M3 Core

Выпуск 2.0.9.6

BARS Group

17 September 2014

Оглавление

Содержание:

Глава 1

Общее описание

1.1 Решаемые задачи

Построение гибких масштабируемых платформонезависимых корпоративных приложений с Rich Client интерфейсом.

1.2 Введение в платформу m3

В Django все адреса запросов задаются с помощью регулярных выражений. Адреса могут быть сгруппированы по отдельным приложениям, но в итоге получается общий список всех выражений и соответсвующие им вьюшки. Он называется *urlpatterns*.

Примечание: Вьюшка - это метод в Django принимающий HttpRequest и возвращающий HttpResponse. Как правило вьюшки находятся в модуле views.py приложения. Writing views

---

При поступлении запроса, он сначала проходит через все подключенные Middleware, затем проверяется на соответсвие выражениям из *urlpatterns*. В итоге либо вызывается соответсвующая вьюшка, либо генерируется исключение 404. Так вкратце работает механизм разруливания запросов в Django URL dispatcher

Такой подход очень прост и вполне подходит для приложений уровня homepage ;) Но для корпоративных приложений появляются новые требования:

1. Отсутсвие вшитых в программу адресов (URL)
2. Изменение адресов на ходу
3. Перехват выполнения перед вьюшкой и после вьюшки
4. Автоматическое извлечение и контроль входных параметров вьюшки
5. Гибкое встраивание нового функционала

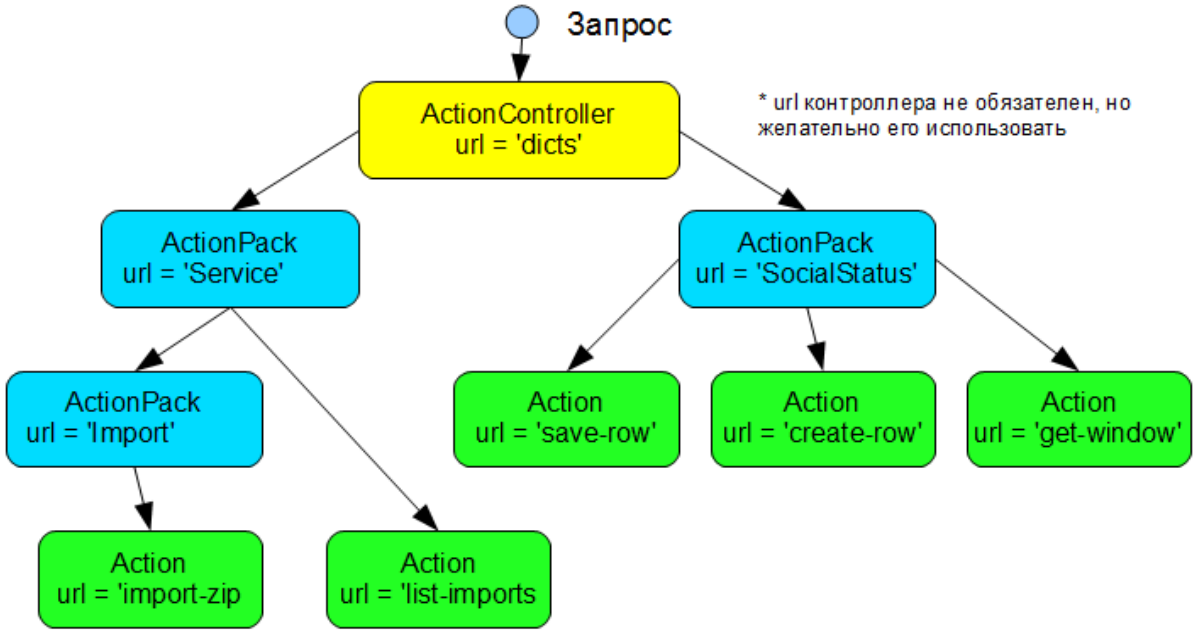
1.2.1 Концепция

Чтобы покрыть недостающий функционал URL dispatcher было принято решение - отказаться от вьюшек и использовать их только как точки входа в приложение. Т.е. одно приложение - одна вьюшка, в которую приходят все запросы приложения. Далее будем называть её - *вьюшка контроллера*.

Вьюшки были заменены классами *Action* (экшены). Экшены аналогичные по смыслу и стремящиеся к одной цели, сгруппированы в классы *ActionPack* (экшенпаки). Таким образом экшены и паки представляют собой дерево, каждый узел которого имеет свой кусочек адреса. Путь от корня до конечного экшена является полным адресом, аналогичным адресу в URL dispatcher.

Связующим звеном между деревом экшенов и вьюшкой котроллера, является класс *ActionController*. Он хранит экшенпаки и отвечает за обработку запросов. Как правило контроллер бывает один на приложение.

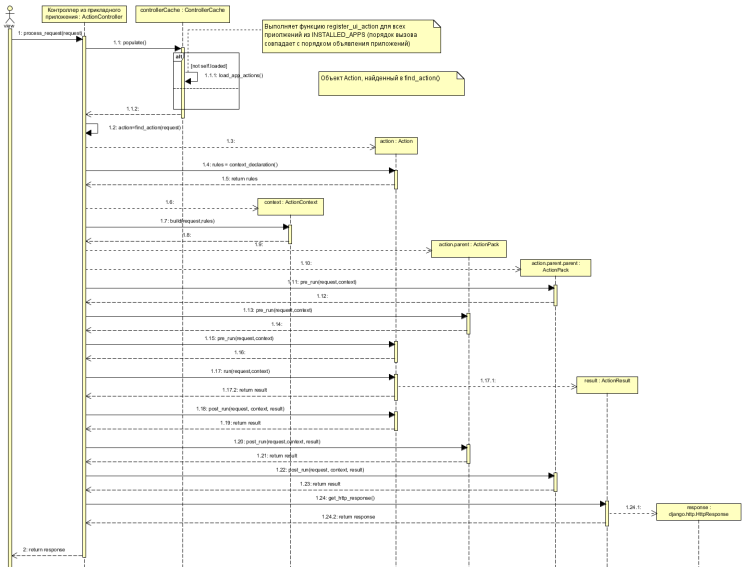
Схема формирования адресов:



Адреса получившиеся при инициализации контроллера:

- /dicts/Service/list-imports
- /dicts/Service/Import/import-zip
- /dicts/SocialStatus/save-row
- /dicts/SocialStatus/create-row
- /dicts/SocialStatus/get-window

Диаграмма обработки запроса:



Самый первый запрос к вьюшке контроллера вызывает метод populate() в ControllerCache. Этот класс отвечает за хранение глобального списка контроллеров и начальную инициализацию дерева экшенов.

После первичного формирования дерева экшенов и контроллеров, запрос отправляется к конкретному котроллеру приложения. Получив сырой запрос, он разбирает его и проходит по дереву экшенов в поисках нужного. Если не находит, то генерирует исключение 404. Если находит то выполняет все встретившиеся на пути, от корня до экшена, методы pre_run(). Выполнив экшен, в обратном порядке выполняются все методы post_run(). Причем если хоть один из них вернул результат отличный от None, обработка прерывается и результат уходит наружу, во вьюшку контроллера.

Глава 2

Установка

Самый простой способо установки - это склонировать с репозитария, обновиться до нужной ветки и запустить setup.py с командой install

Глава 3

Простое использование

3.1 Класс Action и его использование

Пример экшена, который читает даннные из POST'а. Проверяет значение C и сохраняет в БД значения A и B. Код:

```
from m3.ui.actions import Action
from m3.ui.actions.results import OperationResult
...

class MyFirstAction(Action):
    url = 'save'
    verbose_name = u'Сохранение данных'
    short_name = 'my-first-action-alias'

    def pre_run(self, request, context):
        """ pre_run удобно использовать для начальных проверок """
        value = request.POST.get('C')
        if value < 0:
            # Обработка запроса на этом прерывается
            return OperationResult(success=False, message=u'Недопустимое значение')

    def run(self, request, context):
        """ Основная работа экшена производится здесь """
        a = request.POST.get('A')
        b = request.POST.get('B')
        SomeModel.objects.create(a=a, b=b)
        return OperationResult(message=u'Данные сохранены')
```

OperationResult это один из возможных результатов работы экшена, который автоматически преобразуется в Django HttpResponse внутри контроллера. В нашем случае класс *OperationResult* указывает успешно ли завершилась операция *success* и несет соответствующее сообщение *message*.

3.2 Класс ActionPack и его использование

Экшены и паки входящие внутрь нашего пака задаются с помощью атрибутов *actions* и *subpacks*. Пример:

```
from m3.ui.actions import ActionPack
```

```
class MyFormActionPack(ActionPack):
    def __init__(self):
        super(MyFormActionPack, self).__init__()
        self.actions.extend([
            GetWindowAction, GetDataAction(), SaveAction, DeleteAction()
        ])
        self.subpacks.append( InnerActionPack )

class InnerActionPack(ActionPack):
    url = 'inner'
    def __init__(self):
        super(InnerActionPack, self).__init__()
        self.actions.extend([
            SubAction1, SubAction2, ...
        ])
```

Обратите внимание, что в список *actions* можно добавлять как классы, так и экземпляры экшенов. На самом деле это не имеет значения, т.к. контроллер при инициализации автоматически создает экземпляры экшенов и паков, а также дополняет их специальными атрибутами. Об этом в главе *Класс ActionController и его использование*

Часто бывает необходимо получить прямой доступ к экшенам внутри пака через атрибуты. Живой пример из М3:

```
class BaseDictionaryActions(ActionPack):
    def __init__(self):
        super(BaseDictionaryActions, self).__init__()
        self.list_window_action   = DictListWindowAction()
        self.select_window_action = DictSelectWindowAction()
        self.edit_window_action   = DictEditWindowAction()
        ...
        self.actions = [
            self.list_window_action,
            self.select_window_action,
            self.edit_window_action,
            ...
        ]
```

3.3 Класс ActionController и его использование

Определение контроллера состоит из нескольких этапов:

1. Экземпляр контроллера как правило создается в файле *app_meta.py* внутри приложения.
2. Чтобы передавать в него запросы Django нужно создать вьюшку контроллера *dict_view* и зарегистрировать url pattern для неё в методе *register_urlpatterns*. Он вызывается автоматически для всех приложений.
3. Регистрация паков в контроллере производится методом *register_actions*.

Пример файла app_meta.py внутри приложения dicts:

```
# Именованный экземпляр контроллера
dict_controller = ActionController(url='/core-dicts', name=u'Справочники')

def register_urlpatterns():
    """ Регистрация вьюшки контроллера """
    return urls.defaults.patterns('',
```

```
            (r'^core-dicts/', 'mis.core.dicts.app_meta.dict_view'),
        )

    def dict_view(request):
        """ Вьюшка контроллера """
        return dict_controller.process_request(request)

    def register_actions():
        """ Регистрация паков в контроллере """
        dict_controller.packs.extend([
            MyFormActionPack,
            MyDictionaryActions
        ])
```

При добавлении пака в контроллер вызывается метод *_build_pack_node*, который создает экземпляр пака и всех вложенных в него экшенов и паков. Заполняются служебные атрибуты, с помощью которых можно обходить дерево:

- parent - ссылка на родительский пак
- controller - ссылка на родительский контроллер

Чаще всего бывает нужно найти экшен или пак в уже сформированной иерархии. Для этого есть несколько методов:

- По известному адресу экшен можно найти методом *get_action_by_url*
- Пак по имени или классу можно найти методом *find_pack*
- Рекомендуется искать экшены и паки по короткому имени (псевдониму). Для него есть атрибут short_name.

Пример:

```
from m3.helpers import urls

my_action = urls.get_action('my-action-alias')
my_pack = urls.get_pack('my-pack-alias')
```

3.3.1 Контекст запросов в M3 (m3.actions.context)

В Django параметры запросов передаются с помощью класса HttpRequest в трех словарях POST, GET и REQUEST. Извлекаемые из них значения не проверяются и не конвертируются в сложные типы Python. Ещё один минус, что извлечение, как правило, делают внутри кода вьюшки, таким образом загромождается место под бизнес логику.

В M3 был разработан механизм контекста, который позволяет:

- Автоматически извлекать значения из запроса по заранее заданным правилам.
- Преобразовывать сырое значение в указанный в правилах тип.
- Передавать контекст в ответе к визуальным компонентам.

3.4 Определение контекста в экшенах

Правила извлечения контекста задаются внутри метода *context_declaration* экшена. Представляют собой список экземпляров класса *ActionContextDeclaration* или его упрощенное написание *ACD*. Так же

можно использовать декларативное описание по определенной схеме

`Action.context_declaration()`

Метод объявления необходимости наличия определенных параметров в контексте.

Должен возвращать список из экземпляров *ActionContextDeclaration* либо словарь описания контекста для *DeclarativeActionContext*

Результат описание необходимости наличия определенных параметров в запросе

Тип результата list of m3_core.actions.context.ActionContextDeclaration либо m3_core.actions.context.DeclarativeActionContext

class `m3.actions.context.ActionContextDeclaration`(*name=''*, *default=None*, *type=None*, *required=False*, *verbose_name=''*, **args*, ***kwargs*)

Класс, который определяет правило извлечения параметра из запроса и необходимость его наличия в объекте контекста ActionContext.

Параметры

- **name** (*str*) – имя параметра
- **type** – тип извлекаемого значения
- **required** (*bool*) – указывает что параметр обязательный
- **default** – значение параметра по умолчанию, используется если его нет в запросе, но наличие обязательно
- **verbose_name** (*unicode*) – человеческое имя параметра, необходимо для сообщений об ошибках

`human_name()`

Возвращает человеческое название параметра *verbose_name*

Пример определения правил:

```
from m3.actions.context import ActionContextDeclaration, ACD
...

class GetRowsAction(Action):
    def context_declaration(self):
            return [
                ActionContextDeclaration(name='id', type=int, required=True, verbose_name=u'Идентификатор модели'),
                ActionContextDeclaration(name='start', type=int, required=True),
                ActionContextDeclaration(name='limit', type=int, required=True)
            ]
        ...

class GetRowsAction(Action):
    url = 'save'

    def context_declaration(self):
        # декларативное опимание контекста
        return {
            # параметр запроса -> параметры разбора
            'ids': {
                # значение по умолчанию,
                # используется при отсутствии параметра в запросе
                # если не указано - параметр считается обязательным
                'default': '',
```

```
                # тип парсера, может быть:
                # - строкой - именем одного из предопределенных парсеров
                # - callable-объектом, выполняющим парсинг. Такой объект
                #   может возбуждать ValueError/TypeError/KeyError/IndexError
                #   в случае неправильного формата данных, что позволяет
                #   использовать в качестве парсера что-то вроде:
                #     'type': ['on', 'yes'].__contains__
                #     'type': {1: 'Male', 2: 'Female'}.get
                #     'type': float
                #     'type': json.loads
                'type': int

                # наименование параметра, понятное пользователю
                # используется в сообщениях об ошибках
                'verbose_name': u'Идентификатор объекта'
            },
            #
            'date': {
                'type': datetime.date,
                'verbose_name': u'Дата начала действия изменений'
            },
            'comment': {
                'type': str
            }
        }

    @transaction.commit_on_success
    def run(self, request, context):
        if not hasattr(context, 'comment'):
            context.comment = u'Без комментариев'

        for id in context.ids:
            SomeModel.objects.create(pid=id, comment=context.comment)

        msg = u'Изменения внесены на дату %s' % context.date
        return OperationResult(message=msg)
```

Разберем правило:

```
ACD(name='date', required=True, type=datetime.date, verbose_name=u'Дата начала действия изменений')
```

Из словаря REQUEST запроса HttpRequest будет извлечено значение с именем “date” и приведено к типу datetime.date. Если его нет в REQUEST, то до run() управление не дойдет и будет сгенерировано исключение RequiredFailed.

Разберем правило:

```
ACD(name='comment', type=str)
```

Из словаря REQUEST запроса HttpRequest будет извлечено значение с именем “comment”. Если его нет в REQUEST, то соответствующий атрибут не будет добавлен в context, т.к. required=False по умолчанию. Управление будет передано в run().

3.5 Возможные результаты работы экшенов (m3.actions.results)

В Django в качестве конечного ответа вьюшек используется класс HttpResponse. В M3, при работе с экшенами и паками, используются более высокие абстракции - класс производные от *ActionResult*.

Главные отличия ActionResult от HttpResponse:

- ActionResult используется для хранения и трансформации ответа приложения на запрос. HttpResponse же напротив является готовым ответом и содержит в себе данные специфичные для протокола http, например status_code и cookie. Которые не нужны при написании бизнес-логики.
- Тип ответа в ActionResult определяется классом и интерфейсом им предоставляемым, а в HttpResponse только mimetype.
- ActionResult поддерживает передачу контекста
- ActionResult преобразуется в HttpResponse после обработки запроса в контроллере с помощью метода *get_http_response*.

Благодаря этим особенностям в процессе обработки запроса можно изменять и даже подменять ответы. Например, подключенный плагин может модифицировать форму, передаваемую через ExtUIScriptResult, добавив в нее новые контролы.

class `m3.actions.results.ActionResult`(*data=None*, *http_params={}*)

`get_http_response`()

Результат соответствующий данному результату выполнения действия ответ

Тип результата django.http.HttpResponse

`process_http_params`(*response*)

Добавляет параметры http в ответ

Параметры response (*наследник m3_core.actions.results.ActionResult*) – ответ, в который добавляются параметры

Результат http-ответ с добавленными параметрами

Тип результата наследник m3_core.actions.results.ActionResult

От него наследуется более сложный базовый класс BaseContextedResult, который может передавать контекст в визуальные компоненты и формы.

class `m3.actions.results.BaseContextedResult`(*data=None*, *context=None*, *http_params={}*)

Абстрактный базовый класс, который оперирует понятием результата выполнения операции, 'отягощенного некоторым контектом'

3.5.1 Простые обёртки над HttpResponse

class `m3.actions.results.HttpReadyResult`(*data=None*, *http_params={}*)

Результат выполнения операции в виде готового HttpResponse. Для данного класса в *data* храниться объект класса HttpResponse.

class `m3.actions.results.TextResult`(*data=None*, *http_params={}*)

Результат, данные *data* которого напрямую передаются в HttpResponse

class `m3.actions.results.XMLResult`(*data=None*, *http_params={}*)

Результат в формате xml, данные которого напрямую передаются в HttpResponse

3.5.2 Ответы передающие JSON

Предназначены для работы с JSON и готовыми к JSON-сериализации данными

class m3.actions.results.JsonResult(*data=None*, *http_params={}*)
Результат выполнения операции в виде готового JSON объекта для возврата в response. Для данного класса в *data* храниться строка с данными JSON объекта.

class m3.actions.results.PreJsonResult(*data=None*, *secret_values=False*, *dict_list=None*)
Результат выполнения операции в виде, например, списка объектов, готовых к сериализации в JSON формат и отправке в HttpResponse. В *data* передается объект для сериализации. В *dict_list* указывается список объектов и/или атрибутов вложенных объектов для более глубокой сериализации. Смотри класс т3.core.json.M3JSONEncoder. Параметр специфичный для проекта *secret_values* - используется чтобы указать, что передаются персональные обезличенные данные и их расшифровать перед отправкой клиенту.

3.5.3 Результат выполнения операции

class m3.actions.results.OperationResult(*success=True*, *code=''*, *message=''*, **args*, ***kwargs*)
Результат выполнения операции, описанный в виде Ajax результата ExtJS: success или failure. В случае если операция выполнена успешно, параметр *success* должен быть True, иначе False.

Параметры

- **success** (*boolean*) – флаг успеха операции
- **message** (*unicode*) – сообщение, поясняющее результат выполнения операции.
- **code** (*unicode*) – текст javascript, который будет выполнен на клиенте в результате обработки результата операции.

static by_message(*message*)
Возвращает экземпляр OperationResult построенный исходя из сообщения *message*. Если сообщение не пустое, то операция считается проваленной и success=False, иначе операция считается успешной success=True.

Параметры **message** (*unicode*) – текст сообщения об ошибке, или не указан

get_http_response()
Возвращает объект HttpResponse, соответствующий данному результату выполнения операции

Результат http-ответ, соответствующий данному результату

Тип результата django.http.HttpResponse

Состав модуля

4.1 Основные объекты библиотеки: механизмы проверки прав, экшены, паки, контроллеры, кэш контроллеров

class m3.actions.__init__.AbstractPermissionChecker
Абстрактный механизм проверки прав

static get_perm_code(*action_or_pack*, *subpermission=None*)
Возвращает код действия, для контроля прав доступа

Параметры

- **action_or_pack** (*m3_core.actions.Action либо m3_core.actions.ActionPack*) – экшен или пак, для которого возвращается код доступа
- **subpermission** (*basestring*) – код подправа

Результат код действия для контроля прав доступа

Тип результата basestring

has_action_permission(*request*, *action*, *subpermission=None*)
Метод должен возвращать True, если выполнение экшена допустимо в контексте запроса

Параметры

- **request** (*django.http.Request*) – запрос
- **action** (*m3_core.actions.Action*) – Экшен, наличие прав на выполнение которого проверяется

Результат Допустимость выполнения экшена в контесте запроса

Тип результата True или False

has_pack_permission(*request*, *pack*, *permission*)
Метод должен возвращать True, если действие, характеризуемое парой pack/permission, допустимо в контексте запроса

Параметры

- **request** (*django.http.Request*) – запрос
- **pack** (*m3_core.actions.ActionPack*) – пак, для которого возвращается код доступа

Результат Допустимость пака в контесте запроса

Тип результата True или False

class m3.actions.__init__.Action

Базовый класс, от которого должны наследоваться все Action'ы в системе. Заменяет собой классические вьюшки Django.

classmethod absolute_url()

Возвращает полный путь до действия. НО при условии что этот экшен используется ТОЛЬКО В ОДНОМ ПАКЕ И КОНТРОЛЛЕРЕ, иначе валим всех! Ищет перебором!

Результат полный путь до действия

Тип результата basestring

context_declaration()

Метод объявления необходимости наличия определенных параметров в контексте.

Должен возвращать список из экземпляров *ActionContextDeclaration* либо словарь описания контекста для *DeclarativeActionContext*

Результат описание необходимости наличия определенных параметров в запросе

Тип результата list of m3_core.actions.context.ActionContextDeclaration либо m3_core.actions.context.DeclarativeActionContext

controller = **None**

Ссылка на контроллер к которому принадлежит данный Action

get_absolute_url()

Возвращает полный путь от хоста до конечного экшена @deprecated: Дублирует absolute_url

get_packs_url()

Возвращает строку, полный адрес от контроллера до текущего экшена

Результат полный адрес от контроллера до текущего экшена

Тип результата basestring

get_perm_code(*subpermission=None*)

Получение кода (под)права

has_perm(*request*, *subpermission=None*)

Интерфейсный метод проверки (под)прав экшна.

Использовать ВМЕСТО has_permission/has_sub_permission

Параметры **request** (*django.http.Request*) – запрос

need_check_permission = **False**

Признак обработки прав доступа, при выполнении действия (по-умолчанию отключен) Как обрабатывается этот признак - смотри в has_permission

parent = **None**

Ссылка на ActionPack к которому принадлежит данный Action

path = **None**

Логический путь действия в прикладной системе. Используется только для отображения и группировки действий с одинаковым путем. Также может использоваться для создания меню. Например, путь может быть: "СправочникиОбщие" или "Реестры"

post_run(*request*, *context*, *response*)

Метод для постобработка результата работы экшенаа.

Параметры

- **request** (*django.http.Request*) – запрос
- **context** (*m3_ core.actions.context.ActionContext*) – Контекст выполнения операции, востанавливаемый из контекстов
- **response** (*наследник m3_ core.actions.results.ActionResult*) – результат выполнения экшена

`pre_run`(*request*, *context*)

Метод для предварительной обработки входящего запроса и контекста, перед передачей в run(). Если возвращает значение отличное от None, обработка запроса прекращается и результат уходит во вьюшку контроллера.

Параметры

- **request** (*django.http.Request*) – запрос
- **context** (*m3_ core.actions.context.ActionContext*) – Контекст выполнения операции, восстанавливаемый из контекстов

Результат None либо объект для обработке во вьюшке контроллера

`run`(*request*, *context*)

Обеспечивает непосредственное исполнение запроса (аналог views в Django). Обязательно должен быть перекрыт в наследнике.

Параметры

- **request** (*django.http.Request*) – запрос
- **context** (*m3_ core.actions.context.ActionContext*) – Контекст выполнения операции, востанавливаемый из контекстов

Результат результат выполнения экшена

Тип результата наследник m3_core.actions.results.ActionResult

`sub_permissions` = **{}**

Словарь внутренних прав доступа, используемых в действии ключ - код права, который совмещается с кодом действия значение - наименование права Пример: {'tab2':u'Редактирование вкладки Доп. сведения', 'work_visible':u'Просмотр сведений о работе'} Общий код права доступа будет иметь вид: /edit#tab2 и /edit#work_visible соответственно Как обрабатывается этот список - смотри в has_sub_permission

`url` = ''

Часть адреса запроса которая однозначно определяет его принадлежность к конкретному Action'y

`verbose_name` = **None**

Наименование действия для отображения

class `m3.actions.__init__.ActionController`(*url=''*, *name=None*)

Класс коонтроллер - обеспечивает обработку пользовательских запросов путем передачи их на исполнение соответствущим Action'ам

class `FakePacks`

Класс содержит заглушки методов, чтобы инспектор кода не ругался, т.к. настоящие методы присваиваются классу в рантайме.

`ActionController.append_pack`(*pack*)

Добавляет пак в контроллер.

Параметры **pack** (*m3_ core.actions.ActionPack*) – пак, который добавляется в контроллер

`ActionController.build_context(request, rules)`
Выполняет построение контекста вызова операции ActionContext на основе переданного request

Параметры

- **request** (*django.http.Request*) – запрос
- **rules** (*dict*) – построение контекста

Результат пустой экземпляр контекста

Тип результата m3_core.actions.context.DeclarativeActionContext() или m3_core.actions.context.ActionContext

`ActionController.dump_urls()`
Отладочный метод. Выводит в консоль список всех адрес зарегистрированных в контроллере.

`ActionController.extend_packs(packs)`
Производит массовое добавление экшенпаков в контроллер.

Параметры packs (*список объектов m3_core.actions.ActionPack*) – список паков, которые необходимо зарегистрировать в контроллере

`ActionController.get_action_by_url(url)`
Получить Action по url

`ActionController.get_action_url(type)`
Возвращает полный URL адрес для класс или имени класса экшена *action*

`ActionController.get_packs()`
Возвращение всех паков в контроллере

`ActionController.get_top_actions()`
Получение списка действий или наборов, находящихся на первом уровне

`ActionController.packs` = **None**
ДЛЯ СОВМЕСТИМОСТИ. Имитирует список паков торчащий наружу

`ActionController.process_request(request)`
Обработка входящего запроса *request* от клиента. Обрабатывается по аналогии с UrlResolver'ом Django

Параметры request (*django.http.Request*) – запрос

Результат результат выполнения экшена

Тип результата наследник m3_core.actions.results.ActionResult

Raise http.Http404

`ActionController.reset()`
Сброс всего, что наделал контроллер с паками и экшенами

`ActionController.urlpattern`
Возвращает кортеж вида (pattern, method), пригодный для регистрации в urlpatterns Django

`ActionController.verbose_name` = **None**
Наименование Контроллера для отображения

`ActionController.wrap_action(dest_pack, dest_action, wrap_pack)`
Вставляет перед экшеном dest_action, входящим в пак dest_pack, промежуточный пак wrap_pack.

ВНИМАНИЕ! Экшены как правило обращаются к своим пакам через атрибут “parent”, поэтому, вероятно, будут возникать ошибки, из-за того, что оборачивающий пак не предоставляет методы изначального пака. Оборачивающий пак можно наследовать от оригинального, но тогда вместо оборачивая целесообразно использовать подмену паков.

Параметры

- **dest_pack** (*m3_core.actions.ActionPack*) – Пак в который входит оборачиваемый экшен
- **dest_action** (*m3_core.actions.Action*) – Оборачиваемый экшен
- **wrap_pack** (*m3_core.actions.ActionPack*) – Оборачивающий пак

`ActionController.wrap_pack(`*dest_pack*, *wrap_pack*`)`
Вставляет экшенпак wrap_pack внутрь иерархии перед dest_pack. Таким образом можно перехватывать запросы и ответы пака dest_pack.

Параметры

- **dest_pack** (*m3_core.actions.ActionPack*) – Пак который будем оборачивать
- **wrap_pack** (*m3_core.actions.ActionPack*) – Оборачивающий пак

class `m3.actions.__init__.ActionPack`
Базовый класс для всех ActionPack’ов. Предназначен для хранения в себе других экшенов и паков, схожих по целям.

classmethod `absolute_url()`
Возвращает полный адрес (url) от контроллера до текущего экшенпака

`actions` = **None**
Список действий зарегистрированных на исполнение в данном пакете

`controller` = **None**
Ссылка на родительский ActionController

`get_absolute_url()`
Возвращает абсолютный путь (НОРМАЛЬНО, в отличие от absolute_url)

`get_perm_code(`*permission=None*`)`
Получение кода (под)права

classmethod `get_short_name()`
Имя пака для поиска в ControllerCache

Результат Имя пака

Тип результата basestring

classmethod `get_verbose_name()`
Получение понятного имени пака

Тип результата unicode

`has_perm(`*request*, *permission*`)`
Интерфейсный метод проверки подправ пака.

Использовать ВМЕСТО has_sub_permission

Параметры **request** (*django.http.Request*) – запрос

`need_check_permission` = **False**
Признак обработки прав доступа, при выполнении дочерних действий (по-умолчанию отключен) Как обрабатывается этот признак - смотри в Action.has_permission

parent = **None**
Ссылка на вышестоящий пакет, тот в котором зарегистрирован данный пакет

path = **None**
Логический путь набора действий в прикладной системе. Используется только для отображения и группировке наборов с одинаковым путем. Также может использоваться для создания меню. Например, путь может быть: “СправочникиОбщие” или “Реестры”

post_run(*request*, *context*, *response*)
Метод для постобработки результата работы вышестоящего экшена или пака. Принимает исходный запрос *request*, результат работы *response* и извлеченный контекст *context*.

Параметры

- **request** (*django.http.Request*) – запрос
- **context** (*m3_ core.actions.context.ActionContext*) – Контекст выполнения операции, восстанавливаемый из контекств
- **response** (*m3_ core.actions.results.ActionResult*) – ответ, полученный в результате выполнения действия

pre_run(*request*, *context*)
Метод для предварительной обработки входящего запроса и контекста перед передачей в нижестоящий экшен или пак. Если возвращает значение отличное от None, обработка запроса прекращается и результат уходит во вьюшку контроллера.

Параметры

- **request** (*django.http.Request*) – запрос
- **context** (*m3_ core.actions.context.ActionContext*) – Контекст выполнения операции, восстанавливаемый из контекств

sub_permissions = **{}**
Словарь внутренних прав доступа, используемых в наборе действий ключ - код права, который совмещается с адресом (кодом) набора действий значение - наименование права Пример: {‘edit’:u’Редактирование записи’} Общий код права доступа будет иметь вид: /users#edit Как обрабатывается этот список - смотри в has_sub_permission

subpacks = **None**
Список дочерних пакетов (подпакетов), зарегистрированных на исполнение в данном пакете

verbose_name = **None**
Наименование Набора действий для отображения

class m3.actions.__init__.AuthUserPermissionChecker
Backend, проверяющий права через django.contrib.auth.models.User.has_perm

Требует подключения в проекте session-middleware, т.к. опирается на наличие в request атрибута ‘user’, указывающего на текущего пользователя

class m3.actions.__init__.BypassPermissionChecker
Механизм проверки прав, “разрешающий всем и всё”

class m3.actions.__init__.ControllerCache
Внутренний класс платформы, который отвечает за хранение кеша контроллеров и связанных с ним экшенов и паков.

classmethod dump_urls()
Отладочный метод. Выводит в консоль адреса всех контроллеров зарегистрированных в кэше.

classmethod `find_action(`*action*`)`
Ищет заданный экшен по имени класса или классу во всех зарегистрированных контроллерах. Возвращает экземпляр первого найденного экшена.

Параметры **action** (*строка формата package.Class или класс*) – имя класса или класс экшена

Результат экземпляр найденного пака

Тип результата m3_core.actions.Action

classmethod `find_node_by_perm(`*perm*`)`
Возвращает экшн/пак по коду права, или None, если таковых не найдено :param string perm: код права :return: экземпляр найденного пара/экшна

classmethod `find_pack(`*pack*`)`
Ищет заданный пак по имени класса или классу во всех зарегистрированных контроллерах. Возвращает экземпляр первого найденного пака.

Параметры **pack** (*строка формата package.Class или класс*) – имя класса или класс пака

Результат экземпляр найденного пака

Тип результата m3_core.actions.ActionPack

classmethod `get_action_by_url(`*url*`)`
Возвращает Action по переданному *url*

classmethod `get_action_url(`*type*`)`
Возвращает URL экшена *type* по его имени или классу

classmethod `get_controllers()`
Возвращает множество всех контроллеров зарегистрированных в кэше

classmethod `populate()`
Загружает в кэш ActionController'ы из перечисленных в INSTALLED_APPS приложений. В каждом из них загружает модуль *app_meta* и пытается выполнить метод *register_actions* внутри него. Выполняется только один раз. Возвращает истину в случае успеха.

Результат флаг успешного завершения

Тип результата boolean

classmethod `register_controller(`*controller*`)`
Выполняет регистрацию контроллера *controller* во внутреннем кеше.

classmethod `require_update()`
Сбрасывает внутренний флаг заполненности контроллера. Следующий запрос к контроллеру вызовет перестройку иерархии экшенов и паков.

class `m3.actions.__init__.LazyContainer(`*fabric*`)`
Ленивая обёртка для объектов, указываемых в settings. Предназначена для позднеё загрузки объектов - по первому обращению

class `m3.actions.__init__.LegacyPermissionChecker`
Мех-м проверки прав, созданный для совместимости со старыми проектами

`has_action_permission(`*request*, *action*, *subpermission=None*`)`
Проверка пака на выполнение действия для указанного пользователя

`has_pack_permission(`*request*, *pack*, *permission*`)`
Проверка на внутреннее право пака для указанного пользователя

4.2 Модуль, реализущий работу с контекстом выполнения операции

class m3.actions.context.ActionContext(***kwargs*)

Контекст выполнения операции, восстанавливаемый из запроса.

exception RequiredFailed(*reason*)

Для совместимости

class ActionContext.ValuesList(*separator=', ', type=<type 'int'>, allow_empty=True*)

Класс для описания параметров, которые будут передаваться в виде списка значений, разделенных определенным символом

ActionContext.build(*request*, *rules*)

Выполняет заполнение собственных атрибутов согласно переданному запросу, исходя из списка правил

:param request:запрос, на основе которого производится заполнение контекста

Параметры **rules** (*список m3_core.actions.context.ActionContextDeclaration*) – правила извлечения контекста из запроса

ActionContext.check_required(*rules*)

Проверяет наличие обязательных параметров

Параметры **rules** (*список m3_core.actions.context.ActionContextDeclaration*) – правила извлечения контекста из запроса

Raise ActionContext.RequiredFailed

ActionContext.combine(*context*)

Объединение контекстов друг с другом.

Параметры **context** (*m3_core.actions.context.ActionContext*) – контекст, который объединяеться с текущим

Результат новый экземпляр контекста, который получился в результате слияния с текущим. Все существовавшие значения сохраняются.

Тип результата m3_core.actions.context.ActionContext

ActionContext.convert_value(*raw_value*, *arg_type*)

Возвращает значение *raw_value*, преобразованное в заданный тип *arg_type*

ActionContext.json()

Рендеринг контекста в виде javascript объекта

class m3.actions.context.ActionContextDeclaration(*name=''*, *default=None*, *type=None*, *required=False*, *verbose_name=''*, **args*, ***kwargs*)

Класс, который определяет правило извлечения параметра из запроса и необходимость его наличия в объекте контекста ActionContext.

Параметры

- **name** (*str*) – имя параметра
- **type** – тип извлекаемого значения
- **required** (*bool*) – указывает что параметр обязательный

- **default** – значение параметра по умолчанию, используется если его нет в запросе, но наличие обязательно
- **verbose_name** (*unicode*) – человеческое имя параметра, необходимо для сообщений об ошибках

`human_name()`
Возвращает человеческое название параметра *verbose_ name*

exception `m3.actions.context.ActionContextException`
Базовый класс для исключений контекста

exception `m3.actions.context.ContextBuildingError(`*requiremets=None*, *errors=None*`)`
Ошибка построения контекста

exception `m3.actions.context.ConversionFailed(`*value*, *type*, **args*, ***kwargs*`)`
Исключение, которое выбрасывается, если значение из запроса *value* не удалось привести к типу *type*, указанному в правиле ActionContextDeclaration

exception `m3.actions.context.CriticalContextBuildingError(`*requiremets=None*, *errors=None*`)`
Критическая ошибка построения контекста

class `m3.actions.context.DeclarativeActionContext(`***kwargs*`)`
ActionContext, использующий декларативное описание контекста

`build(`*request*, *rules*`)`
Выполняет заполнение собственных атрибутов согласно переданному запросу, исходя из списка правил

:param request:запрос, на основе которого производится заполнение контекста

Параметры **rules** (*список m3_ core.actions.context.ActionContextDeclaration*) – правила извлечения контекста из запроса

Raise TypeError, ContextBuildingError, CriticalContextBuildingError

classmethod `matches(`*data*`)`
Возвращает True, если объект data “похож” на правила для DeclarativeActionContext :param data: проверяемый объект :param type: object

Результат True, если data “похож” на набор правил

Тип результата boolean

classmethod `register_parser(`*name*, *parser*`)`
Регистрация парсера

Параметры

- **parser** (*callable-object*) – парсер
- **name** (*unicode*) – имя, подо которым регистрируется парсер

exception `m3.actions.context.RequiredFailed(`*reason*`)`
Исключительная ситуация, которая выбрасывается в случае если фактическое наполнение контекста действия не соответствует описанным правилам

4.3 Результаты выполнения экшенов

class m3.actions.results.ActionRedirectResult(*action*, *context=None*)
Перенаправляет обработку запроса на другой экшен. Экшен предварительно находится с помощью метода ActionController.get_action_url()

class m3.actions.results.BaseContextedResult(*data=None*, *context=None*, *http_params={}*)
Абстрактный базовый класс, который оперирует понятием результата выполнения операции, 'отягощенного некоторым контектом'

class m3.actions.results.HttpReadyResult(*data=None*, *http_params={}*)
Результат выполнения операции в виде готового HttpResponse. Для данного класса в *data* храниться объект класса HttpResponse.

class m3.actions.results.JsonResult(*data=None*, *http_params={}*)
Результат выполнения операции в виде готового JSON объекта для возврата в response. Для данного класса в *data* храниться строка с данными JSON объекта.

class m3.actions.results.OperationResult(*success=True*, *code=''*, *message=''*, **args*, ***kwargs*)
Результат выполнения операции, описанный в виде Ajax результата ExtJS: success или failure. В случае если операция выполнена успешно, параметр *success* должен быть True, иначе False.

Параметры

- **success** (*boolean*) – флаг успеха операции
- **message** (*unicode*) – сообщение, поясняющее результат выполнения операции.
- **code** (*unicode*) – текст javascript, который будет выполнен на клиенте в результате обработки результата операции.

static by_message(*message*)
Возвращает экземпляр OperationResult построенный исходя из сообщения *message*. Если сообщение не пустое, то операция считается проваленной и success=False, иначе операция считается успешной success=True.

Параметры **message** (*unicode*) – текст сообщения об ошибке, или не указан

get_http_response()
Возвращает объект HttpResponse, соответствующий данному результату выполнения операции

Результат http-ответ, соответствующий данному результату

Тип результата django.http.HttpResponse

class m3.actions.results.PreJsonResult(*data=None*, *secret_values=False*, *dict_list=None*)
Результат выполнения операции в виде, например, списка объектов, готовых к сериализации в JSON формат и отправке в HttpResponse. В *data* передается объект для сериализации. В *dict_list* указывается список объектов и/или атрибутов вложенных объектов для более глубокой сериализации. Смотри класс т3.core.json.M3JSONEncoder. Параметр специфичный для проекта *secret_values* - используется чтобы указать, что передаются персональные обезличенные данные и их расшифровать перед отправкой клиенту.

class m3.actions.results.TextResult(*data=None*, *http_params={}*)
Результат, данные *data* которого напрямую передаются в HttpResponse

class m3.actions.results.XMLResult(*data=None*, *http_params={}*)
Результат в формате xml, данные которого напрямую передаются в HttpResponse

4.4 Исключения, возникающие при работе контроллера

exception m3.actions.exceptions.ActionNotFoundException(*clazz*, **args*, ***kwargs*)
Возникает в случае, если экшен не найден ни в одном контроллере

exception m3.actions.exceptions.ActionPackNotFoundException(*clazz*, **args*, ***kwargs*)
Возникает в случае, если пак не найден ни в одном контроллере

exception m3.actions.exceptions.ActionUrlIsNotDefined(*clazz*, **args*, ***kwargs*)
Возникает если в классе экшена не задан атрибут url. Это грозит тем, что контроллер не сможет найти и вызвать данный экшен при обработке запросов.

exception m3.actions.exceptions.ApplicationLogicException(*message*)
Исключительная ситуация уровня бизнес-логики приложения.

exception m3.actions.exceptions.ReinitException(*clazz*, **args*, ***kwargs*)
Возникает, если из-за неправильной структуры паков один и тот же экземпляр экшена может быть повторно инициализирован неверными значениями.

4.5 Хелперы для отработки расширяемых конфигураций url'ов

class m3.actions.urls.ActionsNameCache(**args*, ***kwargs*)
Кеш, используемый для хранения соответствия имен экшенов и паков соответствующим пакам

handler(*cache*, *dimentions*)
Хендлер сборки кеша

class m3.actions.urls.PacksNameCache(**args*, ***kwargs*)
Кеш, используемый для хранения соответствия имен экшенов и паков соответствующим пакам

handler(*cache*, *dimentions*)
Хендлер сборки кеша

m3.actions.urls.get_action(*action_name*)
Возвращает полный класс экшена, объявленного с указанным квалифицирующим именем.

m3.actions.urls.get_acton_url(*action*)
Возвращает абсолютный путь до

m3.actions.urls.get_app_urlpatterns()
Возвращает конфигурацию урлов, объявленных в app_meta приложений.

Данная функция не проглатывает ошибки, а выбрасывает все наружу. Перехват исключительных ситуаций данной фунции необходимо осуществлять вручную в urls.py прикладных приложений

m3.actions.urls.get_pack(*pack_name*)
Получает экшенпак по имени

m3.actions.urls.get_pack_by_url(*url*)
Возвращает набор экшенов по переданному url

m3.actions.urls.get_pack_instance(*pack_name*)
Получает экземпляр набора экшенов по имени из контроллеров

m3.actions.urls.get_pack_url(*pack_name*)
Возвращает абсолютный путь для набора экшенов

m3.actions.urls.get_url(*action*)
Возвращает абсолютный путь до

m3.actions.urls.inner_name_cache_handler(*for_actions=True*)
Внутренний метод обхода дерева паков и экшенов. Используется в хендлерах сборки кешей

4.6 Вспомогательные функции используемые в паках

m3.actions.utils.apply_column_filter(*query*, *request*, *map*)
Накладывает колоночный фильтр

Параметры

- **query** (*django.db.models.query.QuerySet*) – Запрос
- **request** – Данные с клиента включающие фильтры
- **map** – карта связи фильтров в request и полей в объекте: ключ - поле объекта, значение - параметр фильтра, например:

  {'unit__name':'unit_ref_name'}

m3.actions.utils.apply_search_filter(*query*, *filter*, *fields*)
Накладывает фильтр поиска на запрос. Вхождение каждого элемента фильтра ищется в заданных полях. @param query: django.db.models.query.QuerySet @param filter: Строка фильтра @param fields: Список полей модели по которым будет поиск

m3.actions.utils.apply_sort_order(*query*, *columns*, *sort_order*)
Закладывает на запрос порядок сортировки. Сначала если в описании колонок columns есть code, то сортируем по нему, иначе если есть по name. Если задан sort_order, то он главнее всех.

Параметры

- **query** (*django.db.models.query.QuerySet*) – запрос, к которому накладывается порядок сортировки
- **columns** (*dict или tuple*) – список колонок
- **sort_order** (*list*) – если не пустой, то сортирует по нему

Результат запрос с наложенной сортировкой

Тип результата django.db.models.query.QuerySet

m3.actions.utils.bind_object_from_request_to_form(*request*, *obj_factory*, *form*, *request_id_name='id'*, *exclusion=None*)
Функция извлекает объект из запроса по id, создает его экземпляр и биндит к форме

Параметры

- **request** (*django.http.Request*) – Запрос от клиента содержащий id объекта
- **obj_factory** (*callable-object*) – Функция возвращающая объект по его id
- **form** – Класс формы к которому привязывается объект
- **request_id_name** – Имя параметра запроса соответствующего ID объекта

m3.actions.utils.bind_request_form_to_object(*request*, *obj_factory*, *form*, *request_id_name='id'*, *exclusion=None*)
Функция создает объект по id в запросе и заполняет его атрибуты из данных пришедшей формы @param request: Запрос от клиента содержащий id объекта @param obj_factory: Функция возвращающая объект по его id @param form: Класс формы к которому привязывается объект @param request_id_name: Имя параметра запроса соответствующего ID объекта

m3.actions.utils.create_search_filter(*filter_ text*, *fields*)
Фильтрация производится по списку полей fields и введеному пользователем тексту filter_text. работает по следующей схеме

Пример: fields = ['name', 'family'] filter_text = u'Вася Пупкин'

Получится условие WHERE: (name like '%Вася%' or family like '%Вася%') and (name like '%Пупкин%' or family like '%Пупкин%')

если один из параметров пуст, то возвращает пустой Q()

Параметры

- **filter_text** (*str*) – текст, пот которому ведется поиск
- **fields** (*list*) – список полей, по которым нужно искать

Результат фильтр, реализующий поиск слов по полям

Тип результата django.db.models.query_utils.Q

m3.actions.utils.detect_related_fields(*query*, *list_ columns*)
Определяет необходимость выполнения select_related на запросе.

Параметры query (*django.db.models.query.QuerySet*) – запрос, к которому применяется select_related

m3.actions.utils.extract_int(*request*, *key*)
Извлекает целое число из запроса

Параметры key (*str*) – имя параметра

Результат целочисленное значение параметра с заданным именем

Тип результата int

m3.actions.utils.extract_int_list(*request*, *key*)

Извлекает список целых чисел из запроса ..note:: разделителем в строком значении должна быть запятая

Параметры key (*unicode*) – имя параметра

Результат список целочисленных значений из параметра с заданным именем

Тип результата list

m3.actions.utils.fetch_search_tree(*model_ or_ query*, *filter*, *branch_ id=None*, *parent_field_ name='parent'*)
По заданному фильтру filter и модели model формирует развернутое дерево с результатами поиска. Если filter пустой, то получается полностью развернутое дерево.

m3.actions.utils.safe_delete_record(*model*, *id=None*)
Безопасное удаление записи в базе. В отличие от джанговского ORM не удаляет каскадно. Возвращает True в случае успеха, иначе False @deprecated нужно использовать BaseModel.safe_delete()

или m3.db.safe_delete(obj)

4.7 Экшены для работы в асинхронном режиме

class m3.actions.async.AsyncAction
Экшен обработки запросов с клиента. В данном варианте инстанс класса воркера общий для

всех http запросов экземпляра приложения, поэтому если один пользователь начал фоновую операцию, все пользователи этого сервера будут видеть такой же прогресс. Можно с помощью словаря сессий, где бы хранились инстансы воркеров, реализовать для каждого пользователя свою операцию, или же с помощью мутексов глобальную блокировку операции на уровне приложения. При наследовании должен быть определен атрибут worker_cls - класс наследующий/имплементирующий IBackgroundWorker

class `m3.actions.async.AsyncOperationResult`(*value=0.0*, *text=''*, *alive=True*)
Результат выполнения асинхронной операции.

class `m3.actions.async.IBackgroundWorker`(*boundary=''*, *context=None*, **args*, ***kwargs*)
Абстрактный класс для исполнения кода в фоновом режиме. Соответсвующие методы должны быть определены разработчиком. Тк класс представляет собой наследник Thread, нужно понимать что фактически при использовании будет создан новый тред, и учитывать это при доступе к ресурсам приложения, кои могут быть модифицированы другими тредами. Методы stop, start и ping должны возвращать экземпляры AsyncOperationResult. Вообще говоря, можно реализовать данный класс подругому, главное чтобы был имплементирован интерфейс (например если есть нужда использовать мутекс на уровне нескольких инстансов приложений)

`check_state()`
Проверяет состояние глобальной блокировки операции. Возвращает кортеж из двух элементов (is_active, status_data), где is_active=True/False - показывает активность установленной блокировки, и status_data - произвольный объект (чаще строка), в котором находится описание состоания операции.

`lock()`
Устанавливает глобальную блокировку по инстансу с использованием механизма мютексов

`lock_result`(*result*)
Блокирует результат выполнения операции

`refresh_state`(*status_data*)
Обновляет состояние блокировки

`request()`
Запрос состояния операции

`result()`
Запрос на получение результата асинхронной операции

`run()`
Метод вызывается после вызоыва метода start() в новом потоке, здесь фактически следует писать собственный код

`start()`
Запускает исполнение кода. Метод не абстрактный, при замещении обязательно вызывать суперкласс

`stop()`
Метод остановки операции

`unlock()`
Освобождает глобальную блокировку состояния

4.8 Интерфейсы

class `m3.actions.interfaces.IMultiSelectablePack`

Интерфейс pack-классов для множественного выбора значений из полей ExtMultiSelectField

`get_display_dict`(*key*, *value_field='id'*, *display_field='name'*)
Получить список словарей, необходимый для представления выбранных значений ExtMultiSelectField Пример результата: [{'id':0,'name':u''},]

`get_multi_select_url`()
Получить адрес для запроса диалога множественного выбора элемента

class `m3.actions.interfaces.ISelectablePack`
Интерфейс pack-классов для выбора значений из полей ExtDictSelectField

`get_autocomplete_url`()
Получить адрес для запроса элементов подходящих введенному в поле тексту

`get_display_text`(*key*, *attr_name=None*)
Получить отображаемое значение записи (или атрибута attr_name) по ключу key

`get_edit_url`()
Получить адрес для запроса диалога редактирования выбранного элемента

`get_select_url`()
Получить адрес для запроса диалога выбора элемента

- *genindex*
- *modindex*
- *search*

m