

2.2

Руководство разработчика модулей

Система модулей позволяет создавать и устанавливать дополнительные функциональные модули, расширяющие возможности основной системы. Модуль представляет собой полноценное Django приложение, упакованное в ZIP архив.

Система модулей позволяет создавать и устанавливать дополнительные функциональные модули, расширяющие возможности основной системы. Модуль представляет собой полноценное Django приложение, упакованное в ZIP архив.

Структура модуля

Модуль должен быть упакован в ZIP архив и содержать следующие компоненты:

- module_name/ |— module.json # Конфигурационный файл (обязателен)
- |— models.py # Модели Django
- |— views.py # Представления
- urls.py # URL маршруты
- |— admin.py # Админка
- |— apps.py # Конфигурация приложения (обязателен)
- |— migrations/ # Миграции базы данных
 - |— 0001_initial.py # Начальная миграция
- templates/ # Шаблоны
- module_name/ # Папка с именем модуля
- |— static/ # Статические файлы
- |— ...

Другие файлы приложения

Важные требования к файлам модуля

1. apps.py

Файл apps.py должен содержать класс AppConfig с явно указанным label:

```
from django.apps import AppConfig from django.utils.translation import gettext_lazy as _
class YourModuleConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'your_module_name' # Имя модуля (будет автоматически обновлено при установке)
    label = 'your_module_name' # ОБЯЗАТЕЛЬНО: явно указываем label для правильного app_label
    verbose_name = _('Your Module')
    def ready(self): """Инициализация модуля при загрузке""" pass
```

Важно: Поле label должно совпадать с app_name из module.json -> settings.app_name. Это необходимо для правильного определения app_label

моделей.

2. models.py

В Meta классах всех моделей обязательно указывайте `app_label`: `from django.db import models from django.utils.translation import gettext_lazy as _`
`class YourModel(models.Model): name = models.CharField(_('Name'), max_length=255)`
`class Meta: app_label = 'your_module_name' # △ ОБЯЗАТЕЛЬНО: явно указываем app_label verbose_name = _('Your Model') verbose_name_plural = _('Your Models')`
Важно: `app_label` должен совпадать с `app_name` из `module.json` -> `settings.app_name`. Без этого Django может неправильно определить имя таблицы в базе данных.

3. migrations/0001_initial.py

В классе миграции обязательно указывайте `app_label`: `from django.db import migrations, models`
`class Migration(migrations.Migration): initial = True`
`# △ ОБЯЗАТЕЛЬНО: явно указываем app_label для миграции app_label = 'your_module_name'`
`dependencies = [# ...]`
`operations = [# ...]`
Важно: `app_label` должен совпадать с `app_name` из `module.json` -> `settings.app_name`. Это необходимо для правильного применения миграций.

4. templates/

Шаблоны должны находиться в папке с именем модуля: `templates/ your_module_name/` # Папка с именем модуля |— `list.html` |— `detail.html` |— `form.html`
В `views.py` используйте полный путь к шаблону: `class YourListView(ListView): template_name = 'your_module_name/list.html' # Полный путь с именем модуля`
Важно: Система автоматически добавляет пути к шаблонам модулей в `TEMPLATES['DIRS']` при старте приложения, поэтому шаблоны будут найдены автоматически.

Создание шаблонов для модулей

Все шаблоны модулей должны наследоваться от базового шаблона `partials/base.html`. Это обеспечивает единообразный внешний вид и правильную загрузку всех необходимых библиотек.

Базовый шаблон

Обязательно используйте:

Структура блоков базового шаблона

Базовый шаблон `partials/base.html` предоставляет следующие блоки для переопределения:

\ - Заголовок страницы (отображается в) \ - Дополнительные CSS файлы (загружаются перед основными) \ - Основные CSS (Bootstrap, иконки, `app.min.css`) - НЕ ПЕРЕОПРЕДЕЛЯЙТЕ, используйте `extra_css` \ - Шапка страницы (опционально) \ - Боковая панель (опционально) \ - Основной контент страницы (обязательно) \ - Дополнительный контент после основного \ - Основные JavaScript библиотеки (jQuery, Bootstrap, toastr, HTMX) - НЕ ПЕРЕОПРЕДЕЛЯЙТЕ! \ - Дополнительные JavaScript файлы и скрипты (внутри блока `javascript`)

⚠ Важно: Правильное использование блоков JavaScript

НЕПРАВИЛЬНО - переопределение блока `javascript`:

```
$(document).ready(function() { // код });
```

ПРАВИЛЬНО - использование блока `extra_javascript`:

```
$(document).ready(function() { // код });
```

Почему это важно:

Блок `javascript` в базовом шаблоне загружает jQuery, Bootstrap, toastr и другие необходимые библиотеки. При переопределении блока `javascript` эти библиотеки не загружаются, что приводит к ошибкам типа "jQuery is not defined" или "toastr is not defined". Блок `extra_javascript` находится внутри блока `javascript`, поэтому все базовые библиотеки уже загружены.

Пример минимального шаблона

Добавление дополнительных CSS

Используйте блок `extra_css` для подключения дополнительных стилей:

Важно: Если вам нужно включить базовые стили (Bootstrap и т.д.), используйте:

Добавление JavaScript

Используйте блок `extra_javascript` для подключения дополнительных скриптов:

```
$(document).ready(function() { // jQuery доступен, так как он загружен в базовом шаблоне $('#my-select').select2({ theme: 'bootstrap-5', width: '100%' }); $('#my-datepicker').datepicker({ format: 'dd.mm.yyyy', autoclose: true }); });
```

Bootstrap разметка

Система использует Bootstrap 5. Используйте стандартные классы Bootstrap:

Контейнеры:

Сетка:

Карточки: Заголовок Текст

Кнопки:

Таблицы:

Формы:

Badges (значки статусов):

Пагинация:

Иконки

Система использует Boxicons (класс bx) и Bootstrap Icons (класс bi). Примеры:

Переводы (i18n)

В шаблонах:

Используйте в начале шаблона Используйте для всех строк, отображаемых пользователю Используйте для сложных строк с переменными

Примеры:

Item "" was created successfully.

В Python коде:

Используйте `from django.utils.translation import gettext_lazy as _` Используйте `_("Text")` для всех строк, отображаемых пользователю

`from django.utils.translation import gettext_lazy as _`

```
class MyModel(models.Model):
    name = models.CharField(_("Name"),
                             max_length=255)
    status = models.CharField(_("Status"), max_length=50)
```

Статические файлы

В шаблонах:

Структура статических файлов модуля: `static/` | `your_module_name/` # Папка с именем модуля | `css/` | `custom.css` | `js/` | `custom.js` | `images/` | `logo.png`

Использование:

Использование HTMX

Система включает HTMX для динамических обновлений без JavaScript. Используйте HTMX вместо чистого JavaScript там, где это возможно:

Полный пример шаблона списка

```
$(document).ready(function() { $('#status').select2({ theme: 'bootstrap-5', width: '100%' }); });
```

Рекомендации

Всегда используйте \ - это обеспечивает единообразный внешний вид
Используйте extra_javascript вместо javascript - это гарантирует загрузку всех базовых библиотек
Используйте переводы - все строки должны быть переводимыми через
Используйте Bootstrap классы - не создавайте кастомные стили без необходимости
Используйте HTMX - предпочитайте HTMX чистому JavaScript для динамических обновлений
Следуйте структуре - используйте стандартную структуру с page-title-box, card, card-body
Используйте иконки - используйте Boxicons (bx) или Bootstrap Icons (bi) для визуального оформления

Файл конфигурации module.json

Файл module.json является обязательным для модулей, устанавливаемых из ZIP архива. Для локальных модулей, созданных вручную, этот файл не требуется, но может быть добавлен позже при создании версии модуля.

Пример конфигурации

```
{ "name": "Example Module", "code": "example_module", "version": "1.0.0", "description": "Описание модуля", "author": "Имя автора", "min_system_version": "5.2.0", "dependencies": [], "category": "Business", "tags": ["example", "demo"], "menu": { "title": "Example Module", "icon": "bx bx-package", "permission": "example_module.view", "items": [ { "title": "Dashboard", "url": "/example/dashboard/", "icon": "bx bx-dashboard", "permission": "example_module.view_dashboard" }, { "title": "Settings", "url": "/example/settings/", "icon": "bx bx-cog", "permission": "example_module.view_settings" }, { "title": "Submenu", "icon": "bx bx-menu", "items": [ { "title": "Sub Item 1", "url": "/example/sub/item1/", "permission": "example_module.view_sub_item1" }, { "title": "Sub Item 2", "url": "/example/sub/item2/", "permission": "example_module.view_sub_item2" } ] } ] }, "urls": [ { "path": "example/", "include": "example_module.urls", "namespace": "example" } ], "settings": { "app_name": "example_module", "migrations": true }, "notification_templates": { "email": [ { "name": "ExampleNotification", "title": "Example Notification", "subject": "Example: {title}", "body": "Hello {user_name}, this is an example notification about {title}.", "note": "Example email notification template", "section": "Example Module" } ], "messenger": [ { "name": "ExampleNotification", "title": "Example Notification", "body": "Hello {user_name}, this is an example notification about {title}.", "note": "Example messenger notification template", "section": "Example Module" } ] } }
```

Важно: Меню поддерживает до 3 уровней вложенности подменю.

Описание полей

Основные поля

name (обязательно): Название модуля **code** (обязательно): Уникальный код модуля (используется для идентификации) **version** (обязательно): Версия модуля (формат: X.Y.Z) **description**: Описание модуля **author**: Автор модуля **min_system_version**: Минимальная версия системы (Django) **dependencies**: Список зависимостей от других модулей **category**: Категория модуля **tags**: Список тегов для поиска

Меню (menu)

Конфигурация пункта меню в боковой панели:

title: Название пункта меню **icon**: CSS класс иконки (например, "bx bx-package") **permission**: Право доступа для отображения пункта меню **items**: Список подпунктов меню **title**: Название подпункта **url**: URL подпункта **icon**: CSS класс иконки **permission**: Право доступа **items**: Вложенные подпункты (поддержка до 3 уровней вложенности)

URLs (urls)

Список URL маршрутов для регистрации:

path: Путь URL (например, "example/") **include**: Путь к модулю urls (например, "example_module.urls") **namespace**: Пространство имен (опционально)

Настройки (settings)

app_name (обязательно): Имя Django приложения **migrations**: Выполнять ли миграции при установке (по умолчанию: true)

Шаблоны уведомлений (notification_templates)

Модули могут определять свои шаблоны уведомлений для email и мессенджеров. Структура: { "notification_templates": { "email": [{ "name": "TemplateName", "title": "Template Title", "subject": "Email subject with {variables}", "body": "Email body with {variables}", "style": "CSS styles (optional)", "note": "Template description (optional)", "section": "Section name (optional)" }], "messenger": [{ "name": "TemplateName", "title": "Template Title", "body": "Message body with {variables}", "note": "Template description (optional)", "section": "Section name (optional)" }] } }

Поля email шаблона:

name (обязательно): Системное имя шаблона (будет префиксовано кодом модуля) **title**: Отображаемое название шаблона **subject** (обязательно): Тема

письма с переменными в формате {variable} body (обязательно): Тело письма с переменными style: CSS стили для email (опционально) note: Описание шаблона (опционально) section: Раздел для группировки шаблонов (опционально)

Поля messenger шаблона:

name (обязательно): Системное имя шаблона (будет префиксовано кодом модуля) title: Отображаемое название шаблона body (обязательно): Тело сообщения с переменными в формате {variable} note: Описание шаблона (опционально) section: Раздел для группировки шаблонов (опционально)

Использование шаблонов:

Имя шаблона в коде: {module_code}_{template_name} Например, для модуля example_module и шаблона ExampleNotification полное имя будет: example_module_ExampleNotification Используйте полное имя при вызове send_notify():

```
python from notifications.services import send_notify send_notify(obj, recipients, template_name='example_module_ExampleNotification')
```

Переменные в шаблонах: Используйте переменные в формате {variable_name} Доступны все поля объекта, переданного в send_notify() Для связанных объектов используйте префикс: {related_field_name}_field_name Примеры: {title}, {user_name}, {created_at}, {author_display_name}

Работа с данными основной системы

Модули могут взаимодействовать с данными основной системы двумя способами:

Через API - для чтения данных с защитой от изменений в моделях Напрямую через модели - для сложных операций, записи данных и использования методов моделей

Рекомендация: Используйте API для чтения данных, если хотите защитить модуль от изменений в моделях. Используйте прямые импорты для сложных запросов, методов моделей и операций записи.

Когда использовать API

Используйте API в следующих случаях:

Чтение данных - для получения информации о сущностях системы (сотрудники, задачи, проекты и т.д.) Защита от изменений - когда хотите защитить модуль от изменений во внутренней структуре моделей Стандартизированный интерфейс - когда нужен единый способ работы с данными

Преимущества API:

Абстракция от внутренней структуры моделей Защита от изменений в моделях (API может оставаться совместимым) Стандартизированный интерфейс Безопасность через систему прав доступа Django

Ограничения API:

Только чтение данных (read-only) Ограниченный набор полей (только те, что в сериализаторе) Нет доступа к методам моделей Нет возможности создавать/изменять/удалять объекты

Когда напрямую обращаться к моделям

Используйте прямые импорты моделей в следующих случаях:

Создание/изменение/удаление объектов - API предоставляет только чтение

Сложные запросы - когда нужны сложные фильтры, аннотации, агрегации

Использование методов моделей - когда нужно вызвать методы объектов (например, `get_absolute_url()`, `save()` с дополнительной логикой)

Связанные объекты - когда нужен доступ к связанным объектам через `ForeignKey/ManyToMany`

Транзакции - когда нужны атомарные операции с несколькими объектами

Производительность - когда нужны оптимизированные запросы с `select_related()`, `prefetch_related()`

Преимущества прямого доступа:

Полный доступ ко всем полям и методам моделей

Возможность создания, изменения и удаления объектов

Сложные запросы с использованием Django ORM

Использование методов моделей и менеджеров

Транзакции и атомарные операции

Важно:

При прямом доступе к моделям вы работаете с внутренней структурой системы

Изменения в моделях основной системы могут повлиять на ваш модуль

Всегда проверяйте права доступа перед операциями записи

Примеры использования

Пример 1: Получение списка сотрудников через API

```
import requests from django.conf import settings
def get_employees_via_api(request): """Получение сотрудников через API"""
response = requests.get( f'{settings.BASE_URL}/modules/api/core/employees/',
cookies=request.COOKIES # Используем сессию пользователя )
if response.status_code == 200: employees = response.json()
return employees
```

Пример 2: Получение списка сотрудников напрямую через модели

```
from personal.models import Employee
def get_employees_direct(): """Получение сотрудников напрямую через модели"""
# Простой запрос employees = Employee.objects.filter(status='working')
# Сложный запрос с оптимизацией employees = Employee.objects.select_related('manager', 'org').filter(status='working', org__name__icontains='Отдел').order_by('last_name', 'first_name')
return employees
```

Пример 3: Создание задачи (только через модели)

```
from tasks.models import Task from personal.models import Employee from django.utils import timezone from datetime import timedelta
```

```
def create_task_for_employee(employee_id, title, description): """Создание задачи - только через модели, так как API read-only""" employee = Employee.objects.get(id=employee_id) task = Task.objects.create( title=title, description=description, executor=employee, author=employee, # или текущий пользователь due_date=timezone.now() + timedelta(days=7), completed=False ) return task
```

Пример 4: Сложный запрос с агрегацией (только через модели)

```
from tasks.models import Task from django.db.models import Count, Q, Avg from django.utils import timezone from datetime import timedelta
def get_task_statistics(employee_id): """Получение статистики по задачам - сложный запрос через модели""" now = timezone.now() week_ago = now - timedelta(days=7) stats = Task.objects.filter(executor_id=employee_id).aggregate( total=Count('id'), completed=Count('id', filter=Q(completed=True)), overdue=Count('id', filter=Q(completed=False, due_date__lt=now)), due_soon=Count('id', filter=Q(completed=False, due_date__gte=now, due_date__lte=now + timedelta(days=3) )), avg_progress=Avg('progress') ) return stats
```

Пример 5: Использование методов модели (только через модели)

```
from tasks.models import Task from notifications.services import send_notify
def notify_about_task(task_id): """Отправка уведомления о задаче - используем методы модели""" task = Task.objects.get(id=task_id) # Используем метод get_absolute_url() модели task_url = task.get_absolute_url() # Используем связанные объекты executor = task.executor if executor and executor.user: send_notify( obj=task, recipients=[executor.user], template_name='module_code_TaskNotification', extra_context={ 'task_url': task_url, 'executor_name': executor.display_name } )
```

Пример 6: Транзакция с несколькими объектами (только через модели)

```
from tasks.models import Task from django.db import transaction from personal.models import Employee
@transaction.atomic def create_tasks_for_team(team_member_ids, title_template): """Создание задач для команды - атомарная операция""" tasks = [] for member_id in team_member_ids: employee = Employee.objects.get(id=member_id) task = Task.objects.create( title=f"{title_template} - {employee.display_name}", executor=employee, # ... другие поля ) tasks.append(task) # Если произойдет ошибка, все изменения откатятся return tasks
```

Пример 7: Работа с ForeignKey и ManyToMany (только через модели)

```
from tasks.models import Task from projects.models import Project
def get_project_tasks(project_id): """Получение задач проекта с оптимизацией"""
# Используем select_related для ForeignKey tasks = Task.objects.select_related(
'executor', 'author', 'project' ).filter(project_id=project_id)
# Для ManyToMany используем prefetch_related # (если бы у Task было поле
ManyToMany)
return tasks
```

Рекомендации по выбору подхода

Используйте API, если:

- Нужно только чтение данных
- Хотите защитить модуль от изменений в моделях
- Нужна стандартизированная интеграция
- Достаточно полей, предоставляемых API

Используйте прямые импорты моделей, если:

- Нужно создавать/изменять/удалять объекты
- Нужны сложные запросы с агрегациями
- Нужно использовать методы моделей
- Нужны оптимизированные запросы (select_related, prefetch_related)
- Нужны транзакции
- Нужен доступ ко всем полям модели

Импорт моделей основной системы

```
При прямом доступе к моделям используйте полные пути импорта: #
Сотрудники from personal.models import Employee, Organigram
# Пользователи from users.models import CustomUser, CustomGroup
# Задачи from tasks.models import Task, TaskSource
# Проекты from projects.models import Project, ProjectRole, ProjectType
# Заявки from servicedesk.requests import Request
# Договоры from finance.contracts import Contract
# Уведомления from system.notifications import Notification from
notifications.services import send_notify
# Клиенты from clients.models import Clients
```

Важно:

Всегда проверяйте наличие прав доступа перед операциями записи
Используйте get_object_or_404() для безопасного получения объектов
Используйте транзакции для атомарных операций Оптимизируйте запросы с помощью select_related() и prefetch_related()

API для модулей

API для управления модулями

Базовый URL: /modules/api/

Список модулей

GET /modules/api/modules/

Параметры фильтрации:

category: Фильтр по категории installed: Фильтр по статусу установки (true/false)

Установка модуля

POST /modules/api/modules/{id}/install/

Тело запроса (опционально): { "version_id": 1 }

Обновление модуля

POST /modules/api/modules/{id}/update-module/

Тело запроса (опционально): { "version_id": 2 }

Отключение модуля

POST /modules/api/modules/{id}/disable/

Включение модуля

POST /modules/api/modules/{id}/enable/

Удаление модуля

POST /modules/api/modules/{id}/uninstall/

Тело запроса: { "confirm": true, "rollback_migrations": false }

API для основных сущностей системы

API предоставляет доступ к основным сущностям системы для авторизованных пользователей внутри системы. Модули могут использовать API для работы с данными, что обеспечивает защиту от изменений в моделях.

Базовый URL

/modules/api/core/

Доступные эндпоинты

Сотрудники (Employees)

GET /modules/api/core/employees/ GET /modules/api/core/employees/{id}/

Параметры фильтрации:

status: Фильтр по статусу org: Фильтр по подразделению

Пользователи (Users)

GET /modules/api/core/users/ GET /modules/api/core/users/{id}/

Параметры фильтрации:

user_type: Тип пользователя is_active: Активность пользователя

Договоры (Contracts)

GET /modules/api/core/contracts/ GET /modules/api/core/contracts/{id}/

Параметры фильтрации:

status: Статус договора client: ID клиента

Заявки (Requests)

GET /modules/api/core/requests/ GET /modules/api/core/requests/{id}/

Параметры фильтрации:

status: Статус заявки executor: ID исполнителя client: ID клиента

Проекты (Projects)

GET /modules/api/core/projects/ GET /modules/api/core/projects/{id}/

Параметры фильтрации:

status: Статус проекта manager: ID менеджера

Задачи (Tasks)

GET /modules/api/core/tasks/ GET /modules/api/core/tasks/{id}/

Параметры фильтрации:

executor: ID исполнителя project: ID проекта completed: Статус выполнения (true/false)

Уведомления (Notifications)

GET /modules/api/core/notifications/ GET /modules/api/core/notifications/{id}/ POST
/modules/api/core/notifications/{id}/mark_read/ POST
/modules/api/core/notifications/mark_all_read/

Аутентификация и доступ к API

API модулей доступно для авторизованных пользователей внутри системы. API модулей предназначено для использования модулями и другими компонентами системы. Оно служит слоем абстракции, который защищает модули от изменений во внутренней структуре моделей.

Доступ к API

API доступно для:

- Авторизованных пользователей через сессионную аутентификацию Django
- Модулей - модули могут использовать API через HTTP запросы с сессией пользователя
- Внутренних системных запросов - с заголовком X-Internal-Request или с localhost

API блокирует:

- Неавторизованные запросы
- Внешние запросы без сессии Django

Использование API из модулей

Модули могут использовать API для работы с данными основной системы. Это обеспечивает:

Абстракцию - модули не зависят от внутренней структуры моделей
Защиту от изменений - при изменении моделей API может оставаться совместимым
Стандартизированный интерфейс - единый способ работы с данными

Пример использования API из модуля: `import requests from django.conf import settings`

```
def get_employees_via_api(request): """Получение сотрудников через API"""  
response = requests.get( f'{settings.BASE_URL}/modules/api/core/employees/',  
cookies=request.COOKIES # Используем сессию текущего пользователя )  
if response.status_code == 200: return response.json() return []
```

Альтернатива - прямые импорты моделей: `from personal.models import Employee`

```
def get_employees_direct(): """Получение сотрудников напрямую через модели"""  
return Employee.objects.filter(status='working')
```

Рекомендация: Используйте API, если хотите защитить модуль от изменений в моделях. Используйте прямые импорты, если нужны сложные запросы, методы моделей или операции записи (API предоставляет только чтение).

Внешний API

Для внешних интеграций, мобильных приложений и внешних сервисов будет создан отдельный внешний API с соответствующей аутентификацией (токены, API ключи и т.д.).

Примеры использования API

Пример 1: Использование API из модуля `import requests from django.conf import settings`

```
def get_employees_via_api(request): """Получение сотрудников через API из модуля""" # Используем сессию текущего пользователя response = requests.get(f'{settings.BASE_URL}/modules/api/core/employees/', cookies=request.COOKIES ) if response.status_code == 200: return response.json() return []
```

Пример 2: Использование API с фильтрацией `import requests from django.conf import settings`

```
def get_active_employees(request): """Получение активных сотрудников через API""" response = requests.get(f'{settings.BASE_URL}/modules/api/core/employees/', params={'status': 'working'}, # Фильтрация cookies=request.COOKIES ) if response.status_code == 200: return response.json() return []
```

Пример 3: Альтернатива - прямые импорты моделей # ПРАВИЛЬНО - используйте прямые импорты для сложных операций `from personal.models import Employee from tasks.models import Task`

```
def get_employees_with_tasks(): """Получение сотрудников с задачами - требует сложный запрос""" return Employee.objects.filter(status='working').prefetch_related('task_set').annotate(task_count=Count('task_set'))
```

Рекомендация: Используйте API для простого чтения данных, прямые импорты - для сложных операций и записи.

Модели системы: описание полей

Для правильной работы с API и шаблонами уведомлений важно знать точные названия полей моделей системы. Ниже приведено подробное описание каждой модели, доступной через API.

Сотрудник (Employee)

Модель: `personal.models.Employee`

Доступные поля через API:

Поле	Тип	Описание	Пример
<code>id</code>	<code>Integer</code>	Уникальный идентификатор	1
<code>display_name</code>	<code>String</code>	Отображаемое имя	"Иванов Иван Иванович"
<code>first_name</code>	<code>String</code>	Имя	"Иван"
<code>last_name</code>	<code>String</code>	Фамилия	"Иванов"
<code>middle_name</code>	<code>String</code>	Отчество	"Иванович"

title String Должность "Менеджер проектов"
email String Email адрес "ivanov@example.com"
mobile_number String Мобильный телефон "+7 (999) 123-45-67"
telephone_number String Рабочий телефон "+7 (495) 123-45-67"
manager Integer (FK) ID руководителя 5
manager_name String Имя руководителя (read-only) "Петров Петр Петрович"
org Integer (FK) ID подразделения 10
org_name String Название подразделения (read-only) "Отдел разработки"
status String Статус сотрудника "working", "vacation", "dismissed"
hire_date Date Дата приема на работу "2020-01-15"
avatar String (URL) URL аватара "/media/avatars/user.jpg"
created_at DateTime Дата создания "2020-01-15T10:00:00Z"

Пример использования в шаблоне уведомления:

```
{ "subject": "Задача назначена: {title}", "body": "Здравствуйте,  
{executor_str}!\n\nВам назначена задача: {title}\n\nИсполнитель:  
{executor_str}\nРуководитель: {manager_name}" }
```

Примечание: В шаблонах уведомлений доступны как прямые поля объекта (например, {executor.display_name}), так и вычисляемые значения через extra_context (например, {executor_str}).

Пользователь (User)

Модель: users.models.CustomUser

Доступные поля через API:

Поле	Тип	Описание	Пример
id	Integer	Уникальный идентификатор	1
email	String	Email адрес (уникальный)	"user@example.com"
display_name	String	Отображаемое имя	"Иванов Иван"
first_name	String	Имя	"Иван"
last_name	String	Фамилия	"Иванов"
middle_name	String	Отчество	"Иванович"
title	String	Должность/роль	"Разработчик"
mobile_number	String	Мобильный телефон	" +7 (999) 123-45-67"
avatar	String (URL)	URL аватара	"/media/avatars/user.jpg"
is_staff	Boolean	Является ли сотрудником	true
is_active	Boolean	Активен ли пользователь	true
date_joined	DateTime	Дата регистрации	"2020-01-15T10:00:00Z"
employee_id	Integer	ID связанного сотрудника (read-only)	5

Пример использования в шаблоне уведомления:

```
{ "subject": "Добро пожаловать, {user_name}!", "body": "Здравствуйте,  
{user_name}!\n\nВаш email: {user.email}\n\nДата регистрации:  
{user.date_joined}" }
```

Договор (Contract)

Модель: `finance.contracts.Contract`

Доступные поля через API:

Поле	Тип	Описание	Пример
<code>id</code>	Integer	Уникальный идентификатор	1
<code>display_name</code>	String	Отображаемое название	"Договор №123 от 01.01.2024"
<code>number</code>	String	Номер договора	"123/2024"
<code>date</code>	Date	Дата договора	"2024-01-01"
<code>date_expiry</code>	Date	Дата окончания	"2024-12-31"
<code>client</code>	Integer (FK)	ID клиента	10
<code>client_name</code>	String	Название клиента (read-only)	"ООО Рога и Копыта"
<code>contractor</code>	Integer (FK)	ID подрядчика	20
<code>contractor_name</code>	String	Название подрядчика (read-only)	"ООО Подрядчик"
<code>recipient</code>	Integer (FK)	ID получателя	30
<code>recipient_name</code>	String	Название получателя (read-only)	"ООО Получатель"
<code>status</code>	String	Статус договора	"planning", "actual", "noactual"
<code>currency</code>	String	Валюта	"RUB", "USD", "EUR"
<code>currency_price</code>	Decimal	Сумма договора	"1000000.00"
<code>pay_method</code>	String	Способ оплаты	"cash", "cashless"
<code>pay_date</code>	Date	Дата оплаты	"2024-06-01"
<code>prolong</code>	Boolean	Автопродление	true
<code>note</code>	String	Примечание	"Дополнительная информация"
<code>created_at</code>	DateTime	Дата создания	"2024-01-01T10:00:00Z"
<code>modified_at</code>	DateTime	Дата изменения	"2024-01-15T12:00:00Z"

Пример использования в шаблоне уведомления:

```
{ "subject": "Договор {number} истекает", "body": "Договор {number} от {date} с клиентом {client_name} истекает {date_expiry}.\n\nСумма: {currency_price} {currency}" }
```

Заявка (Request)

Модель: `servicedesk.requests.Request`

Доступные поля через API:

Поле	Тип	Описание	Пример
<code>id</code>	Integer	Уникальный идентификатор	1
<code>title</code>	String	Название заявки	"Проблема с принтером"
<code>theme</code>	String	Тема заявки	"Техническая поддержка"
<code>message</code>	String	Текст заявки	"Принтер не печатает"

client Integer (FK) ID клиента 10
source Integer (FK) ID источника 1
type Integer (FK) ID типа заявки 5
type_name String Название типа (read-only) "Инцидент"
category Integer (FK) ID категории 3
priority Integer (FK) ID приоритета 2
priority_name String Название приоритета (read-only) "Высокий"
impact Integer (FK) ID влияния 1
status Integer (FK) ID статуса 1
status_name String Название статуса (read-only) "В работе"
executoruser Integer (FK) ID исполнителя (пользователь) 15
executor_name String Имя исполнителя (read-only) "Петров Петр"
executorgroup Integer (FK) ID группы исполнителей 20
creator Integer (FK) ID создателя 12
creator_name String Имя создателя (read-only) "Иванов Иван"
startdate DateTime Дата начала "2024-01-15T10:00:00Z"
duedate DateTime Срок выполнения "2024-01-20T18:00:00Z"
deadline DateTime Дедлайн "2024-01-20T18:00:00Z"
responsetime DateTime Время отклика "2024-01-15T11:00:00Z"
respondedtime DateTime Время ответа "2024-01-15T12:00:00Z"
resolvedtime DateTime Время решения "2024-01-18T15:00:00Z"
createdtime DateTime Дата создания "2024-01-15T10:00:00Z"
lastedittime DateTime Дата последнего изменения "2024-01-18T15:00:00Z"
is_draft Boolean Черновик false

Пример использования в шаблоне уведомления:

```
{ "subject": "Новая заявка: {title}", "body": "Создана новая заявка:\n\nНазвание:\n{title}\nТип: {type_name}\nПриоритет: {priority_name}\nСтатус:\n{status_name}\nИсполнитель: {executor_name}\nСрок: {duedate}" }
```

Проект (Project)

Модель: projects.models.Project

Доступные поля через API:

Поле	Тип	Описание	Пример
id	UUID	Уникальный идентификатор	(UUID) "550e8400-e29b-41d4-a716-446655440000"
name	String	Название проекта	"Разработка нового сайта"
code	String	Код проекта	"PROJ-001"
description	String	Описание проекта	"Полная разработка корпоративного сайта"
manager	Integer (FK)	ID менеджера проекта (Employee)	5
manager_name	String	Имя менеджера (read-only)	"Иванов Иван Иванович"
status	Integer	Статус проекта	1 (Initiation), 2 (Planing), 3 (Execution), 4 (Completion), 5 (Control), 6 (Completed), 7 (Canceled)
status_display	String	Название статуса (read-only)	"В работе"
contractor	Integer (FK)	ID подрядчика	20
owner	String	Функциональный заказчик	"ООО Заказчик"
budget	Decimal	Бюджет проекта	"5000000.00"

type Integer (FK) ID типа проекта 3
class_importance String Класс важности "A" (высокий), "B" (средний), "C" (низкий)
class_deadline String Класс дедлайна "A", "B", "C"
class_budget String Класс бюджета "A", "B", "C"
goals_justification String Цели и обоснование (HTML) "Цель проекта..."
expected_results String Ожидаемые результаты (HTML) "Результаты..."
scope_description String Описание области (HTML) "Область..."
created_at DateTime Дата создания "2024-01-01T10:00:00Z"
modified_at DateTime Дата изменения "2024-01-15T12:00:00Z"

Пример использования в шаблоне уведомления:

```
{ "subject": "Проект {name} - обновление статуса", "body": "Проект {name} (код: {code}) изменил статус на {status_display}.\n\nМенеджер: {manager_name}\nБюджет: {budget}" }
```

Задача (Task)

Модель: tasks.models.Task

Доступные поля через API:

Поле Тип Описание Пример

id Integer Уникальный идентификатор 1
title String Название задачи "Разработать API"
description String Описание задачи "Создать REST API для модуля"
author Integer (FK) ID автора задачи (Employee) 5
author_name String Имя автора (read-only) "Иванов Иван Иванович"
executor Integer (FK) ID исполнителя (Employee) 10
executor_name String Имя исполнителя (read-only) "Петров Петр Петрович"
control Integer (FK) ID контролера (Employee) 15
project Integer (FK) ID проекта 100
project_name String Название проекта (read-only) "Разработка нового сайта"
request Integer (FK) ID связанной заявки 50
duedate DateTime Срок выполнения "2024-01-20T18:00:00Z"
completeddate Date Дата выполнения "2024-01-18"
completed Boolean Выполнена ли задача false
canceled Boolean Отменена ли задача false
priority Boolean Приоритетная задача true
start_date DateTime Дата начала (для диаграммы Гантта) "2024-01-15T09:00:00Z"
duration_days Integer Длительность в днях 5
progress Integer Прогресс выполнения (0-100) 50
is_milestone Boolean Является ли контрольной точкой false
is_project_lifecycle Boolean Является ли задачей жизненного цикла проекта false
result_type String Тип результата контрольной точки "event", "document"
parent Integer (FK) ID родительской задачи 2
created_at DateTime Дата создания "2024-01-15T10:00:00Z"
modified_at DateTime Дата изменения "2024-01-18T15:00:00Z"

Пример использования в шаблоне уведомления:

```
{ "subject": "Просроченная задача: {title}", "body": "Здравствуйте, {executor_str}!\n\nУ вас есть просроченная задача:\n\nНазвание: {title}\nОписание: {description}\nСрок выполнения: {duedate}\nПроект: {project_str}\n\nПожалуйста, выполните задачу или обновите срок выполнения.\n\nСсылка: {url}" }
```

Важно: В шаблонах уведомлений для задач доступны следующие вычисляемые поля через `extra_context`:

`{executor_str}` - строковое представление исполнителя (обычно `executor.display_name`)
`{project_str}` - строковое представление проекта (обычно `project.name`)
`{url}` - URL задачи (через `task.get_absolute_url()`)

Уведомление (Notification)

Модель: `system.notifications.Notification`

Доступные поля через API:

Поле	Тип	Описание	Пример
<code>id</code>	Integer	Уникальный идентификатор	1
<code>title</code>	String	Заголовок уведомления	"Новая задача"
<code>message</code>	String	Текст уведомления	"Вам назначена новая задача"
<code>user</code>	Integer (FK)	ID пользователя	10
<code>content_type</code>	Integer (FK)	ID типа контента	15
<code>content_type_name</code>	String	Название типа контента (read-only)	"task"
<code>object_id</code>	String	ID связанного объекта	"123"
<code>cached_url</code>	String (URL)	Кэшированный URL объекта	"/tasks/123/"
<code>is_read</code>	Boolean	Прочитано ли уведомление	false
<code>created_at</code>	DateTime	Дата создания	"2024-01-15T10:00:00Z"

Пример использования:

Уведомления обычно не используются напрямую в шаблонах модулей, но доступны через API для отображения в интерфейсе модуля.

Работа с полями в шаблонах уведомлений

При создании шаблонов уведомлений в `module.json` вы можете использовать следующие способы доступа к полям:

Прямой доступ к полям объекта:

```
json { "subject": "Задача: {title}", "body": "Исполнитель: {executor.display_name}\nПроект: {project.name}" }
```

Использование вычисляемых значений через `extra_context`: В коде модуля при отправке уведомления:

```
python send_notify(obj=task, recipients=[user],
template_name='module_code_TemplateName', extra_context={ 'executor_str':
str(task.executor), 'project_str': str(task.project) if task.project else 'N/A',
'user_name': str(user.get_full_name() or user.username) } ) В шаблоне: json {
"subject": "Задача: {title}", "body": "Исполнитель: {executor_str}\nПроект:
{project_str}" }
```

Форматирование дат: Django автоматически форматирует даты в шаблонах. Для кастомного форматирования используйте фильтры Django в extra_context.

Типы данных полей

String - строка текста Integer - целое число Decimal - десятичное число (деньги, проценты) Boolean - логическое значение (true/false) Date - дата в формате YYYY-MM-DD DateTime - дата и время в формате ISO 8601 FK - внешний ключ (Foreign Key), возвращает ID связанного объекта UUID - уникальный идентификатор в формате UUID

Важные замечания

Read-only поля - поля с пометкой (read-only) вычисляются автоматически и не могут быть изменены через API. Связанные объекты - при работе с API связанные объекты (например, manager, project) возвращаются как ID. Для получения полной информации используйте отдельный запрос к API или используйте вычисляемые поля (например, manager_name, project_name). Null значения - многие поля могут быть null. Всегда проверяйте наличие значения перед использованием в шаблонах. Форматирование - для форматирования дат, чисел и других значений в шаблонах уведомлений используйте extra_context в коде модуля.

Процесс разработки модуля

1. Создание Django приложения

Создайте стандартное Django приложение: `python manage.py startapp example_module`

2. Разработка функционала

Реализуйте необходимый функционал модуля:

Модели данных Представления URL маршруты Шаблоны Статические файлы

Важно при работе с переводами:

Используйте `gettext_lazy (_())` для всех строк, которые будут отображаться пользователю При объединении списков с переведенными строками всегда преобразуйте их в обычные строки:

```
python error_strings = [str(error) for error in errors] message = ', '.join(error_strings)
```

- В миграциях используйте обычные строки, а не переведенные

3. Создание module.json

Создайте файл module.json в корне приложения с необходимой конфигурацией.

4. Упаковка модуля

Упакуйте приложение в ZIP архив: `cd example_module zip -r ../example_module.zip` .
Важно: В корне ZIP архива должен находиться файл module.json.

5. Установка модуля

Вариант 1: Через каталог модулей (рекомендуется)

Перейдите в раздел настроек -> "Modules" (/settings/modules/) Нажмите кнопку "Add Module" Выберите ZIP архив с модулем Система автоматически: Распакует архив Проверит наличие и валидность module.json Извлечет все данные модуля из конфигурации Создаст запись модуля в каталоге Создаст версию модуля с ZIP файлом После успешного добавления модуль появится в каталоге Нажмите кнопку "Install" для установки модуля
Важно: При загрузке модуля все поля (название, описание, автор и т.д.) автоматически заполняются из файла module.json. Если файл отсутствует или имеет неверный синтаксис, будет показана ошибка.

Вариант 2: Через админ-панель

Перейдите в раздел "Modules" в админ-панели Нажмите "Upload Module" Загрузите ZIP архив Модуль будет автоматически добавлен в каталог Установите модуль через каталог (/settings/modules/)

Вариант 3: Через API

POST /modules/api/modules/{id}/install/

Ограничения и рекомендации

1. Именованние: Используйте уникальные коды модулей, чтобы избежать конфликтов
2. Зависимости: Указывайте все зависимости от других модулей
3. Миграции: Всегда создавайте миграции для изменений моделей
4. Права доступа: Используйте систему прав доступа Django для защиты функционала
5. Версионирование: Следуйте семантическому версионированию (SemVer)
6. Локальные модули: Используйте для прототипирования или внутренних модулей
7. Глобальные модули: Используйте для распространения модулей через каталог

Работа с версиями модулей

Создание новой версии

Перейдите в админ-панель -> Modules Выберите модуль В разделе "Versions" добавьте новую версию: Загрузите ZIP файл с новой версией Укажите номер версии Добавьте changelog (опционально)

Обновление модуля

Убедитесь, что модуль установлен Создайте новую версию модуля в админ-панели Используйте API или админ-панель для обновления:
bash POST /modules/api/modules/{id}/update-module/ Или через админ-панель: выберите модуль -> "Update Module" Важно: При обновлении сохраняются все данные модуля. В случае ошибки выполняется откат к предыдущей версии.

Процесс установки модуля: подробное описание

При установке модуля в системе происходит следующий процесс:

1. Загрузка и валидация ZIP архива

Загрузка ZIP файла - модуль загружается через админ-панель или API
Распаковка во временную директорию - ZIP архив распаковывается во временную папку (/tmp/...) Валидация module.json: Проверка наличия обязательных полей (name, code, version, settings) Проверка структуры конфигурации Проверка версии системы (если указана min_system_version) Проверка зависимостей от других модулей Валидация структуры меню (до 3 уровней вложенности) Валидация структуры URL

2. Копирование файлов модуля и обновление apps.py

Путь установки: {BASE_DIR}/modules/installed/{app_name}/

Где:

BASE_DIR - корневая директория проекта Django app_name - имя Django приложения из module.json -> settings.app_name

Процесс копирования:

Создается директория modules/installed/ (если не существует) Если в ZIP архиве есть папка с именем модуля, используется она, иначе - корень архива Если модуль уже установлен, старая версия удаляется Все файлы модуля копируются в целевую директорию: models.py, views.py, urls.py, admin.py migrations/ - папка с миграциями templates/ - шаблоны static/ - статические файлы apps.py - конфигурация приложения Все остальные файлы модуля

Автоматическое обновление apps.py: После копирования файлов система автоматически обновляет файл apps.py модуля:

Поле name в классе AppConfig обновляется на полный путь: modules.installed.{app_name} Это необходимо для корректной работы Django, так как приложение добавляется в INSTALLED_APPS с полным путем Например, если app_name = 'task_reminder_module', то name будет изменено на 'modules.installed.task_reminder_module' Важно: Поле label остается без изменений и должно быть указано в исходном файле

Пример обновления apps.py: # До установки: class TaskReminderConfig(AppConfig): name = 'task_reminder_module' label = 'task_reminder_module' # Остается без изменений

После установки (автоматически): class TaskReminderConfig(AppConfig): name = 'modules.installed.task_reminder_module' # Обновляется автоматически label = 'task_reminder_module' # Остается без изменений

Пример структуры после установки: /home/kvasnikov/askdev/ask/ |— modules/ |
└─ installed/ | └─ example_module/ | └─ __init__.py | └─ models.py |
views.py | └─ urls.py | └─ admin.py | └─ apps.py | └─ migrations/ | }
0001_initial.py | └─ templates/ | | └─ example_module/ | └─ static/ | └─
example_module/

3. Добавление в INSTALLED_APPS

Автоматическое добавление:

Модули автоматически добавляются в INSTALLED_APPS при старте приложения Это происходит через modules/apps.py -> ModulesConfig.ready() Метод загружает все установленные и активные модули из базы данных Каждый модуль добавляется как modules.installed.{app_name}

Процесс:

При старте Django загружается ModulesConfig Вызывается метод ready() Загружаются все записи ModuleInstallation со статусом 'installed' и is_active=True Каждое приложение добавляется в settings.INSTALLED_APPS Django автоматически загружает приложения из INSTALLED_APPS

Перезапуск приложения:

После установки модуля требуется перезапуск сервера для применения изменений При следующем запуске модуль будет автоматически загружен

4. Добавление в INSTALLED_APPS и перезагрузка конфигурации

Процесс:

Приложение добавляется в `settings.INSTALLED_APPS` с полным путем: `modules.installed.{app_name}` Конфигурация приложений Django перезагружается через `apps.app_configs.clear()` и `apps.set_installed_apps()` Это необходимо, чтобы Django узнал о новом приложении перед выполнением миграций

Важно: Перезагрузка конфигурации приложений происходит автоматически во время установки модуля.

5. Выполнение миграций базы данных

Автоматическое выполнение: `python manage.py makemigrations {app_label}`
`python manage.py migrate {app_label}`

Процесс:

Если в `module.json` -> `settings.migrations = true` (по умолчанию), выполняются миграции Сначала выполняется `makemigrations` для создания новых миграций (если они есть) Затем выполняется `migrate` с использованием метки приложения (`app_label`) Метка приложения соответствует `app_name` из `module.json` -> `settings.app_name` Все миграции из папки `migrations/` применяются к базе данных Создаются необходимые таблицы, индексы и связи

В случае ошибки:

Если миграции не удалось, модуль не устанавливается Файлы модуля остаются на диске, но запись об установке не создается Необходимо исправить ошибки в миграциях и повторить установку

6. Регистрация URL маршрутов

Автоматическая регистрация:

URL автоматически регистрируются при старте приложения Это происходит через `modules/url_loader.py` -> `get_module_urlpatterns()` Функция вызывается в `config/urls.py` и добавляет URL всех установленных модулей

Процесс:

При загрузке `config/urls.py` вызывается `get_module_urlpatterns()` Загружаются все установленные и активные модули из базы данных Для каждого модуля читается конфигурация URL из `config_data.urls` URL добавляются в основной `urlpatterns` через `path()` и `include()` Поддерживаются как относительные, так и абсолютные пути импорта Поддерживаются `namespace` для URL

7. Регистрация шаблонов

Автоматическая регистрация:

Пути к шаблонам модулей автоматически добавляются в `TEMPLATES['DIRS']` при старте приложения Это происходит через `modules/apps.py` -> `ModulesConfig.ready()` Система находит все установленные модули и добавляет путь `modules/installed/{app_name}/templates/` в список директорий шаблонов

Процесс:

При старте Django загружается `ModulesConfig` Вызывается метод `ready()` Для каждого установленного модуля проверяется наличие папки `templates/` Если папка существует, её путь добавляется в `TEMPLATES['DIRS']` Django

автоматически находит шаблоны в этих директориях

Важно: Шаблоны должны находиться в папке с именем модуля: `templates/{module_name}/template.html` Формат конфигурации URL: `{ "urls": [{ "path": "example/", "include": "example_module.urls", "namespace": "example" }] }`

Перезапуск приложения:

После установки модуля требуется перезапуск сервера для применения изменений. При следующем запуске URL будут автоматически зарегистрированы.

7. Регистрация меню

Регистрация в реестре:

Конфигурация меню из `module.json` -> `menu` сохраняется в `ModuleRegistry`. Меню отображается динамически через `template tag`.

Отображение:

Меню отображается в боковой панели через `modules/templatetags/module_tags.py`. Проверяются права доступа для каждого пункта меню. Поддерживается до 3 уровней вложенности.

Перезагрузка не требуется - меню отображается динамически при каждом запросе.

8. Загрузка шаблонов уведомлений

Автоматическая загрузка:

Шаблоны уведомлений из `module.json` -> `notification_templates` загружаются в базу данных. Email шаблоны создаются в модели `EmailTemplate`, Messenger шаблоны создаются в модели `MessengerTemplate`.

Процесс:

При установке модуля читается секция `notification_templates` из конфигурации.

Для каждого шаблона формируется уникальное имя:

`{module_code}_{template_name}`. Шаблоны создаются или обновляются в базе данных. Шаблоны помечаются как системные (`system=True`). Раздел (`section`) устанавливается в название модуля, если не указан явно.

Использование:

После установки шаблоны доступны для использования через `send_notify()`.

Используйте полное имя шаблона: `{module_code}_{template_name}`. Шаблоны отображаются в разделе настроек уведомлений (`/settings/notify/`).

При обновлении модуля:

Существующие шаблоны обновляются новыми значениями. Имя шаблона остается прежним (для совместимости).

При удалении модуля:

Все шаблоны модуля удаляются из базы данных.

9. Создание записи в базе данных

Создается запись `ModuleInstallation`:

`module` - ссылка на модуль, `installed_version` - установленная версия, `status = 'installed'`, `is_active = True`, `app_name` - имя Django приложения, `install_path` - путь к установленным файлам, `config_data` - полная конфигурация из `module.json`, `installed_by` - пользователь, установивший модуль, `installed_at` - дата и время установки.

Обновляется запись Module:
is_installed = True

10. Очистка временных файлов

Временная директория с распакованным ZIP архивом удаляется Остаются только файлы в modules/installed/{app_name}/

Примеры модулей

В системе доступны следующие тестовые модули для изучения и тестирования:

1. Task Reminder Module (task_reminder_module)

Описание: Модуль для напоминаний о просроченных задачах и задачах, срок которых скоро истечет. Функционал:

Dashboard с статистикой просроченных и приближающихся задач
Список просроченных задач
Настройки напоминаний для пользователя
Отправка тестовых напоминаний

Используемые сущности системы:

Task (tasks.models.Task) - для получения задач Employee (personal.models.Employee) - для связи с сотрудниками

Шаблоны уведомлений:

task_reminder_OverdueTaskReminder (email и messenger)
task_reminder_TaskDueSoon (email и messenger)

Расположение: test_modules/task_reminder_module/

2. Inventory Management Module (inventory_module)

Описание: Модуль для управления выдачей инвентаря сотрудникам. Функционал:

Учет инвентаря (название, описание, инвентарный номер, категория)
Таблица выдачи инвентаря с фильтрами
Добавление и редактирование выдачи
Отслеживание статусов: "выдан" (issued) и "возвращён" (returned)
Список инвентаря

Модели:

Inventory - инвентарь InventoryIssue - выдача инвентаря

Используемые сущности системы:

Employee (personal.models.Employee) - для связи с сотрудниками
CustomUser (users.models.CustomUser) - для отслеживания создателя выдачи

Меню:

Issues (/inventory/) - таблица выдачи инвентаря
Inventory List (/inventory/inventory/) - список инвентаря

Расположение: test_modules/inventory_module/ Особенности:

Валидация дат (дата возврата не может быть раньше даты выдачи)
Фильтрация по статусу, сотруднику, инвентарю
Поиск по названию инвентаря,

инвентарному номеру, сотруднику

Часто встречающиеся ошибки и их решение

Ошибка: "sequence item 0: expected str instance, proxy found"

Причина: В валидаторе используются объекты `gettext_lazy` для сообщений об ошибках, которые не могут быть напрямую объединены в строку. Решение: Система автоматически преобразует все ошибки в строки перед объединением. Если вы создаете свой валидатор, убедитесь, что все сообщения об ошибках преобразуются в строки перед использованием в `join()`:
`error_strings = [str(error) for error in errors]`
`error_message = ', '.join(error_strings)`

Ошибка: "No installed app with label 'module_name'"

Причина: Модуль не был правильно добавлен в `INSTALLED_APPS` или не была выполнена перезагрузка конфигурации приложений. Решение: Убедитесь, что модуль установлен корректно. Проверьте, что файл `apps.py` обновлен с правильным `name`. Перезапустите сервер Django.

Ошибка: "column modules_module.is_local does not exist"

Причина: Миграция для добавления поля `is_local` не была применена. Решение: Выполните миграции: `python manage.py migrate modules`

Ошибка: "module.json not found in ZIP archive"

Причина: В ZIP архиве отсутствует файл `module.json` в корне или в подпапке модуля. Решение: Убедитесь, что файл `module.json` находится в корне ZIP архива или в папке с именем модуля.

Ошибка: "Module configuration is invalid"

Причина: Файл `module.json` содержит ошибки валидации. Решение: Проверьте: Наличие всех обязательных полей (`name`, `code`, `version`, `settings`) Правильность структуры JSON Наличие обязательного поля `app_name` в секции `settings` Правильность типов данных для всех полей

Ошибка: "relation 'modules_{model_name}' does not exist" или "relation '{app_name}_{model_name}' does not exist"

Причина: В моделях не указан app_label в Meta классе, или в apps.py не указан label. Решение:

Убедитесь, что в apps.py указан label:

```
python class YourModuleConfig(AppConfig): name = 'your_module_name' label = 'your_module_name' # ⚠ ОБЯЗАТЕЛЬНО
```

Убедитесь, что в Meta классах всех моделей указан app_label:

```
python class YourModel(models.Model): class Meta: app_label = 'your_module_name' # ⚠ ОБЯЗАТЕЛЬНО
```

Убедитесь, что в миграциях указан app_label:

```
python class Migration(migrations.Migration): app_label = 'your_module_name' # ⚠ ОБЯЗАТЕЛЬНО
```

Пересоберите архив модуля и переустановите его

Ошибка: "TemplateDoesNotExist at /module-url/"

Причина: Django не может найти шаблоны модуля. Это может происходить, если:

Шаблоны не находятся в правильной папке Путь к шаблонам не добавлен в TEMPLATES['DIRS'] Неправильно указан путь к шаблону в views.py

Решение:

Убедитесь, что шаблоны находятся в папке templates/{module_name}/:

```
templates/ └─ your_module_name/ └─ your_template.html
```

Убедитесь, что в views.py указан полный путь к шаблону:

```
python template_name = 'your_module_name/your_template.html'
```

Перезапустите сервер Django - система автоматически добавит пути к шаблонам при старте Если проблема сохраняется, проверьте, что модуль установлен и активен

Ошибка: "ProgrammingError: relation 'modules_{model_name}' does not exist"

Причина: Django неправильно определяет app_label для моделей, что приводит к использованию неправильного имени таблицы. Решение:

Убедитесь, что в apps.py указан label:

```
python label = 'your_module_name' # Должно совпадать с app_name из module.json
```

Убедитесь, что в Meta классах всех моделей указан app_label:

```
python class Meta: app_label = 'your_module_name' # Должно совпадать с app_name из module.json
```

Пересоберите архив модуля и переустановите его

Использование gettext_lazy в модулях

Важно: При использовании `gettext_lazy` в модулях убедитесь, что все строки преобразуются в обычные строки перед объединением или форматированием:

```
from django.utils.translation import gettext_lazy as _
```

❌ Неправильно: `errors = [_('Error 1'), _('Error 2')]` `message = ', '.join(errors)` #
Ошибка: `expected str instance, __proxy__ found`

```
# ✅ Правильно: errors = [_('Error 1'), _('Error 2')] error_strings = [str(error) for error in errors]
```

`message = ', '.join(error_strings)`

Система автоматически обрабатывает это в валидаторах, но при создании собственного кода модуля следите за этим.