
16            { пользователь ? пользователь . имя пользователя : '' }
17          < i className = 'раскрывающийся значок' / >
18          < div className = 'menu' >

```

```

21         < / a >
22     < / div >
23 < / div >
24 < / div >
25 ) ;
26
27 // добавлено
28 const guestLinks = (
29     < div className = 'правильное меню' >
30         < Link to = '/ register' className = 'item' >
31             Зарегистрироваться
32         < / Link >
33         < Link to = '/ login' className = 'item' >
34             Авторизоваться
35         < / Link >
36     < / div >
37 ) ;
38
39 // обновлено
40 возврат (
41     < div className = 'UI перевернутое меню' style = { { borderRadius : '0' } } >
42         < Link to = '/' className = 'header header' >
43             TodoCRUD
44         < / Link >
45         < Link to = '/' className = 'item' >
46             Дом
47         < / Link >
48         { Аутентифицирован ? userLinks : guestLinks }
49     < / div >
50 ) ;
51 }
52 }
53
54 // добавлено
55 const mapStateToProps = state => ( {
56     auth : гос . авт
57 } ) ;
58
59 // обновлено
60 экспорт по умолчанию подключить (
61     mapStateToProps ,
62     { выход }
63 ) ( Заголовок ) ;

```

Теперь давайте попробуем выйти из системы:

The screenshot shows the TodoCRUD application interface on the left and the Redux DevTools on the right. The application header includes 'TodoCRUD', 'Home', and a user profile 'John' with a 'Logout' button. The main form has a 'Task' input and an 'Add' button. Below the form, two tasks are listed: 'John's Task 1' and 'John's Task 2', each with a 'Delete' button. The Redux DevTools on the right shows the state of the application. The state tree includes 'form' (with 'todoForm'), 'todos' (an array of tasks), 'auth' (with 'isLoading' and 'isAuthenticated'), and 'user' (with 'id', 'username', 'email', and 'token'). The Redux DevTools also shows a list of actions, including '@@redux-form/SET_SUB...', 'LOGIN_SUCCESS', '@@redux-form/UNREGIS...', '@@redux-form/UNREGIS...', '@@redux-form/UNREGIS...', '@@redux-form/UPDATE...', '@@redux-form/DESTROY', '@@redux-form/REGISTE...', '@@redux-form/UPDATE...', and 'GET_TODOS'.

Выйти

The screenshot shows the TodoCRUD application interface on the left and the Redux DevTools on the right. The application header includes 'TodoCRUD', 'Home', 'Sign Up', and 'Login' buttons. The main form has 'Username' and 'Password' inputs and a 'Login' button. Below the form, there is a link 'Don't have an account? Register'. The Redux DevTools on the right shows the state of the application. The state tree includes 'form' (with 'todoForm' and 'loginForm'), 'todos' (an array of tasks), 'auth' (with 'isLoading' and 'isAuthenticated'), and 'user' (with 'id', 'username', 'email', and 'token'). The Redux DevTools also shows a list of actions, including '@@redux-form/UPDATE...', '@@redux-form/DESTROY', '@@redux-form/REGISTE...', '@@redux-form/UPDATE...', 'GET_TODOS', and 'LOGOUT_SUCCE...'.

После выхода

Реализация Реестра

Создатели действий

Откройте `actions/auth.js` файл и добавьте создателя нового действия для регистрации:

```

2
3 import {
4   USER_LOADING ,
5   USER_LOADED ,
6   AUTH_ERROR ,
7   REGISTER_SUCCESS , // добавлено
8   REGISTER_FAIL , // добавлено
9   LOGIN_SUCCESS ,
10  LOGIN_FAIL ,
11  LOGOUT_SUCCESS
12 } из './types' ;
13
14 // РЕГИСТРАЦИЯ ПОЛЬЗОВАТЕЛЯ
15 export const register = ( { имя пользователя , электронная почта , пароль } ) =>
16   // Заголовки
17   const config = {
18     заголовки : {
19       'Content-Type' : 'application / json'
20     }
21   } ;
22
23   // Тело запроса
24   const body = JSON . stringify ( { имя пользователя , адрес электронной почты , пар
25
26   попытаться {
27     const res = ждать axios . post ( '/ api / auth / register' , body , config )
28     отправка ( {
29       тип : REGISTER_SUCCESS ,
30       полезная нагрузка : рез . данные
31     } ) ;
32   } catch ( err ) {
33     отправка ( {
34       тип : REGISTER_FAIL
35     } ) ;
36     рассылка ( stopSubmit ( 'registerForm' , ошибка . ответ . данные ) ) ;
37   }
38 } ;

```

Мы используем `stopSubmit()` здесь снова, чтобы предотвратить двойную регистрацию пользователей.

Переходники

Откройте `reducers/auth.js` файл и добавьте действия в редуктор:

```

1  // frontend / src / reducers / auth.js
2
3  import {
4    USER_LOADING ,
5    USER_LOADED ,
6    AUTH_ERROR ,
7    REGISTER_SUCCESS , // добавлено
8    REGISTER_FAIL , // добавлено
9    LOGIN_SUCCESS ,
10   LOGIN_FAIL ,
11   LOGOUT_SUCCESS
12 } из './actions/types' ;
13
14 функция экспорта по умолчанию ( state = initialState , action ) {
15   switch ( action . type ) {
16     // ...
17     case REGISTER_SUCCESS : // добавлено
18     case LOGIN_SUCCESS :
19       localStorage . setItem ( 'token' , action . payload . token ) ;
20       возврат {
21         ... состояние ,
22         isLoading : false ,
23         isAuthenticated : правда ,
24         ... действие . полезная нагрузка
25       } ;
26     case AUTH_ERROR :
27     case REGISTER_FAIL : // добавлено
28     case LOGIN_FAIL :
29     case LOGOUT_SUCCESS :
30       localStorage . removeItem ( 'токен' ) ;
31       возврат {
32         ... состояние ,
33         isLoading : false ,
34         isAuthenticated : ложь ,
35         пользователь : null ,
36         токен : ноль
37       } ;
38     // ...
39   }
40 }

```

auth.js, размещенного на ❤️ в GitHub

[просмотр raw](#)

Это было очень легко, потому что мы только устанавливаем типы действий.

Создание формы регистрации

Давайте создадим регистрационную форму, используя **Redux Form** . Добавьте новый файл с именем RegisterForm.js в components/auth каталог:

```
1 // frontend / src / components / auth / RegisterForm.js
2
3 импортировать React , { Component } из 'response' ;
4 импорт { Ссылка , Перенаправление } от «реагировать-маршрутизатор-дом» ;
5 импорта { подключения } от 'реагируют-Redux' ;
6 импорт { Field , reduxForm } из 'redux-формы' ;
7 импорт { register } из '../actions/auth' ;
8
9 Класс RegisterForm расширяет Компонент {
10   renderField = ( { input , label , type , meta : { touch , error } } ) => {
11     возврат (
12       < DIV имя класса = { `поле $ { прикоснулся && ошибку ? 'error' : '' } ` } >
13         < label > { label } < / label >
14         < input { ... input } type = { type } / >
15         { тронут && ошибка && (
16           < span className = 'интерфейс, указывающий на красную базовую метку' > { er
17         ) }
18       < / div >
19     ) ;
20   } ;
21
22   onSubmit = formValues => {
23     это . реквизит . зарегистрироваться ( formValues ) ;
24   } ;
25
26   render ( ) {
27     если ( это . реквизит . isAuthenticated ) {
28       return < Redirect to = '/' / > ;
29     }
30     возврат (
31       < div className = 'ui container' >
32         < div className = 'UI сегмент' >
33           < форма
34             onSubmit = { это . реквизит . handleSubmit ( this . onSubmit ) }
35             className = 'UI Form '
36           >
37             < Поле
38               name = 'username'
39               type = 'text'
40               компонент = { это . renderField }
41               label = 'Имя пользователя'
42               validate = { [ обязательно , minLength3 , maxLength15 ] }
```

```

45     name = 'email'
46     type = 'email'
47     компонент = { это . renderField }
48     label = 'Email'
49     validate = { обязательно }
50 / >
51 < Поле
52     имя = «пароль»
53     type = 'password'
54     компонент = { это . renderField }
55     label = 'Password'
56     validate = { обязательно }
57 / >
58 < Поле
59     name = 'password2'
60     type = 'password'
61     компонент = { это . renderField }
62     label = 'Confirm Password'
63     validate = { [ обязательно , passwordsMatch ] }
64 / >
65 < button className = 'ui primary button' > Зарегистрироваться < / button
66 < / form >
67 < p style = { { marginTop : '1rem' } } >
68     Уже есть аккаунт? < Link to = '/ login' > Войти < / Link >
69 < / p >
70 < / div >
71 < / div >
72 ) ;
73 }
74 }
75
76 const required = value => ( значение ? undefined: 'обязательно' ) ;
77
78 const minLength = min => значение =>
79     значение && значение . длина < мин
80     ? `Должно быть не менее $ { min } символов`
81     : не определено ;
82
83 const minLength3 = minLength ( 3 ) ;
84
85 const maxLength = max => значение =>
86     значение && значение . длина > макс ? `Должно быть $ { max } символов или меньше`
87
88 const maxLength15 = maxLength ( 15 ) ;
89

```



```

92
93   const mapStateToProps = state => ( {
94     isAuthenticated : состояние . авт . isAuthenticated
95   } ) ;
96
97   RegisterForm = connect (
98     mapStateToProps ,
99     { зарегистрироваться }
100  )(RegisterForm);
101
102   export default reduxForm({
103     form: 'registerForm'
104   })(RegisterForm);

```

Форма регистрации почти такая же, как форма входа в систему, но в этой форме мы используем **проверку на уровне поля** .

Узнайте больше о проверке на уровне поля .

Сопоставление пароля можно определить, как указано выше, используя allValues параметр.

Затем мы включим RegisterForm в App компонент:

```

1  // frontend/src/components/App.js
2
3  import RegisterForm from './auth/RegisterForm'; // added
4
5  class App extends Component {
6    // ...
7
8    render() {
9      return (
10        <Provider store={store}>
11          <Router history={history}>
12            <Header />
13            <Switch>
14              <PrivateRoute exact path="/" component={Dashboard} />
15              <Route exact path="/delete/:id" component={TodoDelete} />
16              <Route exact path="/edit/:id" component={TodoEdit} />
17              <Route точный путь = '/ register' component = { RegisterForm } / > //
18              <Route точный путь = '/ login' компонент = { LoginForm } / >
19            < / Switch >

```

```
22      ) ;
23    }
24  }
```

App is deployed with us GitHub

[ПРОСМОТРЕТЬ RAW](#)

Теперь давайте проверим регистрационную форму:

The screenshot shows the registration form on the left and the Redux DevTools on the right. The form has fields for Username, Email, Password, and Confirm Password. The Username field contains 'John' and the Email field contains 'john@example.com'. Both fields have red error messages: 'A user with that username already exists.' and 'user with this email address already exists.' respectively. The Password and Confirm Password fields are empty. The Register button is blue. Below the form, there is a link 'Already have an account? [Login](#)'. The Redux DevTools on the right shows the state of the application. The state is an object with properties: 'registeredFields', 'fields', 'values', 'anyTouched', 'submitErrors', 'username', 'email', 'todos', and 'auth'. The 'submitErrors' property is an object with 'username' and 'email' properties, both containing error messages. The 'auth' property is an object with 'isLoading' and 'isAuthenticated' properties.

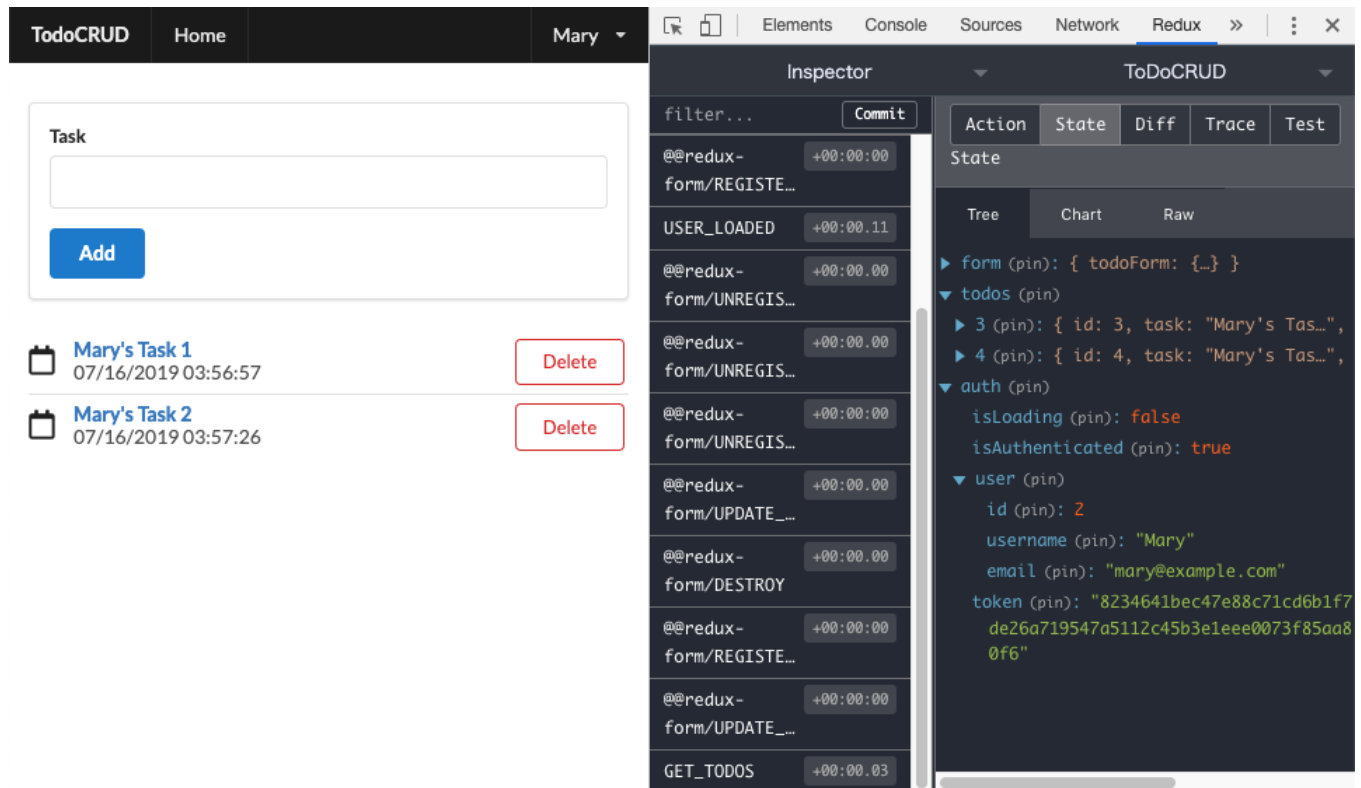
Ошибки на стороне сервера

The screenshot shows the registration form on the left and the Redux DevTools on the right. The form has fields for Username, Email, Password, and Confirm Password. The Username field contains 'Mary' and the Email field contains 'mary@example.com'. The Password field contains '....' and the Confirm Password field contains '.....'. There is a red error message 'Passwords do not match' below the Confirm Password field. The Register button is blue. Below the form, there is a link 'Already have an account? [Login](#)'. The Redux DevTools on the right shows the state of the application. The state is an object with properties: 'registeredFields', 'fields', 'values', 'anyTouched', 'syncErrors', 'password2', 'todos', and 'auth'. The 'syncErrors' property is an object with 'password2' property, containing the error message 'Passwords do not match'. The 'auth' property is an object with 'isLoading' and 'isAuthenticated' properties.

Пароли не соответствуют

Проверки работают правильно.

Давайте зарегистрируем нового пользователя и добавим пару задач:



The screenshot shows the TodoCRUD application interface on the left and the Redux DevTools on the right. The application has a header with 'TodoCRUD', 'Home', and a user profile 'Mary'. Below the header is a form to add tasks with a text input and an 'Add' button. Below the form is a list of tasks: 'Mary's Task 1' (07/16/2019 03:56:57) and 'Mary's Task 2' (07/16/2019 03:57:26), each with a 'Delete' button. The Redux DevTools show the state of the application, including the user object and the list of tasks.

```
@@redux-form/REGISTE... +00:00.00
USER_LOADED +00:00.11
@@redux-form/UNREGIS... +00:00.00
@@redux-form/UNREGIS... +00:00.00
@@redux-form/UNREGIS... +00:00.00
@@redux-form/UPDATE_... +00:00.00
@@redux-form/DESTROY +00:00.00
@@redux-form/REGISTE... +00:00.00
@@redux-form/UPDATE_... +00:00.00
GET_TODOS +00:00.03
```

State

```
form (pin): { todoForm: {...} }
todos (pin)
  3 (pin): { id: 3, task: "Mary's Tas...",
  4 (pin): { id: 4, task: "Mary's Tas...",
auth (pin)
  isLoading (pin): false
  isAuthenticated (pin): true
user (pin)
  id (pin): 2
  username (pin): "Mary"
  email (pin): "mary@example.com"
token (pin): "8234641bec47e88c71cd6b1f7de26a719547a5112c45b3e1eee0073f85aa80f6"
```

Новый пользователь

Отлично!

Обновление URLconf

Наконец, мы добавим login и register URL в frontend/urls.py файл:

```
1 # frontend / urls.py
2
3 urlpatterns = [
4     путь ( '', индекс ),
5     путь ( 'логин' , индекс ), # добавлено
6     путь ( 'регистрация' , индекс ), # добавлено
7     путь ( 'edit / <int: pk>' , TodoDetailView . as_view ()),
8     путь ( 'delete / <int: pk>' , TodoDetailView . as_view ()),
9 ]
```

urls.py с with на GitHub

[посмотреть raw](#)

Теперь, даже если браузер обновляется на странице входа или регистрации, он будет отображаться правильно.

Этот урок заканчивается здесь. Исходный код также доступен на GitHub .
Спасибо за чтение!

[программирование](#)

[кодирование](#)

[Джанго](#)

[реагировать](#)

[Redux](#)

[ОсправкеЮридическая](#)