Ximpia Documentation

Release 0.2

Jorge

December 27, 2013

Contents

CHAPTER 1

What is Ximpia?

1.1 About Ximpia

1.1.1 Overview

Ximpia allows you to model back-end and front-end in an easy way to minimize lines of code for your application.

On the front-end, you parametrize javascript components in HTML5 and define stylesheets to customize look and feel.

On the back-end, you define your services (=use cases) with views, actions and app flow and other services like search, settings, parameters, already defined so you can speed up development.

1.1.2 Views

Views are called from menu items, search or other action components like buttons, links, etc...

They query your database and display information to your users, so framework uses slave databases for them. For example, they would be list of customers, search customers and customer detail.

Views must all have a form which is defined in the decorator. Forms hold fields as well as window success messages and error messages.

In case you need to link views in a flow, you would set views into the workflow. You don't need to write code for this, all flow logic is kept parametrized into the Workflow. You may write variables into the workflow and then define flows for views and actions depending on variable data. Your services may write into workflow as well. It is easy to write wizards, use cases that link to other use cases in a business operational flow.

```
@view(forms.HomeForm)
def viewHome(self):
        db_setting = self._instances('ximpia.xpsite.data.SettingDAO')[0]
        # your code...
```

We have HomeForm with messages and fields for home view.

You would need to register the view, template, menu items and search for each view. In case you don't map views with menu, you can skip menu registering:

```
self._reg.registerView(__name__, serviceName='Users', viewName='login', slug='login',
                                                    className=SiteService, method='view_login')
```

```
self._reg.registerTemplate(__name__, viewName='login', name='passwordReminder', winType='popup',
                                        alias='password_reminder')
self._reg.registerSearch(__name__, text='Login', viewName='login')
```

1.1.3 Forms

Ximpia forms are a bit different from django forms since they keep database fields inyected into fields.

```
class LoginForm(XBaseForm):
        _XP_FORM_ID = 'login'
        _dbUser = User()
        username = UserField(_dbUser, 'username', label='XimpiaId', required=False,
                jsRequired=True, initial='')
        password = PasswordField(_dbUser, 'password', minLength=6, required=False,
                jsRequired=True, initial='')
        socialId = HiddenField()
        socialToken = HiddenField()
        authSource = HiddenField(initial=K.PASSWORD)
        choices = HiddenField(initial=_jsf.encodeDict({'authSources': Choices.SOCIAL_NETS}))
        errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['ERR_wrong_password']]))
        okMessages = HiddenField(initial=_jsf.buildMsgArray([_m, []]))
```

We need `_XP_FORM_ID` to have an unique id used in front-end. Your forms in a service should have an unique id. When we build form data for front-end, we use field attributes from model like maxlength, labels and helptext. You can customize these attributes in the form class as well.

1.1.4 Actions

Visual components associated with actions like buttons and links will call your actions. They may be called from search and menu items as well.

Action operations may be mapped to your services. Each action would have an implementation associated with it in a method.

```
@validation()
def _authen_user(self):
        if self._f()['authSource'] == K.FACEBOOK and self._f()['socialId'] != '':
                self._ctx.user = self._authenticate_user_soc_net(self._f()['socialId'],
                        self._f()['socialToken'], self._f()['authSource'], 'facebook',
                        _m.ERR_wrong_password)
        else:
                self._ctx.user = self._authenticate_user(self._f()['username'],
                        self._f()['password'], 'password', _m.ERR_wrong_password)

@action(forms.LoginForm)
def login(self):
        """
        Performs the login action. Puts workflow parameter username, write context variables
        userChannel and session.
        """
        self._authen_user()
        self._login()
        user_channel_name = self._get_user_channel_name()
        self._dbUserChannel = UserChannelDAO(self._ctx_min)
        self._ctx.userChannel = self._dbUserChannel.get(user=self._ctx.user,
```

```
                name=user_channel_name)
        self._ctx.session['userChannel'] = self._ctx.userChannel
```

You need to map form associated with the action using `action` decorator. Form is validated prior to processing action in decorator logic.

You can implement validation operations that need to be checked in order to execute your actions. You call them inside your action method (like `self._authen_user()`). You can think of this as service-level validaations or business validations.

You would register them like:

```
self._reg.registerAction(__name__, serviceName='Users', actionName='login', slug='login',
        className=SiteService, method='login')
```

1.1.5 Templates and Visual Components

Ximpia templates are plain HTML5 files. You will find them at:

```
myproject/myapp/templates
```

You will find your app directory with `window` and `popup` directories.

You will also find blank templates at your project path, built by ximpia `ximpia-app` script. You would copy those blank templates and rename them in order to start with your own templates.

Here goes an example for change password popup:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Ximpia - Change Password</title>
</head>
<body>
<div id="id_popup"
                data-xp="{title: 'Change Password'}" ></div>
<!-- Content -->
<section id="id_content" class="sectionContent">
<div id="id_changePassword">
<form id="form_userChangePassword" action="" method="post" data-xp="{}">
<!-- ximpiaId -->
<div id="id_username_comp"
                data-xp-type="field"
                data-xp="{tabindex: '1', label: 'XimpiaId', 'readonly': 'readonly'}" > </div>
<!-- password -->
<div id="id_password_comp" data-xp-type="field"  style="margin-top: 10px"
                data-xp="{type: 'password', info: true}" ></div>
<!-- newPassword -->
<div id="id_newPassword_comp" data-xp-type="field" style="margin-top: 10px"
                data-xp="{type: 'password', info: true, class: 'passwordStrength'}" ></div>
<!-- newPasswordConfirm -->
<div id="id_newPasswordConfirm_comp" data-xp-type="field" style="margin-top: 10px"
                data-xp="{type: 'password', info: true}" ></div>
</form>
</div>
<br/>
</section>
<!-- Content -->
```

```
<!-- Page Button Bar -->
<section id="id_sectionButton" class="sectionButton">
<div id="id_popupButton" class="btBar">
<div id="id_doChangePassword_comp" data-xp-type="button"
                                data-xp="{      form: 'form_userChangePassword',
                                                align: 'right',
                                                text: 'Save',
                                                type: 'iconPopup',
                                                mode: 'actionMsg',
                                                action: 'changePassword',
                                                clickStatus: 'disable',
                                                icon: 'save'}" ></div>
</div>
</section>
<!-- Page Button Bar -->
</body>
</html>
```

`div` elements with `_comp` ending in `id` hold the visual components. These visual components will be parsed by our js rendering engine, build html5 and mix server data with visual data.

Visuals for your application are built using what we call visual components. They are `jQuery` plugins that mix server data with parametrized data in HTML5 templates.

We provide a set of visual components tailored for most needs and you only need to parametrize them in HTML5 templates. Most of the time, you will not need to develop js code, simply configure the components with HTML5 `data-` attributes.

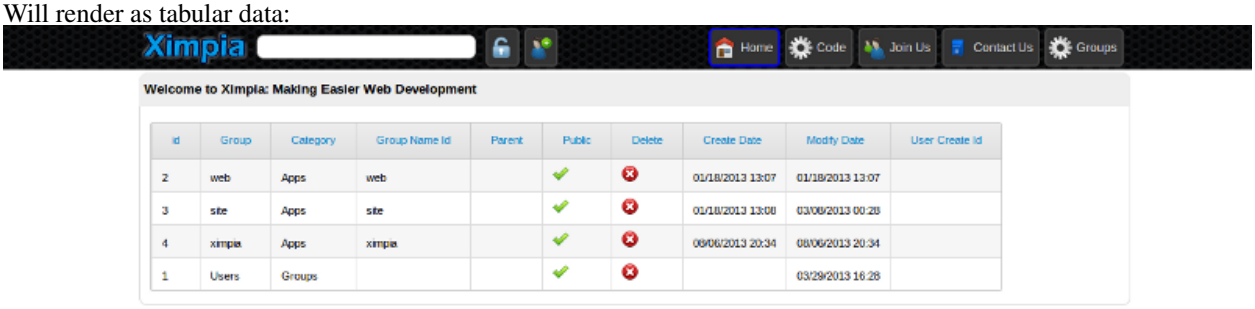
You can link your visual components and apply style themes.

In case this is not enough for you, you can write your own components.

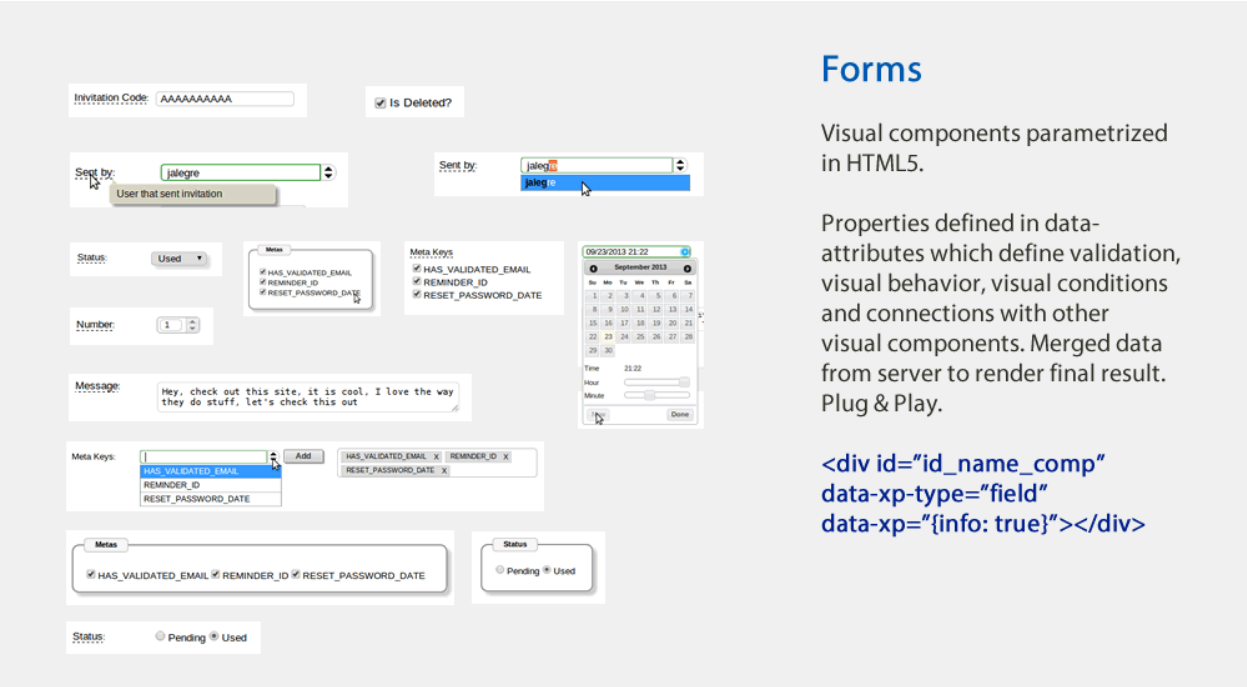
Example for list component:

```
<div id="id_groups_comp" data-xp-type="list.data" style="margin-left: 10px"
        data-xp="{      app: 'ximpia.xpsite',
                dbClass: 'GroupDAO',
                disablePaging: true,
                fields: [       'id',
                                'group__name',
                                'category__name',
                                'groupNameId',
                                'parent',
                                'isPublic',
                                'isDeleted',
                                'dateCreate',
                                'dateModify',
                                'userCreateId']
        }" > </div>
```

Will render as tabular data:

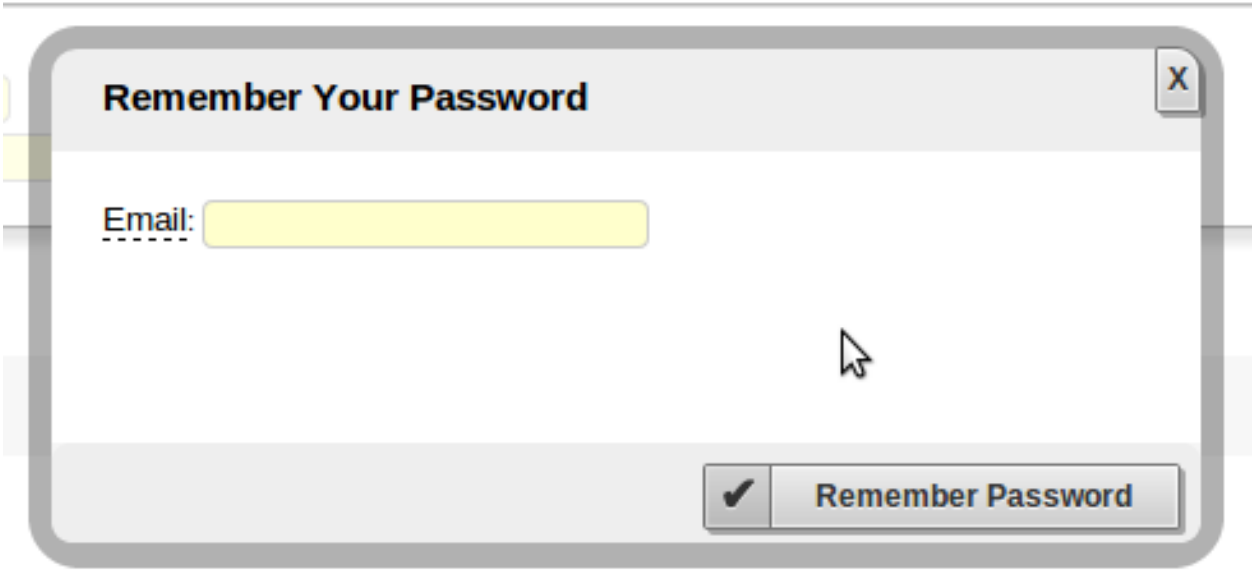


You have many components to choose from:



1.2 Example

Here goes an example for remember password popup.



1.2.1 Model

We use change password operations from default django User.

1.2.2 Visual Components

Define components in template `myproject/myapp/templates/xpsite/popup/password_reminder.html`:

```
<div id="id_password_reminder">
<form id="form_password_reminder" action="" method="post" data-xp="{}">
<div id="id_email_comp" data-xp-type="field" data-xp="{info: true}
    data-xp-val="email"" ></div>
</form>
</div>
```

Email is field type with email validation. You include attributes in `data-` html5 attrs. Info true allows the tool tip when mouse overs.

And this for the button:

```
<section id="id_sectionButton" class="sectionButton">
<div id="id_popupButton" class="btBar">
<div id="id_doPasswordRemind_comp" data-xp-type="button"
        data-xp="{  form: 'form_password_reminder',
                    align: 'right',
                    text: 'Remember Password',
                    type: 'iconPopup',
                    mode: 'actionMsg',
                    action: 'request_reminder',
                    clickStatus: 'disable',
                    icon: 'save'}" ></div>
</div>
</section>
```

1.2.3 Form

```
class PasswordReminderForm(XBaseForm):
    _XP_FORM_ID = 'password_reminder'
    _dbUser = User()
    email = EmailField(_dbUser, 'email', label='Email',
        helpText= _('Email address you signed up with'))
    errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m,
        ['ERR_wrong_password','ERR_email_does_not_exist']]))
    okMessages = HiddenField(initial=_jsf.buildMsgArray([_m,
        ['OK_PASSWORD_REMINDER']]))
```

When we click change password button, `OK_PASSWORD_REMINDER` message would show. This message is included in messages.py file.

1.2.4 View

```
@view(PasswordReminderForm)
def view_password_reminder(self):
    """Show password reminder view"""
    pass
```

1.2.5 Action

Request password reminder will validate that email exists, generate reminderId, send email and return OK or error message.

```
@action(forms.PasswordReminderForm)
def request_reminder(self):
    """Checks that email exists, then send email to user with reset link
    """
    logger.debug('requestReminder...')
    self._dbUser, self._dbSetting, self._dbUserMeta,
        self._dbMetaKey = self._instances(UserDAO, SettingDAO, UserMetaDAO,
            MetaKeyDAO)
    self._validate_email_exist()
    # Update User
    user = self._dbUser.get(email = self._f()['email'])
    days = self._get_setting(K.SET_REMINDER_DAYS).value
    new_date = date.today() + timedelta(days=int(days))
    # Write reminderId and resetPasswordDate
    reminder_id = str(random.randint(1, 999999))
    metas = self._dbMetaKey.metas([K.META_REMINDER_ID, K.META_RESET_PASSWORD_DATE])
    self._dbUserMeta.save_meta(user, metas, {
                                    K.META_REMINDER_ID: reminder_id,
                                    K.META_RESET_PASSWORD_DATE: str(new_date)})
    # Send email with link to reset password. Link has time validation
    xml_message = self._dbSetting.get(name__name='Msg/Site/Login/PasswordReminder/_en').value
    EmailService.send(xml_message, {    'home': settings.XIMPIA_HOME,
                                    'appSlug': K.Slugs.SITE,
                                    'viewSlug': K.Slugs.REMINDER_NEW_PASSWORD,
                                    'firstName': user.first_name,
                                    'userAccount': user.username,
                                    'reminderId': reminder_id}, settings.XIMPIA_WEBMASTER_EMAIL,
                                        [self._f()['email']])
    logger.debug('requestReminder :: sent Email')
    self._set_ok_msg('OK_PASSWORD_REMINDER')
```

1.2.6 Registering

We register components so they work instantly with menu, command search, etc...

```
self._reg.registerView(__name__, serviceName=Services.USERS,
    viewName=Views.REMINDER_NEW_PASSWORD, slug=Slugs.REMINDER_NEW_PASSWORD,
    className=SiteService, method='view_reminder_new_password')
self._reg.registerTemplate(__name__, viewName=Views.REMINDER_NEW_PASSWORD,
    name=Tmpls.REMINDER_NEW_PASSWORD)
self._reg.registerAction(__name__, serviceName=Services.USERS,
    actionName=Actions.REQUEST_REMINDER, slug=Slugs.REQUEST_REMINDER,
    className=SiteService, method='request_reminder')
```

We trigger view with a link, so we don't need menu or search components.

CHAPTER 2

Installation & Setup

2.1 Quick Start

2.1.1 Virtual Environment

Create and activate virtual environment:

```
mkdir envs
virtualenv envs/ximpia-env
source envs/ximpia-env/bin/activate
```

2.1.2 Installation

Using `pip`:

```
pip install ximpia
```

This will install ximpia and required packages:

- Grappelli
- Filebrowser
- South

Go to https://github.com/Ximpia/ximpia/ if you need to download a package or clone the repo.

2.1.3 Upgrading

Using `pip`:

```
pip --upgrade install ximpia
```

Migrate Ximpia apps:

```
python manage.py migrate ximpia.xpcore ximpia.xpsite
```

Update site components:

```
python manage.py xpcomponents ximpia.xpsite
```

2.1.4 Setup Application

To start your application, type:

```
ximpia-app myproject.myapp
```

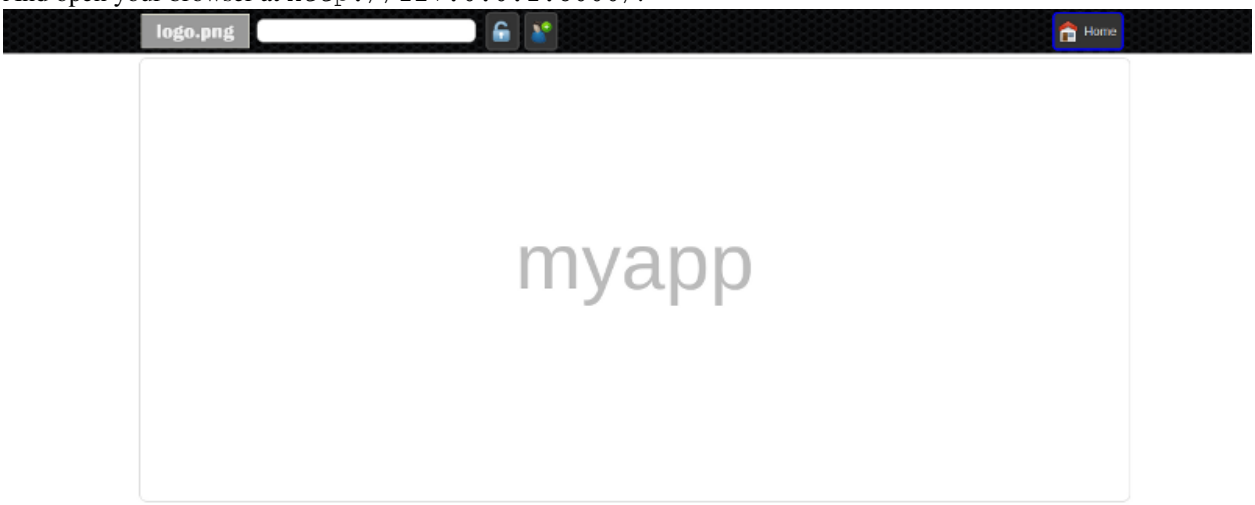
It will create folders and files needed for your application in ximpia. It will prompt for basic information like database connection user, admin name and password and locale.

Creates and registers your application home view.

Then you only need to go to directory for your project:

```
./manage.py runserver
```

And open your browser at `http://127.0.0.1:8000/`:



Customize your logo at `myproject/myproject/myapp/static/images/logo.png`.

You are set to start coding!

CHAPTER 3

Server Side

You start by defining your views and actions in your services using forms as way to map database fields to visual objects:

```
class MyForm(XBaseForm):
    _XP_FORM_ID = 'customer_detail'
    [... your fields with db instances inyected ...]
    errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['ERR_my_error_message']]))
    okMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['success']]))

class MyService(CommonService):

    @view(forms.MyForm)
    def view_customer_detail(self):
        # Logic to get customer detail...
```

Your data layer with common operations (can extend data operations):

```
class MyDAO(CommonDAO):
    model = MyModel
```

And you register components to allow search, menu, workflow and other components consistency:

```
self._reg.registerView(__name__, serviceName='MyService', viewName='customer_detail', slug='customer-
                       className=SiteService, method='view_customer_detail')
self._reg.registerTemplate(__name__, viewName='customer_detail', name='customer_detail')
```

3.1 Services

Ximpia services have these parts:

- *Forms*: Allows to map view fields to database models.
- *Services*: Implement views and action logic.
- *Choices*: List choices for data models and forms.
- *Messages*: Messages used for action results and error messages, displayed to users.

- *Context*: Request, response and service data is kept at service context. Context data is contained in service class with `_ctx` attribute, having forms and additional service data. Minimized version is propagated to business and data layers.

3.1.1 Forms

The way for your service layer to communicate with front-end is forms.

Every view has a form atatched, having fields for messages and visual components fields, weather it is a `list view``or ``detail view` (CRUD).

Form data is produced in `JSON` format and consumed by front-end template parsers.

Ximpia forms are similar to django forms, except that data instances are mapped inside form fields:

```
class CustomerForm(XBaseForm):
    _XP_FORM_ID = 'customer'
    _db_user = User()
    username = UserField(_db_user, 'username', label='XimpiaId', required=False,
        jsRequired=True, initial='')
    errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['ERR_wrong_username']]))
    okMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['success']]))
```

`User` would be the model class for django user.

The first parameter ro ximpia fields is the form data instance. The second one is the data model field in string format. In this case: `_db_user` and `username`.

You map forms to services with service decorators.

XBaseForm

_validate_same_fields(tuple_list)

Attributes

- `tuple_list` (list) : List of tuples. Each tuple with have a couple of fields to match.

_validate_captcha()

Would validate the captcha field with `recaptcha` : `recaptcha_channlenge_field`.

You would only need to call this method in your form views to provide validation for form data:

```
def clean(self):
    """Clean form"""
    self._validate_captcha()
    self._xp_clean()
    return self.cleaned_data
```

_get_field_value(field_name)

Get field value. This method is used in your `clean` method to provide additional validations for your form.

Attributes

- `field_name` (str) : Field name

Returns

Form field value.

_xp_clean()

Clean ximpia form, cheching validations. Used when extending `clean` method.

cleaned_data

Form cleaned data to be compatible with django forms. Used when extending `clean` method.

get_param_dict(param_list:list)

Get dictionary of parameter values for list of parameters.

Attributes

- `param_list` (list) : List of parameters

Returns

Dictionary with parameter values

When we request ['param1', 'param2']

{'param1': 'myValue', 'param2': 'myValue'}

get_form_id()

Get the form id.

Returns

Form Id

has_param(name)

Checks if form has param name.

Attributes

- `name` (str) : Param name

Returns

True/False

put_param_list(args)

Put list of params into form. They will be in params form hidden field:

```
self._f().put_param_list(mode='OK', type='new')
```

These form parameters are attached to your form. You can attach as many as you want from your services to provide visual behavior in your visual components or templates.

save()

Saves the form. It will save all model fields and related tables to the form.

clean()

You would extend this method in your form instances in order to provide cross validations in your form:

```
def clean(self):
    """Clean form"""
    self._validate_same_fields([('newPassword','newPasswordConfirm')])
    self._xp_clean()
    return self.cleaned_data
```

Would provide checking same field values for `newPassword` and `newPasswordConfirm`

You have no need to implement this method in case you don't need extra validations for your form. In that case, all fields would be validated relative to their types and data model associations.

3.1.2 Services

Hold your logic for APIs and use case logic: views, actions and business or service operation validation.

You may map views, actions and action validators inside same service or you may separate action logic in additional services, having view-only services. You may map views and action artifacts any way you want.

Views

Views are rendered to user browser using HTML5 templates for data detail (read operations) and lists (db queries).

This view queries customer data and displays results:

```
from data import CustomerDAO

class SiteService(CommonService):

    @view(forms.CustomerForm)
    def view_customer(self, pk):
        db_customer = self._instances(CustomerDAO)[0]
        self._set_main_form(forms.CustomerForm(instances={
            'db_customer': db_customer.get(pk=pk)
            }))
```

You set the form instance with `_set_main_form`. Keys for data instances must correspond to form fields with data instances. Form `CustomerForm` has db field `db_customer` which is Customer().

The easiest way to get your data instances is through `_instances` common operation, since already inyects context into business and data objects. You can also call data objects directly inyecting context.

This view displays a list of customers:

```
@view(forms.CustomerListForm)
def view_customers(self):
    self._add_list('customers', db_customer.search_fields(['first_name', 'last_name', 'email', 'phone
```

Front-end would have in context field `customers` with the fields selected. Additionaly, `search_fields` takes attribute to search for them to filter out data and has paging support.

Actions

Action visual components like buttons, links and other event driven components would call server-side actions.

You may have different buttons in a view call different actions, mapping directly from your views the service actions (operations).

Service actions have logic validators which in most cases need to be checked before executing action logic. In case validations are not passed, service action is not executed and user sees a warning or error message you specify. You would define service validators for your actions and action logic. You may keep action logic inside your services or place common logic in business layer.

Example that customer has right status:

```
@validation()
def _validate_customer_status(self, customer_id):
    """
    Validates that customer has right status
    """
    self._validate_exists([
            [self._db_customer, {'customer_id': customer_id, 'status': K.PENDING},
                    'status', _m.ERR_customer_wrong_status]
                            ])
```

Validate methods `_validate_exists` and `validate_not_exists` allows you to provide business validation for your actions.

Each element from list of validations has:

- `db instance`: data instance to check for fields existance or not existance
- `field dictionary`: fields to check: key: value in dictionary
- `view field`: View field to highlight in case validation does not pass
- `error message`: Error message to display in view

`@validation` decorator checks that form validates.

And you implement calling validation in your action:

```
@action(forms.CustomerForm)
def activate_customer(self):
    self._validate_customer_status(self._f()['customer_id'])
    [... Logic to stuff to do when customer status is OK]
```

Form Values

You get form cleaned values by calling common service operation `_f`:

```
self._f()['my_field']
```

You may also access like:

```
self._ctx.form['my_field']
```

action decorator checks that form related to action is validated. In case not validated, returns error message to front-end.

In your service logic you don't need to validate form and implement validations, you just fetch form field value using the operation `_f()` which returns a dictionary of field values.

Workflow Actions

Action that will redirect users to other views are managed by the *Workflow*. Worflow allows to define app navigation in a separate place from your code, therefore you don't need to implement flow in your code but in the flow definitions when registering flow components (manage.py xpcomponents yourapp).

You register views and actions related to flows in `components.py` file and you register components calling the xpcore management command `xpcomponents`.

You would place decorator `@workflow_action` when defining these actions, like:

```
@workflow_action(DefaultForm)
def logout(self):
    """Logout user
    """
    self._logout()
```

This decorator will check workflow variables to resolve which view to navigate to.

You may write flow variables from your actions like this:

```
self._put_flow_params(status='OK', mode='new')
```

User flow data is persistent. So when user returns to flow, system knows about last actions and may redirect to right location (you can configure flow behavior with properties). Service action would write `status='OK'` and `mode='new'`. You may define your flows so that different views gets displayed with statuses and modes. You don't need to change code when those requirements change, just update action logic and flow components.

Decorators

- `@view` (form) : View decorator that must send main form to use for view.
- `@action` (form) : Action decorator that must send form to use.
- `@workflow_action` (form) : Workflow action and form to use

CommonService

These are the most common service operations from `CommonService` :

_put_flow_params(args)

Writes flow parameters, like:

```
self._put_flow_params(name=value, ...)
```

_put_form_value(field_name, field_value, form_id=None)

Attributes

- `field_name` (str) : Field name
- `field_value` (str) : Field value
- `form_id` (str) : For multiple forms, set which form field is related to

Writes form field values. Useful when you have form fields not related to your models and need to set value from the service layer.

_f()

You also get form from context:

```
self._ctx.form['my_field']
```

Returns form_values <dict> having format key:value

_ctx

Service context

_validate_exists(db_data_list)

Validates that list of fields for each data entity check. You can include list of entities with list of fields to check. In case rule does not check, front-end highlights field with error message. **All validation data must check**. You keep these in your `_validate_*` method for logic validations using `@validation()` decorator.

```
self._validate_exists([
        [self._dbInvitation, {'invitationCode': invitation_code, 'status': K.PENDING},
                'invitationCode', _m.ERR_invitation_not_valid]
                        ])
```

Attributes

- **`db_data_list` (list)** [Validation data instance list:]
  - `db_instance` (data)
  - `fields` (dict)
  - `view field` (str)
  - `error message` (str)

_validate_not_exists(db_data_list)

Validates that list of fields for each data entity does not check (NOT). You can include list of entities with list of fields to check. In case rule does not check, front-end highlights field with error message. **All validation data must check**. You keep these in your `_validate_*` method for logic validations using `@validation()` decorator.

```
self._validate_not_exists([
            [self._dbUser, {'username': self._f()['username']}, 'username', _m.ERR_ximpia_id],
            [self._dbUser, {'email': self._f()['email']}, 'email', _m.ERR_email]
            ])
```

Attributes

- **`db_data_list` (list)** [Validation data instance list:]
  - `db_instance` (data)
  - `fields` (dict)
  - `view field` (str)
  - `error message` (str)

_get_setting(setting:str)

Get setting. Returns the setting model instance. In case you want value, you would:

```
my_setting_value = self._get_setting('my_setting').value
```

In case you need to check if setting is True/False:

```
if self._get_setting('has_feature').is_checked()
```

Attributes

- `setting` (str) : Setting name

Returns

setting (Setting model instance)

_add_attr(name, value)

Adds attribute to front-end context. You may add any key/value to the context used by front-end. This is useful for writing your own visual components that need additional server-side data, or adding extra data for your views.

Attributes

- `name` (str) : Name
- `value` (str) : Value

Example:

```
self._add_attr('isSocialLogged', False)
```

Used by xpsite login. This attribute, `isSocialLogged` is used by conditions in login view.

_set_main_form(form_instance)

When dealing with multiple forms inside views, allows you to set which one is used for validations, no matter which one you have in decorator.

Attributes

- `form_instance` (XBaseForm) : Form instance

_add_form(form_instance)

We add additional form to view. This form can be mapped into popups.

Attributes

- `form_instance` (XBaseForm) : Form instance

_show_view(view_name, view_attrs={})

Displays view with a set of parameters. In case you need to have advanced workflows, you may redirect flows to views from your service actions.

Attributes

- `view_name` (str) : View name
- `view_attrs` (dict) : View attributes

_set_cookie(key, value)

Sets cookie.

Attributes

- `key` (str) : Key
- `value` (str) : Value

_set_ok_msg(idOK)

Sets which `OK` message will be shown to user. Pretty useful when different messages can be shown depending on conditions. You set which one to show in service logic and users will see that particular message.

Attributes

- `idOK` (str) : Ok message id from messages.py file

_set_form(form_instance)

Set which form instance is used in service context.

Attributes

- `form_instance` (XBaseForm) : Form instance

_instances(args)

Instances data and business classes inyecting context.

args can be list of strings or classes.

When these DAO's are in same app:

```
db_user, db_customer = self._instances('data.UserDAO', 'data.CustomerDAO')
```

When in another app:

```
db_user, db_customer = self._instances('ximpia.xpcore.data.UserDAO',
    'ximpia.xpcore.data.CustomerDAO')
```

For these cases, we import class, create instance and inyect minimized context

```
from my_module import UserDAO, CustomerDAO
db_user, db_customer = self._instances(UserDAO, CustomerDAO)
```

We just create instance from classes without importing.

You may do same with business classes as well as data classes.

3.1.3 Choices

Parametric data which is not meant to change much is contained in `choices.py` for your app:

You refer to choices in form fields and django data model fields.

```
class Choices(object):
# SUBSCRIPTION
SUBSCRIPTION_TRIAL = 'trial'
SUBSCRIPTION_VALID = 'valid'
SUBSCRIPTION_NONE = 'None'
SUBSCRIPTION = (
        (SUBSCRIPTION_TRIAL, _('30-Day Free Trial')),
        (SUBSCRIPTION_VALID, _('Valid')),
        (SUBSCRIPTION_NONE, _('None')),
        )
```

For data lists that are not meant to change often, you may define your lists at `choices.py` inside your app. These choices can be referenced in your models or forms.

You may refer default values in models and forms:

```
from choices import Choices as _Ch

default=_Ch.SUBSCRIPTION_TRIAL
```

3.1.4 Messages

You keep error and `OK` messages in `messages.py` file in your app.

```
# Messages
OK_USER_SIGNUP = _('Your signup has been received, check your email')
OK_SOCIAL_SIGNUP = _('Thanks! Signup complete. You can now login')
OK_PASSWORD_REMINDER = _('OK!. We sent you an email to reset password')
```

```
OK_PASSWORD_CHANGE = _('OK! Password changed')
ERR_change_password = _('Invalid data to change password')
```

You would reference message ids in form messages fields like:

```
import messages as _m
from ximpia.util.js import Form as _jsf

errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['ERR_change_password']]))
okMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['OK_PASSWORD_CHANGE']]))
```

3.1.5 Context

Ximpia context keeps important app data like user profile, service request, service data which is shared by all layer in our applications.

You may extend context to include additional fields and data for your applications.

Attributes

- `app`:String : Application name
- `user`:User : User
- `lang`:String : Language
- `session`:String : Django session object
- `cookies`:object : Django cookies
- `meta`:object : Django META object
- `post`:object . Django POST request
- `request`:object . Django request object
- `get`:object : Django get result
- `userChannel`:String : User channel name
- `auth`:Dict : User has logged in?
- `form`:object : Main form for view
- `forms`:Dict : Forms container for view
- `captcha`:String : Captcha text
- `ctx`:object : Context
- `jsData`:JsResultDict : json data response object, JsResultDict()
- `viewNameSource`:String : For workflows, source view name. In case we have no workflow, this value will be the requested view
- `viewNameTarget`:String : For workflows, target view name.
- `action`:String : Action name
- `isView`:Boolean : View is requested
- `isAction`:Boolean : Action is requested
- `flowCode`:String : Flow code
- `flowData`:String : Flow data

- `isFlow`:Boolean : When True, view is inside workflow.
- `set_cookies`:List
- `device`:String : Device
- `country`:String : Country code
- `winType`:String : Type of windows: window, popup
- `tmpl`:String : Template container
- `wfUserId`:String : Workflow user id
- `isLogin`:Boolean : Weather user has logged in
- `container`:Dict : Container with key->value in dict format
- `doneResult`:Boolean : Used by decorators to define that result has been built.
- `isServerTmpl`:Boolean : Defines if requesting JSON or web response. In case we have an AJAX request, we will have this to False. In case we request an url this value will be True. ServiceDecorator will build different response based on this.
- `dbName` : Resolved connection from data layer. Assigned for first operation, either action or view.
- `path` : Path for actions or vies, like /apps/appSlug/viewSlug or /apps/appSlug/do/actionSlug. Filled by decorators
- `application` : Application model instance

You would have minimized context in business and data layers without forms and other service data.

You would access with `_ctx` attributes in services, data or businesses classes:

```
app = self._ctx.app
```

3.2 Business

Business layer will keep your app logic. Services should communicate with business layer by models or entities, and business layer connect to data layer for data needs.

Business classes have parent `CommonBusiness` class, having attribute `_ctx` and `_instances` common method.

By having context, you may access at your business layer, user, workflow data, session data and other common context data needed in driving logic for your app.

3.2.1 CommonBusiness

_ctx

Attribute which holds the minimized context for the application.

_instances(args)

Method to collect data instances with context inyected:

```
from data import UserDAO, CustomerDAO

db_user, db_customer = self._instances(UserDAO, CustomerDAO)
```

3.2.2 Implementation

```
from ximpia.xpcore.business import CommonBusiness

class MyBusiness(CommonBusiness):

    def alocate_customer(self, customer):
        # logic for allocating a customer
        user = self._ctx.user
        session = self._ctx.session
        [ ... code .... ]
```

3.2.3 Coding Practises

You have organize your app having logic at services or move some or whole 'business' logic to the business layer.

When you have a complex application you probably will have many common operations which define the business. Youy may move them to the business layer and services do the job of managing forms and use case requests.

Having business layer also allows you to have different service classes with unique service logic but having same business operations.

You may have a service which is an external API that creates a new customer and you may have an internal one (which does additional things). By having a common business logic of creating a customer, you may call that business operations from both services. When logic for creating a customer changes, you only need to change at the business operation you define.

3.3 Data

Ximpia provides a data layer to connect to your data sources. These data sources can be relational (SQL) data sources or noSQL data sources.

We follow philosophy that service layer do not call django model data access methods directly and use data classes (what could be similar to a django model manager).

You may have your data operations return django models or simple entities like dictionaries with attribute calling (customer.name) when you deal with data sources and noSQL.

Advantages of data layer:

- You keep simple and complex data access methods in the data layer
- Knows which data nodes to call (master or replicas) depending on type of request (view or action)
- You may change your data sources without changing your services / business layers
- You may change data sources (django, sqlalchemy, redis, mongo)

Currenly we offer CommonDAO which deals with django data model access. This class has common data operations.

3.3.1 Master/Replicas

In case you use master/slave data nodes, you don't need to have routers to route to master or slaves. We think a simpler design is good. Depending on view or action request, the data layer hits master or slave nodes. In case no slaves defined, will allways hit default data node.

3.3.2 BaseModel

Keeps attributes about auditory (create_user and update_user) and create and update fields.

All django models would extend this model.

Keeps delete management. When we delete data, we write `isDeleted=True`.

You can delete data for ever with `is_real` attribute in delete operations.

3.3.3 CommonDAO

Your data layer classes will be related to django models, like:

```
from ximpia.xpcore.data import CommonDAO

class UserMetaDAO(CommonDAO):
    model = UserMeta

    def my_data_operation(self, customer_id):
        # you return django queryset, django model, attribute dictionary or plain dictionary
        # good place to have complex queries, Q objects, etc...
```

Implementation:

```
db_customer = self._instances(CustomerDAO)[0]
customers = db_customer.search(id=23)
```

Will return a django queryset, where you can filter (`customers.filter(name='john')`)

In case you need to access model (from services and business) and do operations on model, do:

```
db_customer.model.objects.all()
```

Exception

Will raise `XpMsgException` with error literal message and attribute `origin='data'`:

```
from ximpia.xpcore.models import XpMsgException
try:
    db_customers.get(id=34)
except XpMsgException:
    # treat exception
```

- *check*
- *create*
- *delete*
- *delete_if_exists*
- *delete_by_id*
- *filter_data*
- *get*
- *get_all*
- *get_create*
- *get_map*

- *get_by_id*
- *save*
- *search*
- *search_fields*

check

Checks if exists

Attributes

keyword attributes, like:

```
if db_customer.check(name='john', status='new'):
    # more...
```

Returns

True/False

create

Will create model with attributes

```
customer = db_customer.create(name='john', status='new')
```

Attributes

keyword attributes

Returns

Django model created

delete

Will delete rows that match the keyword attributes

```
db_customer.delete(name='john', status='new')
```

Attributes

- `is_real` (boolean)

keywork attributes

Returns

None

delete_if_exists

Will delete rows that match the keyword attributes in case exists. If not, does not throw exception.

```
db_customer.delete_if_exists(name='john', status='new')
```

Attributes

- `is_real` (boolean)

keywork attributes

Returns

None

delete_by_id

Delete by primary key

```
db_customer.delete_by_id(23)
db_customer.delete_by_id(23, is_real=True)
```

Attributes

- `pk` (long)

Optional Attributes

- `is_real` (boolean)

Returns

None

filter_data

Search model with ordering and paging

```
db_customer.filter_data(status='OK')
db_customer.filter_data(status='OK', xpNumberMatches=100)
```

Attributes

keyword attributes

Optional Attributes

- `xpNumberMatches` (int) : default 100
- `xpPage` (int) : default 1
- `xpOrderBy` (tuple)

keywork attributes

Returns

queryset

get

Get model which match attributes

Attributes

keyword attributes for query

Returns

django model

get_all

Get all results for model

Returns

django queryset

get_create

Get object. In case does not exist, create model object

Attributes

Keyword attributes

Returns

(object, created) <model, boolean>

get_map

Get container (dictionary) of {id: object, ...} for list of ids

Attributes

- `id_list` (list) : List of ids

Returns

Dictionary with ids and objects

get_by_id

Get object by id

Attributes

- `field_id`

Returns

Model object

save

Saves the model

```
customer = CustomerDAO.model(name='john', status='OK')
customer.save()

db_customer = self._instances(CustomerDAO)[0]
customer = db_customer.get_by_id(23)
customer.name='james'
customer.save()
```

Returns

None

search

Search model to get queryset (like filter)

Search model, like:

```
customers = db_customer.search(name='john')
```

Attributes

- `qs_args` (dict) : Keywork attributes like attr=value

Returns

Django queryset

search_fields

Search table with paging, ordering for set of fields. listMap allows mapping from keys to model fields.

** Attributes **

- `fields`:tuple<str>

Optional Attributes

- `page_start`:int [optional] [default:1]
- `page_end`:int [optional]
- `number_results`:int [optional] [default:from settings]
- `order_by`:tuple<str> [optional] [default:[]]

keyword attributes

** Returns **

Returns the query set with values(*fields).

xpList:ValuesQueryset

3.4 Models

- *BaseModel*
- *Action*
- *Application*
- *Condition*
- *CoreParam*
- *MetaKey*
- *Menu*
- *Params*
- *Service*
- *Setting*
- *View*

- *XpTemplate*
- *Workflow*
- *WorkflowView*
- *WorkflowData*

3.4.1 Introduction

Models for `xpcore` and `xpsite` applications

BaseModel

Abstract Base Model with fields for all other models. Ximpia models have this model as parent. This model provides audit information like date creating and updating, as well as user involved in the update or creation.

When you delete rows in Ximpia, `isDeleted``field is set to True. You can force to physical delete by passing ``real=True` to data delete methods (db.delete, db.deleteById and db.deleteIfExists).

When deleting rows with the django admin interface, you cannot delete database records physically, therefore those rows will have isDeleted=True and will not show in the admin pages. You can delete for ever with your application code or directly connecting to your database.

Attributes

- `id` : Primary key
- `dateCreate`:DateTimeField
- `dateModify`:DateTimeField
- `userCreateId`:IntegerField
- `userModifyId`:IntegerField
- `isDeleted`:BooleanField

Relatinships

Action

Actions are mapped to service operations. Actions can be triggered by clicking on a button, a link, a menu icon or any other visual component that triggers actions.

Here we map action names, implementations, slugs and action properties. Implementations are built by component registering.

Attributes

- `id`:AutoField : Primary key
- `name`:CharField(30)
- `implementation`:CharField(100)
- `slug`:SlugField(50)
- `hasAuth`:BooleanField

- image:FileBrowserField

Relationships

- application -> Application
- service -> Service
- accessGroups <-> xpsite.Group through ActionAccessGroup

Application

Applications. For most sites, they will have single application for N services which relate to use cases for views and actions. In case your application is big or have admin backdoors, your site with have more than one application.

This table holds applications at your site. slug corresponds to the host name that related to the application, like 'slug.domain.com'. You can also access applications by /apps/slug in case application hosts is disabled.

Applications can be grouped together using the parent field.

Applications can have subscription business model or be free. In case subscription required, isSubscription would be True

Applications can be private and accessible only to a group of users. isPrivate would be True in this case. Applications can start private (like a private beta) and then make them public. When making public applications, you can publish at Ximpia directory.

Applications have meta information through model ApplicationMeta. You can attach meta values for application in this model.

Attributes

- id : Primary key
- name:CharField(15) : Application path, like ximpia.xpsite. Must contain package name and application name. Has format similar to installed apps django setting.
- slug:SlugField(30)
- title:CharField(30)
- isSubscription:BooleanField
- isPrivate:BooleanField
- isAdmin:BooleanField

Relationships

- developer -> User
- developerOrg -> 'xpsite.Group'
- parent -> self
- accessGroup -> 'xpsite.Group'
- users <-> UserChannel through ApplicationAccess and related name 'app_access'
- meta <-> Meta through ApplicationMeta and related name 'app_meta'

Condition

Conditions

** Attributes **

- `id`:AutoField : Primary key
- `name`:CharField(30) : Condition name
- `condition`:CharField(255)

** Relationships **

CoreParam

Parameters

You would place tables (key->value) in choices module inside your application for tables that will not change often. When you have data that will change frequently, you can user this model to record parameters, lookup tables (like choices) and any other parametric information you may require.

You can do:

MY_FIRST_PARAM = 67 by inserting name='MY_FIRST_PARAM', value='67', paramType='integer'

Country ||name||value|| ||es||spain|| ||us||United States||

by... mode='COUNTRY', name='es', value='Spain', paramType='string' mode='COUNTRY', name='us', value='United States', paramType='string'

Attributes

- `id` : Primary key
- `mode`:CharField(20) : Parameter mode. This field allows you to group parameter to build lookup tables like the ones found in combo boxes (select boxes) with name->value pairs.
- `name`:CharField(20) : Parameter name
- `value`:CharField(100) : Parameter value
- `paramType`:CharField(10) : Parameter type, as Choices.PARAM_TYPE . Choices are string, integer, date.

Relationships

MetaKey

Model to store the keys allowed for meta values

Attributes

- `id`:AutoField : Primary key
- `name`:CharField(20) : Key META name

Relationships

- `keyType` -> CoreParam : Foreign key to CoreParam having mode='META_TYPE'

Menu

Attributes

- `id`:AutoField : Primary Key
- `name`:CharField(20) : Menu item name
- `titleShort`:CharField(15) : Title short. Text shown in icon. Default menu shows this text right to the icon image.
- `title`:CharField(30) : Title shown in tooptip when mouse goes over.
- `url`:URLField : Url to launch . Used for external urls mapped to menu items.
- `urlTarget`:CharField(10) : target to launch url
- `language`:CharField(2) : Language code, like `es`, `en`, etc...
- `country`:CharField(2) : Country as Choices.COUNTRY
- `device`:CharField(10) : Device. Smartphones, tablets can have their own menu, customized to screen width

Relationships

- `application` -> Application
- `icon` -> CoreParam
- `view` -> View
- `action` -> Action
- `params` <-> Param through MenuParam with related name 'menu_params'

Params

Parameters for WF and Views

Attributes

- `id`:AutoField : Primary Key
- `name`:CharField(15)
- `title`:CharField(30)
- `paramType`:CharField(10) : As Choices.BASIC_TYPES
- `isView`:BooleanField
- `isWorkflow`:BooleanField

Relationships

- `application` -> Application

Service

Attributes

- `id`
- `name`

- implementation

Relationships

- application

Setting

Settings model

Attributes

- value:TextField : Settings value.
- description:CharField(255) : Setting description.
- mustAutoload:BooleanField : Has to load settings on cache?

Relationships

- name -> MetaKey : Foreign key to MetaKey model.

View

View. Pages in ximpia are called views. Views render content obtaine from database or other APIs. They hit the slave databases. In case writing content is needed, could be accomplished by calling queues. Views can show lists, record detalils in forms, reports, static content, etc...

In case no logic is needed by view, simply include pass in the service operation.

Views have name to be used internally in component registering and code and slug which is the name used in urls.

View implementation is the path to the service operation that will produce view JSON data to server the frontend. Implementation is built by registering a view component.

Window types can be 'window', 'popup' and 'panel' (this one coming soon). Windows render full width, popups are modal windows, and panels are tooltip areas inside your content. Popups can be triggered using icons, buttons or any other action. Panels will be triggered by mouse over components, clicking on visual action components.

In case view needs authentication to render, would have hasAuth = True.

Views can be grouped together using the parent field.

Params are entry parameters (dynamic or static) that view will accept. Parameters are inyected to service operations with args variable. The parameter name you include will be called by args['MY_PARAM'] in case your parameter name is 'MY_PARAM'.

Attributes

- id:AutoField : Primary Key
- name:CharField(30)
- implementation:CharField(100)
- winType:CharField(20) : Window type, as Choices.WIN_TYPE_WINDOW
- slug:SlugField(50) : View slug to form url to call view
- hasAuth:BooleanField : Needs view authentication?
- image:FileBrowserField : View image

Relationships

- `parent` -> self
- `application` -> Application
- `service` -> Service
- `category` -> xpsite.Category
- `templates` <-> XpTemplate through ViewTmpl with related name *view_templates*
- `params` <-> Param through ViewParamValue with related nam 'view_params'
- `menus` <-> Menu through ViewMenu with related name 'view_menus'
- `tags` <-> xpsite.Tag through ViewTag
- `accessGroups` <-> xpsite.Group through ViewAccessGroup

XpTemplate

Ximpia Template.

Views can have N templates with language, country and device target features. You can target templates with device and localization. In case you want to provide different templates for user groups, profiles, etc... you would need to create different views and then map those views to access groups. Each of those views would have default templates and templates targetted at pads, smartphones, desktop and localization if required.

Templates can window types:

- Window - Views which render whole available screen area.
- Popup - Modal views that popup when user clicks on actions or menu items.
- Panel (Coming soon) - This window types is embedded within content, as a tooltip when user clicks on action or mouse goes over

Attributes

- `id`:AutoField : Primary Key
- `name`:CharField(50)
- `alias`:CharField(20)
- `language`:CharField(2) : As Choices.LANG
- `country`:CharField(2) : As Choices.COUNTRY
- `winType`:CharField(20) : As Choices.WIN_TYPES
- `device`:CharField(10) : As Choices.DEVICES : Desktop computer, smartphones and tablets

Relationships

- `application` -> Application

Workflow

Ximpia comes with a basic application workflow to provide navigation for your views.

Navigation is provided in window and popup window types.

You "mark" as workflow view any service method with flow code (decorator). Actions are also "marked" as worflow actions with decorators.

When actions are triggered by clicking on a button or similar, action logic is executed, and user displays view based on flow information and data inserted in the flow by actions. You do not have to map navigation inside your service operations.

Plugging in a new view is pretty simple. You code the service view operation, include it in your flow, and view (window or popup) will be displayed when requirements are met

Attributes

- `id`:AutoField : Primary Key
- `code`:CharField(15) : Flow code
- `resetStart`:BooleanField : The flow data will be deleted when user displays first view of flow. The flow will be reset when user visits again any page in the flow.
- `deleteOnEnd`:BooleanField : Flow data is deleted when user gets to final view in the flow.
- `jumpToView`:BooleanField : When user visits first view in the flow, will get redirected to last visited view in the flow. User jumps to last view in the flow.

Relationships

- `application` -> Application

WorkflowView

WorkFlow View. Relationship between flows and your views.

Source view triggers action, logic is executed and target view is displayed to user.

Attributes

- `id`:AutoField : Primary Key
- `order`:IntegerField : View orderi flow. You can place order like 10, 20, 30 for views in our flow. And then later inyect views between those values, like 15, for example.

Relationships

- `flow` -> WorkFlow
- `viewSource` -> View : Source view for flow
- `viewTarget` -> View : Target view for flow
- `action` -> Action : Action mapped to flow. Source view triggers action, logic is executed and target view is rendered and displayed.
- `params` <-> Param through WFParamValue with related name 'flowView_params'

WorkflowData

User Workflow Data

userId is the workflow user id. Flows support authenticated users and anonymous users. When flows start, in case not authenticated, workflow user id is generated. This feature allows having a flow starting at non-authenticated views and ending in authenticated views, as well as non-auth flows.

Attributes

- `id`:AutoField : Primary Key
- `userId`:CharField(40) : Workflow user id
- `data`:TextField : Workflow data encoded in json and base64

Relationships

- `flow` -> Workflow
- `view` -> View

3.5 Workflow

Defines navigation for your application, which views trigger other views and which actions are called that trigger views.

Ximpia would query which view to show when actions are executed. You don't need to code navigation for your application. When your flow changes, you update variables and flow parameters and your flow is changes (For example, to plug in a new view).

We define flow and simple navigation without conditions. You would register flow in `components.py`:

```
self._reg.registerFlow(__name__, flowCode='customer_provision')
self._reg.registerFlowView(__name__, flowCode='customer_provision',
    viewNameSource='create_customer', viewNameTarget='customer_provisioned_profile_a',
    actionName='activate_customer', condition="profile='corporate'")
self._reg.registerFlowView(__name__, flowCode='customer_provision',
    viewNameSource='create_customer', viewNameTarget='customer_provisioned_profile_b',
    actionName='activate_customer', condition="profile='smb'")
```

And then write flow variables in your service layer in action methods with correct decorator:

```
@workflow_action(forms.CustomerForm)
def activate_customer(self, customer_id, activation_code):
    self._put_flow_params(profile='corporate')
```

Workflow would have enough info on which views to navigate to, depending on workflow variable `profile`. You may check for a number of conditions that all must match.

You can also:

- Define flow links with or without events: When having an event (click on button, for example), flow will not continue. But when no event then flow will continue while conditions meet or finds an event.
- Condition-less: You may define flow links without conditions for parameters. In these cases, it is like adding a hard link that will be met no matter what.

3.5.1 Decorators

- `ximpia.xpcore.service.workflow_action` (form) : Set main form to be used for validation.

3.5.2 Extension

In case the provided workflow is not enough for you, you can provide you own flow using provided methods in *CommonService*.

_show_view(view_name, view_attrs={})

Displays view with a set of parameters. In case you need to have advanced workflows, you may redirect flows to views from your service actions.

Attributes

- `view_name` (str) : View name
- `view_attrs` (dict) : View attributes

_get_flow_params(*name_list)

Get flow params by list:

```
params = self._get_flow_params('status', 'profile')
```

_get_wf_user()

Get workflow user. Allows for user id and dealing with advanced flows.

3.5.3 Models

- *Workflow*
- *WorkflowView*
- *WorkflowData*

3.5.4 WorkflowBusiness

build_flow_data_dict

Build the flow data dictionary having the flowData instance

Attributes

- `flow_data`

Returns

flow_data_dict (dict)

gen_user_id

Generate workflow user id.

Returns user_id (long)

get

Attributes

- `flow_code` (str)

Get flow

Returns

workflow:Workflow

get_flow_data_dict

Get flow data dictionary for user and flow code

Attributes

- `wf_user_id`
- `flow_code`

Returns

flow_data_dict (dict)

get_flow_view_by_action

Get flow by action name. It queries the workflow data and returns flow associated with actionName

Attributes

- `action_name`

Returns

flow_view (WorkflowView)

get_param

Get workflow parameter value from context

Attributes

- `name`

Returns

parameter value (str)

get_param_from_ctx

Get flow parameter from context.

Attributes

- `name`

Returns

Parameter value

get_view

Get view from flow

Attributes

- `wf_user_id`
- `flow_code`

Returns

view (View)

get_view_name

Get view name

get_view_params

Get view flow parameters

Attributes

- `flow_code`
- `view_name`

Returns

params (dict) : {name: value, ... }

is_first_view

Is first view in flow?

Attributes

- `flow_code`
- `view_name`

Returns

True/False

is_last_view

Is last view in flow?

Attributes

- `view_name_source`
- `view_name_target`
- `action_name`

Returns

True/False

put_params

Put params in flow.

Attributes

keywork argumens

remove_data

Remove user data from flow.

Attributes

- `wf_user_id`
- `flow_code`

reset_flow

Reset flow

Attributes

- `wf_user_id`
- `flow_code`
- `view_name`

resolve_flow_data_for_user

Resolves flow for user and session key

Attributes

- `wf_user_id`
- `flow_code`

Returns

flow:WorkflowData

resolve_view

Search destiny views with origin viewSource and operation actionName

** Attributes **

- `wf_user_id` : Workflow user id
- `app_name` : App name
- `flow_code` : Flow code
- `view_name_source` : Origin view
- `action_name` : Action name

** Optional Attributes **

- `flow_views` (list<WorkflowView>) : List of flow views to check flow links from. No need to query for flow links.

** Returns ** flowView resolved, which represent flow link data

save

Save flow into data source

Attributes

- `wf_user_id`
- `flow_code`

set_view_name

Set view name

Attributes

- `view_name`

3.6 Fields

3.6.1 Introduction

Ximpia form provide a map with data sources and visual objects in templates. Forms are built from fields and an additional clean method to provide advanced validations.

Fields have:

- model instance
- model field name
- additional attributes like choices, required, min length, etc...

Forms are serialized in **JSON** format and rendered in visual engine. Any rendered object will have a form or a field inside a form, weather or not those objects are rendered as fields that user enter data.

For example, a view to display detail for a customer could have a set of fields in a forms and only show display option for those fields, therefore will never be a box to input text.

Some attributes like `helpText` can be informed in form fields or in visual components in templates. Values entered in templates would overide values found in form fields.

```
class UserSignupInvitationForm(XBaseForm):
    _XP_FORM_ID = 'signup'
    # Instances
    _dbUser = User()
    _dbUserChannel = UserChannel()
    _dbAddress = Address()
    _dbInvitation = Invitation()
    # Fields
    username = UserField(_dbUser, 'username', label='XimpiaId')
    password = PasswordField(_dbUser, 'password', minLength=6, required=False, jsRequired=False,
        helpText = _('Must provide a good or strong password to signup. Allowed characters are lette
```

```
    passwordVerify = PasswordField(_dbUser, 'password', minLength=6, required=False, jsVal=["{equalTo
                                jsRequired=False, label= _('Password Verify'))
    email = EmailField(_dbInvitation, 'email', label='Email')
    firstName = CharField(_dbUser, 'first_name')
    lastName = CharField(_dbUser, 'last_name', required=False)
    city = CharField(_dbAddress, 'city', required=False)
    country = OneListField(_dbAddress, 'country', choicesId='country', required=False, choices=Choice
    invitationCode = CharField(_dbInvitation, 'invitationCode', required=False, jsRequired=True)
    authSource = HiddenField(initial=K.PASSWORD)
    socialId = HiddenField()
    socialToken = HiddenField()
    # Navigation and Message Fields
    params = HiddenField(initial=_jsf.encodeDict({
                                    'profiles': '',
                                    'userGroup': K.SIGNUP_USER_GROUP_ID,
                                    'affiliateId': -1}))
    choices = HiddenField(initial=_jsf.encodeDict( {'country': Choices.COUNTRY } ) )
    errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m,
                                        ['ERR_ximpia_id', 'ERR_email', 'ERR_social_id_exists']]))
    okMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['OK_USER_SIGNUP','OK_SOCIAL_SIGNUP']]))

    def clean(self):
        """Clean form: validate same password and captcha when implemented"""
        if self._get_field_value('authSource') == K.PASSWORD:
            self._validate_same_fields([('password','passwordVerify')])
        self._xp_clean()
        return self.cleaned_data
```

- *Field*
- *BooleanField*
- *CharField*
- *DateField*
- *DateTimeField*
- *DecimalField*
- *EmailField*
- *FileBrowseField*
- *FloatField*
- *GenericIPAddressField*
- *HiddenField*
- *IntegerField*
- *ManyListField*
- *OneListField*
- *PasswordField*
- *TimeField*
- *UserField*

Field

Common field form class. Extends django field.

Required Arguments

- `instance`
- `insField`

Optional Arguments

- `required`:bool [None] : Field is required by back-end
- `jsRequired`:bool [None] : Field is required by front-end
- `jsVal`:bool [None] : Javascript validation
- `label`:str [None] : Field label
- `initial`:str [None] : Field initial value
- `helpText`:str [None] : Field tooltip
- `errorMessages`:dict [None] : Error messages in dict format
- `validators`:list [[]] : List of validators

BooleanField

Boolean field. This field can be rendered into any visual component: checkbox, selection box, etc... The most common use is to render into a checkbox.

Example:

isOrdered = BooleanField(_dbUserOrder, 'isOrdered')

where _dbUserOrder is a form class attribute with the model instance, `_dbUserOrder = UserOrder()`

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str

- `jsRequired`:bool
- `label`:str
- `helpText`:str

CharField

Char field.

Example:

firstName = CharField(_dbUser, 'first_name')

where _dbUser is a form class attribute with the model instance, `_dbUser = User()`

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `minLength`:int : Field minimum length
- `maxLength`:int : Field maximum length
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `minLength`:str
- `maxLength`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

DateField

Example:

where _dbModel is a form class attribute with the model instance.

Input Formats

A list of formats used to attempt to convert a string to a valid datetime.date object.

If no input_formats argument is provided, the default input formats are:

‘%Y-%m-%d’, # ‘2006-10-25’ ‘%m/%d/%Y’, # ‘10/25/2006’ ‘%m/%d/%y’, # ‘10/25/06’

Additionally, if you specify USE_L10N=False in your settings, the following will also be included in the default input formats:

‘%b %d %Y’, # ‘Oct 25 2006’ ‘%b %d, %Y’, # ‘Oct 25, 2006’ ‘%d %b %Y’, # ‘25 Oct 2006’ ‘%d %b, %Y’, # ‘25 Oct, 2006’ ‘%B %d %Y’, # ‘October 25 2006’ ‘%B %d, %Y’, # ‘October 25, 2006’ ‘%d %B %Y’, # ‘25 October 2006’ ‘%d %B, %Y’, # ‘25 October, 2006’

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `inputFormats`:list : Input formats
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

DateTimeField

Example:

where _dbModel is a form class attribute with the model instance.

Input Formats

A list of formats used to attempt to convert a string to a valid datetime.date object.

If no input_formats argument is provided, the default input formats are:

‘%Y-%m-%d %H:%M:%S’, # ‘2006-10-25 14:30:59’ ‘%Y-%m-%d %H:%M’, # ‘2006-10-25 14:30’ ‘%Y-%m-%d’, # ‘2006-10-25’ ‘%m/%d/%Y %H:%M:%S’, # ‘10/25/2006 14:30:59’ ‘%m/%d/%Y %H:%M’, # ‘10/25/2006 14:30’ ‘%m/%d/%Y’, # ‘10/25/2006’ ‘%m/%d/%y %H:%M:%S’, # ‘10/25/06 14:30:59’ ‘%m/%d/%y %H:%M’, # ‘10/25/06 14:30’ ‘%m/%d/%y’, # ‘10/25/06’

Required Arguments

- `instance`:object : Model instance

- `insField`:str : Model field

Optional Arguments

- `inputFormats`:list : Input formats
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

DecimalField

Decimal field with support for maxValue, minValue, maxDigits and decimalPlaces

Example:

amount = DecimalField(_dbModel, 'field', maxValue=9800, minValue=100, maxDigits=4, decimalPlaces=2)

where _dbModel is a form class attribute with the model instance.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `maxValue`:decimal.Decimal : Maximum value
- `minValue`:decimal.Decimal : Minimum value
- `maxDigits`:int : Maximum number of digits (before decimal point plus after decimal point)
- `decimalPlaces`:int : Number of decimal places
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- instance:object
- instanceFieldName:str
- required:bool
- initial:str
- jsRequired:bool
- label:str
- helpText:str

EmailField

Email field. Validates email address

Required Arguments

- instance:object : Model instance
- insField:str : Model field

Optional Arguments

- minLength:int : Field minimum length
- maxLength:int : Field maximum length
- required:bool : Required field by back-end form valdiation
- initial:str : Initial value
- jsRequired:str : Required field by javascript validation
- label:str : Field label
- helpText:str : Field tooptip

Attributes

- instance:object
- instanceFieldName:str
- minLength:str
- maxLength:str
- required:bool
- initial:str
- jsRequired:bool
- label:str
- helpText:str

FileBrowseField

File browser form field.

We keep additional attributes for visual component into `data-xp` html attribute:

- `site` : Site to search for files.
- `directory` : Directory to search for files.
- `extensions` : File extensions to search for files.
- `fieldFormats` : File formats to search for.

These attributes are used to search for files when search icon in file browser field is clicked.

In case these attributes are None, files will be searched in default media home with all extensions and file formats.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `site`:str : Site that keeps media files
- `directory`:str : Directory that keeps media files
- `extensions`:list : Extensions
- `fieldFormat`:list : Field formats

Attributes

- `instance`:object
- `instanceFieldName`:str
- `minLength`:str
- `maxLength`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str
- `site`:str : Site that keeps media files
- `directory`:str : Directory that keeps media files
- `extensions`:list : Extensions
- `fieldFormat`:list : Field formats

FloatField

Integer field with maxValue and minValue

Example:

number = FloatField(_dbModel, 'field', maxValue=9800, minValue=100)

where _dbModel is a form class attribute with the model instance.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `maxValue`:decimal.Decimal : Maximum value
- `minValue`:decimal.Decimal : Minimum value
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

GenericIPAddressField

Generic IP Address field, IPv4 and IPv6

Example:

isOrdered = IPGenericAddressField(_dbModel, 'ip')

where _dbModel is a form class attribute with the model instance.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `protocol`:str ['both'] : Protocol, possible values: both|ipv4|ipv6
- `unpackIpv4`:bool [False]

- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

HiddenField

Hidden Field, name and value.

Optional Arguments

- `initial`:str : Initial value

Attributes

- `initial`:str : Initial value

IPAddressField

Ip Address field, IPv4, like 255.255.255.0

Example:

isOrdered = IPAddressField(_dbModel, 'ip')

where _dbModel is a form class attribute with the model instance.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- instance:object
- instanceFieldName:str
- required:bool
- initial:str
- jsRequired:bool
- label:str
- helpText:str

IntegerField

Integer field with maxValue and minValue

Example:

number = IntegerField(_dbModel, 'field', maxValue=9800, minValue=100)

where _dbModel is a form class attribute with the model instance.

Required Arguments

- instance:object : Model instance
- insField:str : Model field

Optional Arguments

- maxValue:decimal.Decimal : Maximum value
- minValue:decimal.Decimal : Minimum value
- required:bool : Required field by back-end form valdiation
- initial:str : Initial value
- jsRequired:str : Required field by javascript validation
- label:str : Field label
- helpText:str : Field tooptip

Attributes

- instance:object
- instanceFieldName:str
- required:bool
- initial:str
- jsRequired:bool
- label:str
- helpText:str

ManyListField

Selection with many possible values. Will render: select with multiple attribute, list of items with checkbox, other visual components with multiple values from a list.

In case choices is not null, we attempt to skip many search for values and get values from choices object.

From choices... country = ManyListField(_dbAddress, 'country', choicesId='country', required=False, choices=Choices.COUNTRY)

From many relationship... country = ManyListField(_dbAddress, 'country', choicesId='country', required=False)

From Choices

You need to include arguments `choicesId`, `choices`.

From Many to Many relationship

You need to include arguments: `choicesId`. Optional `limitTo`, `listName``and ``listValue`. In case these optional attributes not defined, will search without filter and name will be FK and value string representation of model instance.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field, like '_myModel.fieldName'
- `choicesId`:str: Choice id to save into id_choices hidden field, like {myChoiceId: [(name1,value1),(name2,value2),...] ... }

Optional Arguments

- `limitTo`:dict : Dictionary with attributes sent to model filter method
- `listValue`:str : Model field to be used for value in (name, value) pairs. By default, string notation of model used.
- `values`:tuple
- `orderBy`:tuple : Order by tuples, like ('field', '-field2'). field ascending and field2 descending.
- `choices`:list

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str
- `choicesId`:str
- `limitTo`:dict
- `orderBy`:tuple
- `values`:tuple

- `listValue`:str

Visual Component Attributes

Attributes inside attrs field attribute:

- `choicesId`
- `data-xp-val`
- `help_text`
- `class`
- `label`

Methods

- `buildList()`:list<(name:str, value:str, data:dict)> : Build list of tuples (name, value) and data associated to values argument

OneListField

Select field. Will render to combobox, option lists, autocomplete, etc... when form instance is rendered, values are fetched from database to fill `id_choices` hidden field with data for field values.

In case choices is not null, we attempt to skip foreign key search for values and get values from choices object.

From choices... country = OneListField(_dbAddress, 'country', choicesId='country', required=False, choices=Choices.COUNTRY)

From fk... country = OneListField(_dbAddress, 'country', choicesId='country', required=False)

From Choices

You need to include arguments `choicesId`, `choices`.

From Foreign Key

You need to include arguments: `choicesId`. Optional `limitTo`, `orderBy` and `listValue`. In case these optional attributes not defined, will search without filter and name will be FK and value string representation of model instance.

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field, like '_myModel.fieldName'
- `choicesId`:str: Choice id to save into id_choices hidden field, like {myChoiceId: [(name1,value1),(name2,value2),...] ... }

Optional Arguments

- `limitTo`:dict : Dictionary with attributes sent to model filter method
- `repr`:str : Model field to be used for value in (name, value) pairs. By default, string notation of model used.
- `values`:list : List of values to append to 'id_choices'
- `orderBy`:tuple : Order by tuples, like ('field', '-field2'). field ascending and field2 descending.
- `choices`:list

Attributes

- instance:object
- instanceFieldName:str
- required:bool
- initial:str
- jsRequired:bool
- label:str
- helpText:str
- choicesId:str
- limitTo:dict
- orderBy:tuple
- listValue:str
- values:tuple

Visual Component Attributes

Attributes in attrs field attribute:

- choicesId
- data-xp-val
- help_text
- class
- label

Methods

- buildList():list<(name:str, value:str, data:dict)> : Build list of tuples (name, value) and data associated to values argument

PasswordField

Password field. Checks valid password

Required Arguments

- instance:object : Model instance
- insField:str : Model field

Optional Arguments

- minLength:int : Field minimum length
- maxLength:int : Field maximum length
- required:bool : Required field by back-end form valdiation
- initial:str : Initial value
- jsRequired:str : Required field by javascript validation
- label:str : Field label
- helpText:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `minLength`:str
- `maxLength`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

TimeField

Example:

where _dbModel is a form class attribute with the model instance.

Input Formats

A list of formats used to attempt to convert a string to a valid datetime.date object.

If no input_formats argument is provided, the default input formats are:

'%H:%M:%S', # '14:30:59' '%H:%M', # '14:30'

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `inputFormats`:list : Input formats
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

UserField

User id field

Required Arguments

- `instance`:object : Model instance
- `insField`:str : Model field

Optional Arguments

- `minLength`:int : Field minimum length
- `maxLength`:int : Field maximum length
- `required`:bool : Required field by back-end form valdiation
- `initial`:str : Initial value
- `jsRequired`:str : Required field by javascript validation
- `label`:str : Field label
- `helpText`:str : Field tooptip

Attributes

- `instance`:object
- `instanceFieldName`:str
- `minLength`:str
- `maxLength`:str
- `required`:bool
- `initial`:str
- `jsRequired`:bool
- `label`:str
- `helpText`:str

3.7 Component Registry

You need to register server-side components through `components.py` file.

- **Apps and Services**
- **Views**
- **Templates**
- **Actions**
- **Flows and flow data**
- **Menus**
- **Search commands**

3.7.1 Apps and Services

```
from service import SiteService
class AppReg (AppCompRegCommonBusiness):
    def __init__(self):
        super(AppReg, self).__init__(__name__)
        # Application
        self._reg.registerApp(__name__, title='My App', slug='my-app')
        # Services
        self._reg.registerService(__name__, serviceName='MyService', className=SiteService)
```

By default your app comes with SiteService. You can have as many services as you want, you can create them here and implement in `service.py`. Each use case can become a service, with views and actions associated.

3.7.2 Views

```
self._reg.registerView(__name__, serviceName='MyService', viewName='myView', slug='my-view',
                       className=SiteService, method='view_mine')
```

You refer to above service name, defining a view name, slug and which class and method implements it.

3.7.3 Templates

```
self._reg.registerTemplate(__name__, viewName='myView', name='my_template')
```

You map views and templates. Templates will be found in `templates` directory. They can be window or popup types, holding different directories for ech template type.

3.7.4 Actions

```
self._reg.registerAction(__name__, serviceName='MyService', actionName='changeStatus', slug='change-s
                            className=SiteService, method='change_status')

Your SiteService would need a method ``change_status``.
```

3.7.5 Flows and flow data

```
self._reg.registerFlow(__name__, 'activate-user', resetStart=True, deleteOnEnd=True, jumpToView=False
self._reg.registerFlowView(__name__, 'activate-user', actionName='activateUser', viewNameTarget='acti
```

You would attach additional attributes to map flow variables.

3.7.6 Menus

```
self._reg.registerMenu(__name__, name='signup', title='Signup', description='Signup', iconName='icon
                viewName='signup')
self._reg.registerServMenu(__name__, serviceName='MyService', menus=[
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.LOGIN, _K.CONDITIONS: 'notLogin:re
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.SIGNUP, _K.CONDITIONS: 'notLogin:re
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.HOME_LOGIN, _K.CONDITIONS: 'login:r
            ])
```

The first one creates menu entity. The second creates relationship between a view or service and which menu items holds. In this case, we map a service having icons login, signup and home login with conditions based on user login. In case you map services, you don't need to map menu items for each view. But you may, for example, define shortcut views related to your view. You can do that mapping menu items to views. For example, you can have a service about customer management and link views related to functionality related to your service use case like "Invite Customer".

This is how you map individual views:

```
self._reg.registerViewMenu(__name__, viewName=Views.SIGNUP, menus=[
                {_K.ZONE: _Ch.MENU_ZONE_VIEW, _K.MENU_NAME: Menus.HOME},
                {_K.ZONE: _Ch.MENU_ZONE_VIEW, _K.MENU_NAME: Menus.LOGIN}
            ])
```

We map signup page with home and login.

3.7.7 Search commands

```
self._reg.registerSearch(__name__, text='Change Password', viewName=Views.CHANGE_PASSWORD)
```

Allows to map views and actions to happen at auto-complete box in top menu. When clicked, a new view (window or popup) would come or action be executed, redirecting in this case to another view.

3.8 Commands

3.8.1 xpcomponents

Executes the `components.py` registry for an application:

```
./manage.py xpcomponents myproject.myapp
```

To register and update components for ximpia site app:

```
./manage.py xpcomponents ximpia.xpsite
```

xpcomponents will register views, templates, flows, menus, search server-side components for your application.

CHAPTER 4

Front-End

You define visual components as `div` elements in plain HTML5 template files:

```
<div id="id_fromUser_comp" data-xp-type="select.plus"
    data-xp="{  labelWidth: '100px',
                info: true,
                label: 'Sent by',
                helpText: 'User that sent invitation'}" > </div>
```

This allows your web development to be plug&play: Simply define properties for your visual objects and paste into html5 templates.

4.1 Visual Conditions

You can associate conditions to your visual objects so you control rendering and other visual behavior. When those conditions match or not, then your visual objects gain the property you associate with the condition.

Conditions are mapped in the view element `id_view` as attribute `data-xp-cond-rules`:

```
<div    id="id_view"
    data-xp="{viewName: 'signup'}"
    data-xp-cond-rules="{   hasUserAuth: 'settings.SIGNUP_USER_PASSWORD == true',
                            hasNetAuth: 'settings.SIGNUP_SOCIAL_NETWORK == true',
                            hasNotNetAuth: 'settings.SIGNUP_SOCIAL_NETWORK == false',
                            showCaptcha: '  (   isSocialLogged == false &&
                                                form_signup.invitationCode.value == \'\' &&
                                                    settings.SITE_SIGNUP_INVITATION == true     ) ||
                                            (   isSocialLogged == false &&
                                                    settings.SITE_SIGNUP_INVITATION == false     )',
                            requiresInvitation: 'settings.SITE_SIGNUP_INVITATION == true'
                        }" >
</div>
```

In this case we compare to `settings` and `isSocialLogged`. These variables would relate to the visual context. In case you want to compare with a form field, you would do: `form_myform.myfiel`. You may at your server-side write variables to the visual context with `self._add_attr('isSocialLogged', False)` from service view.

4.1.1 Comparison Operators

You can compare:

- ==
- !=
- >
- <
- >=
- <=

4.1.2 Logical Operators

- OR : `||`
- AND : `&&`

You can include parenthesis (and) to provide right logic.

4.1.3 Visual Objects Conditions

After you define your view conditions, it is time to map those conditions to your objects.

Our current release forces you to include `container` visual object, where you place condition logic. In future releases, we will have condition support in all objects.

```
<div id="id_socialAuth" style="margin-top: 5px;"
    data-xp-type="container"
    data-xp-cond="{conditions: [
                                    {condition: 'hasNetAuth', action: 'render', value: true}
                                ]}" >
<div id="id_facebookSignup_comp"
        data-xp-type="function.render"
        data-xp="{functionName: 'ximpia.external.Facebook.renderSignup'}" > </div>
    <div class="caption"><span>You can also:</span><br/></div>
    <!-- Facebook Signup Button -->
    <div class="fb-login-button"
            data-show-faces="true"
            data-width="200"
            data-max-rows="1"
            scope="email">Signup with Facebook</div>
</div>
</div>
```

When condition `hasNetAuth` matches we render facebook login component. When not, we do not render facebook login.

You can include a set of conditions to match. The first to match, we apply action and skip the rest. The only action currently supported os `render`, where you can place to `true` to show or `false` to hide.

4.2 Visual Components

4.2.1 Common Attributes

- `class` : Will add all classes in this field
- `tabindex` : Will include tabindex support for entry fields with sequence when hitting tab key.
- `readonly` : Component will be read-only.
- `maxlength` : Maximum length for entry components.
- `value` : Value.
- `name` : Name of component

4.2.2 Button

This component shows in top and bottom bars of views. The actions are mapped with ximpia actions to produce validation. Validation results will be shown in message that slides up from button or popup window.

Html

Save button:

```
<div id="id_save_comp" data-xp-type="button"
    data-xp="{  form: 'form_home',
                align: 'left',
                text: 'Save',
                mode: 'actionMsg',
                type: 'color',
                action: 'save'}" > </div>
```

Attributes

- `form` : The form id associated with the action
- `align` : left | right
- `text` : Button text, without multi-language. English only
- `mode` : actionMsg | action | closePopup | closeView | contentInsert
- `action` : Action that executes the button
- `type` : color | icon | iconPopup | simple
- `icon` : add | edit | delete | save | email | like | next | star | spark | play
- `callback` : The callback function to execute after button is clicked
- `title` [optional] : Tooltip for button
- `titleDisabled` [optional] : Title to show when state is disabled
- `clickStatus` : disable : Button status when button has been clicked and action processed.

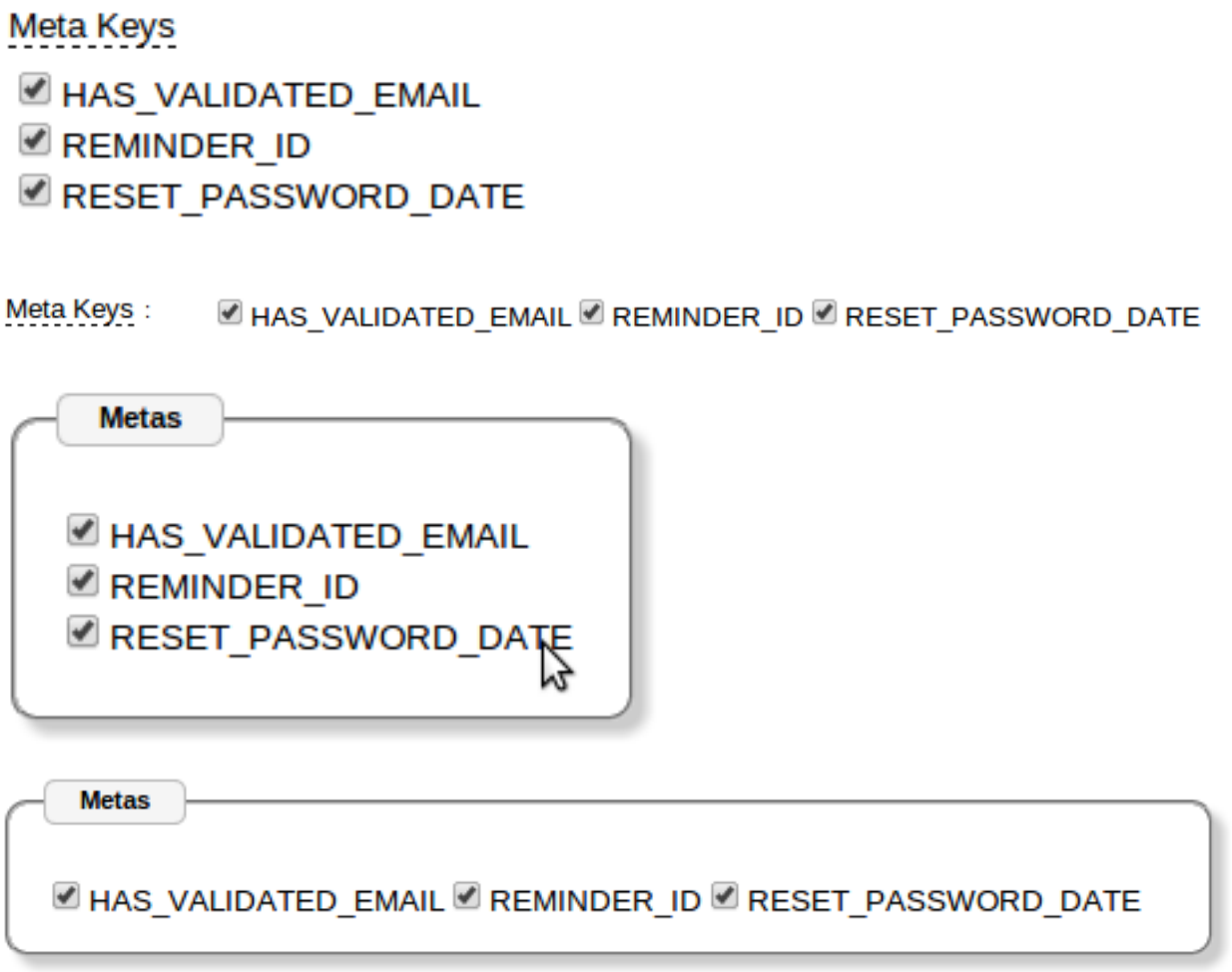
Interfaces

- `IAction`

Methods

- render
- click
- disable
- enable
- unrender

4.2.3 Check



List of options with `checkbox` render. Allows horizontal or vertical render of list of entries. Checkbox can be placed before or after the field entry and labels can be at top or left.

Html

```
<div id="id_mycheck_comp" data-xp-type="check" data-xp="{alignment: 'vertical'}" > </div>
```

Attributes

- `label` : Label
- `size` : Input box size
- `helpText` : Tooltip to show
- `info` : Weather to show tooltip.
- `labelWidth` : Width for label
- `alignment` [optional] : 'vertical', 'horizontal'
- `hasLabel` [optional] : "true" or "false". Weather to show or not a label, at left or top of check controls.
- **`labelPosition` [optional]** ['top'|'left'. Label position, left of check buttons, or top for label at one line and check] controls on a new line.
- `controlPosition` [optional] : 'before'|'after'. Default: 'before'. Position for the check control, after or before text.

Interfaces

- `IInputList` : Interface for list of entries

Methods

- `render` : Render
- `enable` : Enable
- `disable` : Disable
- `unrender` : Reset

4.2.4 Container

Set of visual components.

Supports conditions. You can place your visual objects inside containers to force condition for rendering.

Conditions are defined in the view definitions with attribute `data-xp-cond-rules`, like:

```
<div id="id_view"
      data-xp="{viewName: 'signup'}"
       data-xp-cond-rules="{hasUserAuth: 'settings.SIGNUP_USER_PASSWORD == true',
                            hasNetAuth: 'settings.SIGNUP_SOCIAL_NETWORK == true',
                            socialNetLogged: 'socialNetLogged == true'}" >
</div>
```

Html

```
<div id="id_passwordAuth" data-xp-type="container"
               data-xp-cond="{conditions: [
                                    {condition: 'socialNetLogged', render: false}]}" >
      ...your objects...
      <div ... > </div>
</div>
```

Attributes

- `data-xp-type` : container
- **`data-xp-cond` :ListType** [Condition objects, like [{}, {}, ...] First matched condition will execute action]
  - **`conditions` :ListType** [List of conditions:]
    - `condition` : condition key from `data-xp-cond-rules`
    - `action` : Supported values: 'render'
    - `value` :Boolean : true / false

4.2.5 Content

Allows to embed server-side data into templates.

Inserts into templates data from visual context. If you have `form_mine` and want to refer field `name`, you would:

```
{{form_mine.name}}
```

You can also add objects to your visual context from server-side in your services:

```
self._add_attr('customer', customer)
```

Where customer may be any serilizable object.

and you would:

```
{{customer.name}}
```

To display customer name.

Html

```
<a href="{{object.url}}" title="{{object.title}}" data-xp-type="content" >{{object.title}}</a>
```

We would have `object` in our visual context.

4.2.6 Field

Field with formatting option and tooltip with `helpText` attribute. Option to provide auto-complete from choices or server-side data.

Html

```
<div id="id_countryTxt_comp"    data-xp-type="field"
                                data-xp="{  label: 'Country Code', size: 2}"
                                data-xp-complete="{     choicesId: 'country',
                                                        choiceDisplay: 'name',
                                                        minCharacters: 1    }"> </div>
```

Attributes

- `label` : Label
- `size` : Input box size
- `helpText` : Tooltip to show
- `info` : Weather to show tooltip.
- `labelWidth` : Width for label

Attributes for auto-completion choices

- `choicesId` : Choices id to reference to show list.
- `choiceDisplay` [optional] default:value : name|value. Display either name or value from choices.
- `maxHeight` [optional] : Max height of autocomplete box
- `minCharacters` [optional] : Min characters to trigger auto-complete box.

Attributes for auto-completion server-side

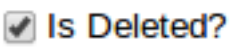
- `app` [optional] : Application code
- `dbClass` : Data class to show results from.
- `searchField` :String : Search field to match for text from input field.
- `maxHeight` [optional] : Max height of autocomplete box
- `minCharacters` [optional] : Min characters to trigger auto-complete box.
- `params` [optional] :Object : Parameters to filter completion list.
- `fieldValue` [optional] :String : Field to show results. In case not defined, will use the model string representation.
- `extraFields` [optional] :List : Fields to show in extra Object

methods

- `render` : Renders the component
- `complete` : Bind autocomplete behavior

- `enable` : Enable field
- `disable` : Disable field
- `unrender` : Reset (remove) component data and remove `data-xp-render` attribute.

4.2.7 FieldCheck



Renders fields that are BooleanField, with values true / false or 1 for true and 0 for false

Support labels. Check control can be before label or after.

Html

```
<div id="id_hasUrl_comp" data-xp-type="field.check" data-xp="{}" > </div>
```

The above code will just show a checkbox with True/False logic. You can include label as well.

This case would show checkbox and a message after the box to agree to terms and conditions in a web site:

```
<div id="id_agree_comp" data-xp-type="field.check"
                        data-xp="{  label: 'I agree to terms and conditions',
                                    controlPosition: 'before'}" > </div>
```

Attributes

- `label` [optional] : Label
- `helpText` [optional] : Tooltip to show at label
- `info` [optional] : Weather to show tooltip at label.
- `labelWidth` [optional] : Width for label
- `controlPosition` [optional] : 'before'l'after'. Default: 'before'. Position for the radio control, after or before text.

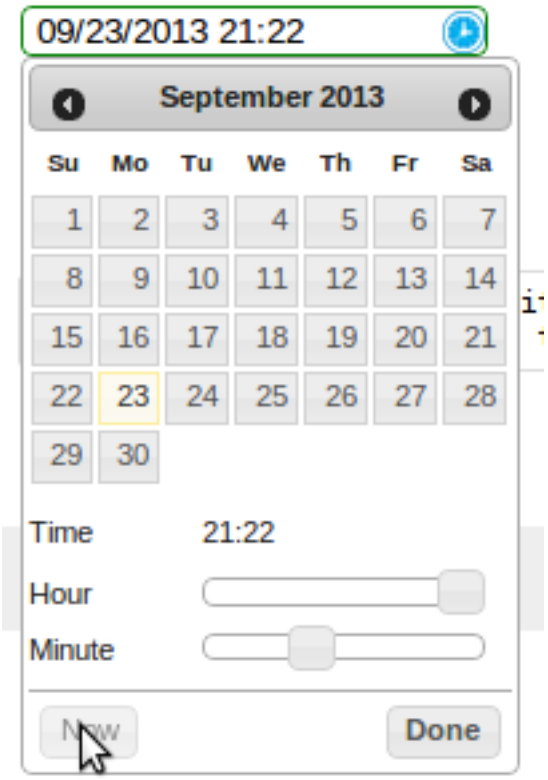
Interfaces

- `IInputField`

Methods

- `render` : Render
- `enable` : Enable
- `disable` : Disable
- `unrender` : Reset

4.2.8 FieldDateTime



- Date and Time field representation. This component renders form fields Date, DateTime and Time.
- When field type is Date, a date tooltip will popup to select date.
- When field type is Time, a time tooltip will popup to select time with two selection bars for hour and minute.
- When field type is DateTime, a date with time tooltip will show up with calendar and time bars.

Html

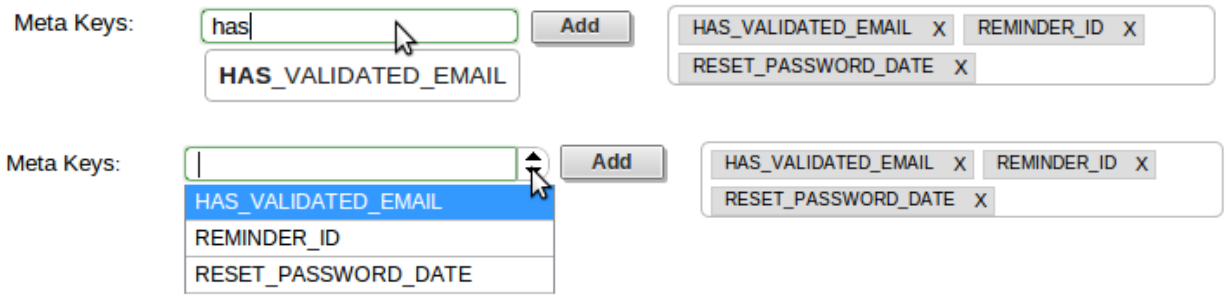
```
<div id="id_updateDate_comp" data-xp-type="field.datetime" data-xp="{}"> </div>
```

Type can be `field.datetime`, `field.date` or `field.time`.

Attributes

- `label` [optional] : Label
- `helpText` [optional] : Tooltip to show at label
- `info` [optional] : Weather to show tooltip at label.
- `labelWidth` [optional] : Width for label
- `hasLabel` [optional]
- `labelPosition` [optional]

4.2.9 FieldList



List of fields. Fields can be added and deleted. Can represent the many-to-many relationships in models.

They can be rendered as tags horizontally.

Html

Input:

```
<div id="id_meta_comp"  data-xp-type="field.list"
                    data-xp="{  type: 'field',
                                labelWidth: '100px',
                                modelField: 'meta__name'}"
                    data-xp-complete="{ choicesId: 'metaKey',
                                        minCharacters: 1  }" > </div>
```

Select:

```
<div id="id_meta_comp" data-xp-type="field.list"
            data-xp="{  type: 'select.plus',
                        selectObjId: 'id_metaKey_comp',
                        labelWidth: '100px',
                        choicesId: 'meta'}" > </div>
```

Attributes

- `type`:string [default: field] [optional] : Type of control for adding values: `field` and `select.plus` possible values.
- `labelWidth`:string [optional]
- `selectObjId`:string [optional]
- `modelField`:string [optiona] : For field input type, the model field value. Required for fields. Not required for select input.
- `choicesId`

Interfaces

- `IInputList`
- `IKeyInput`

Methods

- `render` : Render.
- `keypress` : Deals with `Enter` key stroke and adding clicked to list of entries.
- `enable` : Enable
- `disable` : Disable
- `unrender` : Reset

4.2.10 FieldNumber

Html

```
<div id="id_number_comp" data-xp-type="field.number"
                         data-xp="{  size: 2,
                                     labelWidth: '100px',
                                     info: true,
                                     helpText: 'Number of invitations'}" ></div>
```

Attributes

- `label` : Label
- `size` : Input box size
- `helpText` : Tooltip to show
- `info` : Weather to show tooltip.
- `labelWidth` : Width for label
- `hideSpinner` :Boolean : Hides spinner control

Interfaces

- `IInputField`

Methods

- `render` : Render
- `enable` : Enable
- `disable` : Disable
- `unrender` : Reset

4.2.11 Function

Allows to render content based on a javascript function.

Html

```
<div id="id_facebookSignup_comp"
    data-xp-type="function.render"
    data-xp="{functionName: 'ximpia.external.Facebook.renderSignup'}" > </div>
```

You can add attributes to `data-xp` and refer to those in your javascript function.

Your javascript function would be like:

```
ximpia.external.Facebook.renderSignup = (function(attrs, callable) {
    // Code
}
```

`attrs` are the attributes from `data-xp` html attribute.

Attributes

- `functionName` : Path to javascript function name

Any additional attributes you define

4.2.12 Link

Hyperlink which would trigger a new view or call an action.

Links used to:

1. Launch views (new and popups) - launchView
2. Open popups (openPopup) - openPopup
3. Launch actions - doAction
4. Link to url - callUrl

Types are:

- `link.popup`
- `link.url`
- `link.view`
- `link.action`

Html

```
<div id="id_passwordReminderLinkUrl_comp" data-xp-type="link.url"
        style="margin-top: 20px; margin-left: 20px"
    data-xp="{  op: 'callUrl',
                url: '/',
                target: '_blank',
                width: 800,
                height: 600,
                title: 'go to home...',
                linkText: 'Take me to Home'}" ></div>
```

```
<div id="id_lnkCode_comp" data-xp-type="link.view"
        style="margin-top: 20px; margin-left: 20px"
    data-xp="{  op: 'showView',
                app: 'ximpia_site.web',
                title: 'go to home...',
                linkText: 'Show Code',
                view: 'code'}" ></div>

<div id="id_lnkSignout_comp" data-xp-type="link.action"
        style="margin-top: 20px; margin-left: 20px"
    data-xp="{  op: 'doAction',
                app: 'ximpia.site',
                title: 'will logout...',
                linkText: 'Logout',
                action: 'logout'}" ></div>
```

Attributes

- `class`
- `textSize`

Interfaces

- `IAction`

Methods

- render
- click
- disable
- enable

4.2.13 ListContent

You include a div for the component definition and html inside this div can be any html element that will be repeated for each row in the list. You include data with {{}} notation. Response context has elements for lists with `list_myList``where myList relates to ``id_myList_comp`. This way you don't have to repeat {{list_myList.data.myField}} and only need to include {{data.myField}}. You can include list values, header values and meta values for the list.

You can include any element in three positions: jxListContentHeader, jxListContentBody and jxListContentFoot. Body position will incude the rows to be repeated in the list with values. Header will include content before list and foot includes any content you need at end of list.

```
<div id="id_myList_comp" type="list.content"
        data-xp="{dbClass: 'MyDAO', fields: ['myField']}">
<$htmlElement class="jxListContentHeader">
Here go the results...
</$htmlElement>
<$htmlElement class="jxListContentBody">
{{header.myField}}: {{data.myField}}
```

```
</$htmlElement>
<$htmlElement class="jxListContentFoot">
numberPages: {{meta.numberPages}}
</$htmlElement>
</div>
```

Example:

```
<div id="id_groups2_comp" data-xp-type="list.content"
                            data-xp="{  app: 'ximpia.xpsite',
                                        dbClass: 'GroupDAO',
                                        numberResults: 1,
                                        fields: ['id', 'group__name']   }" >
<div class="jxListContentHeader">
     These are the results you will have...
</div>
<div class="jxListContentBody" style="border: 1px solid; height: 30px; width: 600px">
     {{headers.group__name}}: {{data.id}} - {{data.group__name}}
</div>
<div class="jxListContentFoot" style="margin-top: 10px; margin-bottom: 10px">
     Number pages: {{meta.numberPages}}
</div>
<div class="jxListContentFoot" style="margin-top: 10px">
<div id="id_groups2_paging_comp" data-xp-type="paging.more"
        data-xp="{compId: 'id_groups2_comp'}" class="ui-list-content-paging">
    More Results...
</div>
</div>
```

Attributes

- `dbClass` :string
- `app` :string [optional]
- `method` :string [optional] [default:searchFields] : Data method to execute
- `fields` :object<string>
- `args` :object [optional] : Initial arguments. Object with arguments
- `orderBy` :object [optional] : Order by fields, ascending with '-' sign before field name. Supports relationships, like 'field__value'
- `disablePaging` :boolean [optional] [default: false]
- `pagingStyle` :string [optional] [default:more] : Possible values: more

Interfaces

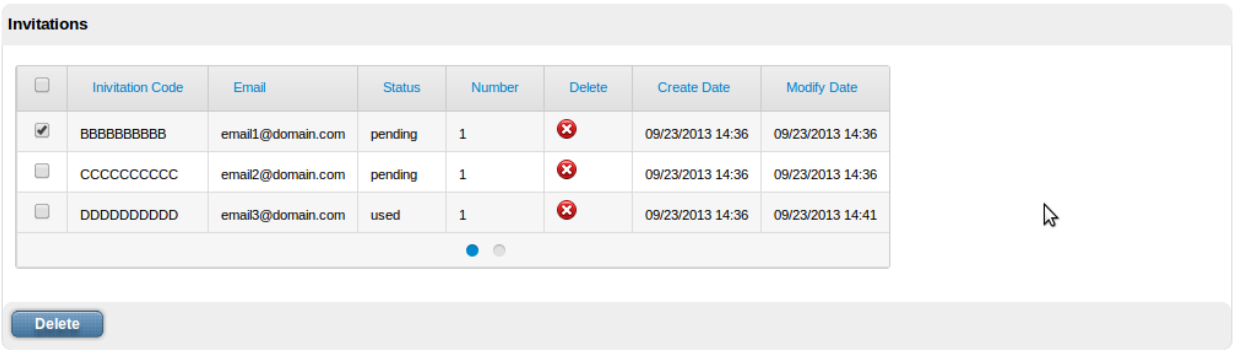
- `IList`

Methods

- `render`
- `insertRows` (xpForm:string, result:object) : Inserts rows into content list.

4.2.14 ListData

Inivitation Code	Email	Status	Number	Delete	Create Date	Modify Date
BBBBBBBBBB	email1@domain.com	pending	1		09/23/2013 14:36	09/23/2013 14:36
CCCCCCCCCC	email2@domain.com	pending	1		09/23/2013 14:36	09/23/2013 14:36
DDDDDDDDDD	email3@domain.com	pending	1		09/23/2013 14:36	09/23/2013 14:36
GGGGGGGGGG	email4@domain.com	pending	1		09/23/2013 14:37	09/23/2013 14:37
KKKKKKKKKK	email5@domain.com	pending	1		09/23/2013 14:37	09/23/2013 14:37



Check out images for PagingMore and PagingBullet for ListData paging support.

Html

```
<div id="id_myList_comp" data-xp-type="list.data"
        data-xp="{dbClass: 'MyDAO', fields: ['name','value']}" > </div>
```

Attributes

- `dbClass` :string
- `app` :string [optional]
- `method` :string [optional] [default:searchFields] : Data method to execute
- `detailView` :object [optional] <viewPath, winType>: View to display detail. hasLinkedRow must be true. Full path, like 'myProject.myApp.myView'. winType can be `window``or ``popup`
- `detailType` :string [optional] [default:window] : Window type: window, popup.
- `fields` :object<string> [optional]
- `args` :object [optional] : Initial arguments. Object with arguments
- `orderBy` :object [optional] : Order by fields, ascending with '-' sign before field name. Supports relationships, like 'field__value'
- `disablePaging` :boolean [optional] [default: false]
- `caption` :string [optional]

- `headComponents` :object [optional] : List of header components. Possible values: search|filter
- `hasCheck` :boolean [optional] : Table has operations linked to row checks. User would check rows and click button to execute actions on checked items.
- `activateOnCheck` :object : List of components to activate when row check is clicked.
- `onCheckClick` :string [optional] [default:enable] . Enable or render action components when user clicks on check.
- `hasHeader` :boolean [optional] [default:true]
- `pagingStyle` :string [optional] [default:more] : Possible values: more, bullet
- `pagingMoreText` :string [optional] [default:More Results...] : More paging text
- `hasLinkRow` :boolean [optional] [default:false]

Build Attributes

- `pageStart`
- `pageEnd`

Interfaces

- IList

Methods

- `render`
- `insertRows` (xpForm:string, result:object) : Result contains keys data, headers and meta for list result

4.2.15 Image

Renders into `img` html element.

`src` html attribute is generated using attributes `file`, `location` and `hostLocation`. Only required attribute is `file`. You can define `src` attribute with full path for images.

Html

By class:

```
<div id="id_myImage_comp" data-xp-type="image" data-xp="{imgClass: 'checkSmall'}" > </div>
```

Using images location and default host location:

```
<div id="id_myImage_comp" data-xp-type="image"
            data-xp="{file: 'github-icon-source.jpg'}" > </div>
```

Using S3 host location:

```
<div id="id_myImage_comp" data-xp-type="image"
        data-xp="{file: 'github-icon-source.jpg', hostLocation: 'S3'}" > </div>
```

Using cloudfont host location:

```
<div id="id_myImage_comp" data-xp-type="image"
        data-xp="{file: 'github-icon-source.jpg', hostLocation: 'cloudfront'}" > </div>
```

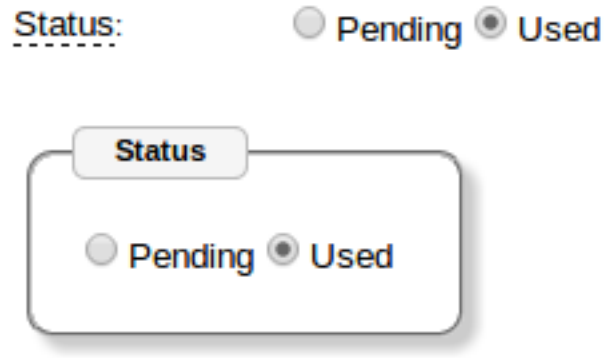
Using src:

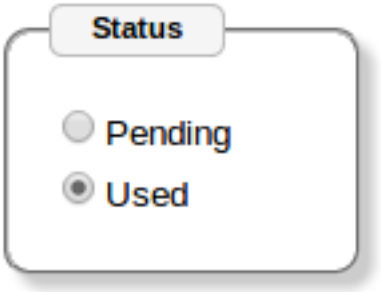
```
<div id="id_myImage_comp" data-xp-type="image"
    data-xp="{src: 'https://ximpia.s3.amazonaws.com/images/github-icon-source.jpg'}" > </div>
```

Attributes

- `imgClass` : Image file name is blank.png. Image has background from css class.
- `file` : Phisical file name with extension, like `myphoto.png`. In case version attribute is defined, phisical file name will be modified to include version in the url. In case src is defined, this field is not required.
- `location` [optional] : Location name. Locations are mapped into settings.js file. In case no location is defined, we use `images` location. Locations are mapped into paths.
- `src` [optional] : In case you want to define path instead of location. In case you have path, you don't need attributes file, location or hostLocation.
- `hostLocation` [optional] : Host location mapping to use. You can define in settings alternate host location for your images, like ximpia.settings.hostLocations['S3'] = 'https://ximpia.s3.amazonaws.com/'. In case not defined, will use the default host location. This way for images can point to S3, local, cloudfront, etc...
- `title` : Tooltip to show when mouse is placed over image.
- `version` [optional] : Version to generate url for image versions. In case to include version you need no ''dimensions''attribute. Dimensions from version will be used.

4.2.16 Option





You can have options integrated into `fieldset` html element or having labels. You have the option of horizontal or vertical layout with labels on left or top.

In case you select `type: 'check'` you would see a list of checkboxes. But they will behave like options, only one can be checked. This has advantage that no entries can be checked by default and you can check uncheck the last option if you wish, having feature to reset the option list.

Html

```
<div id="id_status_comp" data-xp-type="option"
                        data-xp="{  labelWidth: '100px',
                                    hasLabel: true,
                                    info: true,
                                    helpText: 'Invitation status'}" > </div>
```

Attributes

- `type` : 'radio', 'check' default 'radio'
- `alignment` [optional] : 'vertical', 'horizontal'. Default. horizontal.
- `hasLabel` [optional] : "true"|"false". Weather to show or not a label, at left or top of radio controls.
- `label` [optional] : Field label
- **`labelPosition` [optional]** ['top'|'left'. Label position, left of radio buttons, or top for label at one line and radio] controls on a new line. Default `left`
- `controlPosition` [optional] : 'before'|'after'. Default: 'before'. Position for the radio control, after or before text.
- `info` [optional] : Displays tooltip with helpText field data.

Interfaces

- `IInputList`

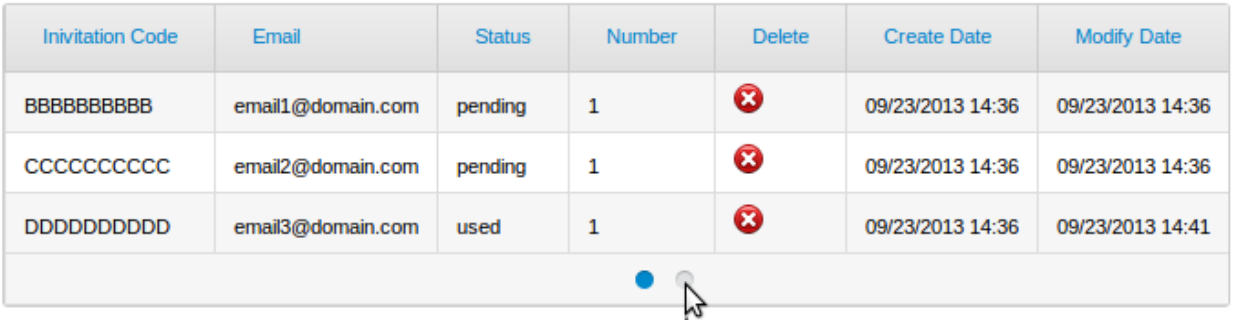
Types

- `radio` : radio option box
- `checkbox` : check box. Behaved like option, when user clicks on one, it gets selected. Ability to have no option cheched. Good for many relationships with null=true.

Methods

- render
- disable
- enable

4.2.17 PagingBullet



Bullet paging component which displays current page, next n (customized, default 5) pages with ability to jump to pages.

Current page has filled bullet. When mouse goes over, shows number of resources to fetch (1-10).

ContentList

Gets integrated into the `jxListContentFoot`:

```
<div class="jxListContentFoot" style="margin-top: 10px">
    <div id="id_groups2_paging_comp" data-xp-type="paging.bullet"
        data-xp="{  compId: 'id_groups2_comp',
                    numberPages: 2,
                    numberResources: 3}" class="ui-list-content-paging"> </div>
</div>
```

DataList

- pagingStyle: 'bullet'
- numberResults: 3
- numberPages: 2

Attributes

- `compId` :string : Id for list component
- `numberPages` :number : Number of pages for list
- `numberResources` :number : Number of resources in the list, used to display result pointers in page links (1-10, etc...)

Interfaces

- IPage

4.2.18 PagingMore

Inivitation Code	Email	Status	Number	Delete	Create Date	Modify Date
BBBBBBBBBB	email1@domain.com	pending	1		09/23/2013 14:36	09/23/2013 14:36
CCCCCCCCCC	email2@domain.com	pending	1		09/23/2013 14:36	09/23/2013 14:36
DDDDDDDDDD	email3@domain.com	used	1		09/23/2013 14:36	09/23/2013 14:41
More Results...						

Paging with `More` link. AJAX call will populate next page.

ListContent

Gets integrated into the `jxListContentFoot`:

```
<div class="jxListContentFoot" style="margin-top: 10px">
    <div id="id_groups2_paging_comp" data-xp-type="paging.more"
            data-xp="{compId: 'id_groups2_comp'}" class="ui-list-content-paging">
        More Results...
    </div>
</div>
```

ListData

- pagingStyle: 'more'

4.2.19 Select

Html

```
<div id="id_status_comp" data-xp-type="select"
                            data-xp="{  labelWidth: '100px',
                                        info: true,
                                        helpText: 'Invitation status'}" > </div>
```

Attributes

- `label` : Label
- `size` : Input box size
- `helpText` : Tooltip to show
- `info` : Weather to show tooltip.
- `labelWidth` : Width for label

Interfaces

- `IInputField`

Methods

- `render`
- `disable`
- `enable`
- `unrender`

4.2.20 SelectPlus

Html

```
<div id="id_fromUser_comp" data-xp-type="select.plus"
                        data-xp="{  labelWidth: '100px',
                                    info: true,
                                    label: 'Sent by',
                                    helpText: 'User that sent invitation'}" > </div>
```

Choices from server-side form field is used by default. You may include field attribute `choicesId` in the properties to modify defult value.

As you type text, auto-complete will drop under to help you on selection. For cases withmany entries you will have paging support on entry list to browse on different pages of results, very handy for big lists.

When you have `hasBestMatch` the best match is highligted and selected in the input box. So when you start typing in a country list "Spa", when best match is Spain will automatically get selected in the text box.

Attributes

- `label` : Label
- `size` : Input box size
- `helpText` : Tooltip to show
- `info` : Weather to show tooltip.
- `labelWidth` : Width for label
- `hasBestMatch` :String : Highlight best match relative to field text. Default true.

- `gmaps` : Google maps association, like 'country'. Used for list of countries. Google maps library will set country based on location.
- `choicesId` : Id for choice list.

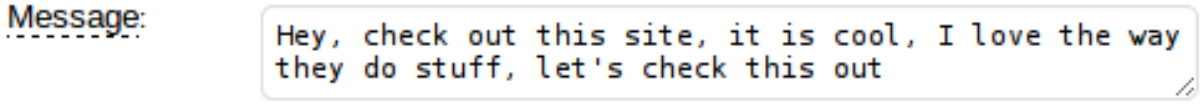
Interfaces

- `IInputField`

Methods

- `render`
- `setValue` : Sets value in selection box
- `disable`
- `enable`
- `unrender`

4.2.21 TextArea



When you set `isCollapsible: true`, as you type and get to end of row, a new row will be added to text area. This way you don't need to size text box and size adapts to size user needs to write. You may start with one or two lines, and as users type, end up with more lines.

Html

```
<div id="id_message_comp" data-xp-type="textarea"
                          data-xp="{  labelWidth: '100px',
                                      cols:50,
                                      rows:1,
                                      isCollapsible: true,
                                      info: true,
                                      helpText: 'Invitation message'}" > </div>
```

Attibutes

- `label` : Label
- `size` : Input box size
- `helpText` : Tooltip to show
- `info` : Weather to show tooltip.
- `labelWidth` : Width for label
- `cols`

- rows
- isCollapsible

Interfaces

- IInputField

Methods

- render
- disable
- enable
- unrender

4.3 Templates

Our templates are plain html5 files with visual components as div elements and variable placeholders with {{}} notation, providing a logic-less template system.

You can place conditions inside Container visual component. You set conditions in view definition and attach rendering actions to conditions defined.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Ximpia - Change Password</title>
</head>
<body>
<div id="id_popup"
        data-xp="{title: 'Change Password'}" ></div>
<!-- Content -->
<section id="id_content" class="sectionContent">
<div id="id_changePassword">
<form id="form_userChangePassword" action="" method="post" data-xp="{}">
<!-- ximpiaId -->
<div id="id_username_comp"
        data-xp-type="field"
        data-xp="{tabindex: '1', label: 'XimpiaId', 'readonly': 'readonly'}" > </div>
<!-- password -->
<div id="id_password_comp" data-xp-type="field"  style="margin-top: 10px"
        data-xp="{type: 'password', info: true}" ></div>
<!-- newPassword -->
<div id="id_newPassword_comp" data-xp-type="field" style="margin-top: 10px"
        data-xp="{type: 'password', info: true, class: 'passwordStrength'}" ></div>
<!-- newPasswordConfirm -->
<div id="id_newPasswordConfirm_comp" data-xp-type="field" style="margin-top: 10px"
        data-xp="{type: 'password', info: true}" ></div>
</form>
</div>
<br/>
</section>
```

```
<!-- Content -->
<!-- Page Button Bar -->
<section id="id_sectionButton" class="sectionButton">
<div id="id_popupButton" class="btBar">
<div id="id_doChangePassword_comp" data-xp-type="button"
                data-xp="{  form: 'form_userChangePassword',
                            align: 'right',
                            text: 'Save',
                            type: 'iconPopup',
                            mode: 'actionMsg',
                            action: 'changePassword',
                            clickStatus: 'disable',
                            icon: 'save'}" ></div>
</div>
</section>
<!-- Page Button Bar -->
</body>
</html>
```

4.3.1 Visual Context

Server-side data is serialized into a visual context, a `JSON` data entity having forms, lists and other attributes you may insert from your service views and actions.

Example visual context for ximpia.com home page:

```
{   'errors': [],
    'response': {   'app': 'ximpia_site',
                    'appSlug': u'front',
                    'defaultApp': 'front',
                    'form_home': {   'ERR_GEN_VALIDATION': {   'class': 'field',
                                                               'data-xp-type': 'input.hidden',
                                                               'fieldType': 'HiddenField',
                                                               'helpText': u'',
                                                               'jsRequired': None,
                                                               'label': None,
                                                               'name': 'ERR_GEN_VALIDATION',
                                                               'required': None,
                                                               'type': 'hidden',
                                                               'value': u'Error validating your data
                                     'action': {   'class': 'field',
                                                   'data-xp-type': 'input.hidden',
                                                   'fieldType': 'HiddenField',
                                                   'helpText': u'',
                                                   'jsRequired': None,
                                                   'label': None,
                                                   'name': 'action',
                                                   'required': None,
                                                   'type': 'hidden',
                                                   'value': ''},
                                     'app': {   'class': 'field',
                                                'data-xp-type': 'input.hidden',
                                                'fieldType': 'HiddenField',
                                                'helpText': u'',
                                                'jsRequired': None,
                                                'label': None,
                                                'name': 'app',
```

```
                'required': None,
                'type': 'hidden',
                'value': 'ximpia_site'},
 'buttonConstants': {   'class': 'field',
                        'data-xp-type': 'input.hidden',
                        'fieldType': 'HiddenField',
                        'helpText': u'',
                        'jsRequired': None,
                        'label': None,
                        'name': 'buttonConstants',
                        'required': None,
                        'type': 'hidden',
                        'value': u"[['close','Close']]"},
 'choices': {   'class': 'field',
                'data-xp-type': 'input.hidden',
                'fieldType': 'HiddenField',
                'helpText': u'',
                'jsRequired': None,
                'label': None,
                'name': 'choices',
                'required': None,
                'type': 'hidden',
                'value': '{}'},
 'dbObjects': {   'class': 'field',
                  'data-xp-type': 'input.hidden',
                  'fieldType': 'HiddenField',
                  'helpText': u'',
                  'jsRequired': None,
                  'label': None,
                  'name': 'dbObjects',
                  'required': None,
                  'type': 'hidden',
                  'value': '{}'},
 'entryFields': {   'class': 'field',
                    'data-xp-type': 'input.hidden',
                    'fieldType': 'HiddenField',
                    'helpText': u'',
                    'jsRequired': None,
                    'label': None,
                    'name': 'entryFields',
                    'required': None,
                    'type': 'hidden',
                    'value': '{}'},
 'errorMessages': {   'class': 'field',
                      'data-xp-type': 'input.hidden',
                      'fieldType': 'HiddenField',
                      'helpText': u'',
                      'jsRequired': None,
                      'label': None,
                      'name': 'errorMessages',
                      'required': None,
                      'type': 'hidden',
                      'value': '{}'},
 'facebookAppId': {   'class': 'field',
                      'data-xp-type': 'input.hidden',
                      'fieldType': 'HiddenField',
                      'helpText': u'',
                      'jsRequired': None,
```

```
                        'label': None,
                        'name': 'facebookAppId',
                        'required': None,
                        'type': 'hidden',
                        'value': ''},
 'msg_ok': {   'class': 'field',
               'data-xp-type': 'input.hidden',
               'fieldType': 'HiddenField',
               'helpText': u'',
               'jsRequired': None,
               'label': None,
               'name': 'msg_ok',
               'required': None,
               'type': 'hidden',
               'value': u' '},
 'okMessages': {   'class': 'field',
                   'data-xp-type': 'input.hidden',
                   'fieldType': 'HiddenField',
                   'helpText': u'',
                   'jsRequired': None,
                   'label': None,
                   'name': 'okMessages',
                   'required': None,
                   'type': 'hidden',
                   'value': '{}'},
 'params': {   'class': 'field',
               'data-xp-type': 'input.hidden',
               'fieldType': 'HiddenField',
               'helpText': u'',
               'jsRequired': None,
               'label': None,
               'name': 'params',
               'required': None,
               'type': 'hidden',
               'value': '{"viewMode": ["update", "delete"]}'},
 'pkFields': {   'class': 'field',
                 'data-xp-type': 'input.hidden',
                 'fieldType': 'HiddenField',
                 'helpText': u'',
                 'jsRequired': None,
                 'label': None,
                 'name': 'pkFields',
                 'required': None,
                 'type': 'hidden',
                 'value': '{}'},
 'result': {   'class': 'field',
               'data-xp-type': 'input.hidden',
               'fieldType': 'HiddenField',
               'helpText': u'',
               'jsRequired': None,
               'label': None,
               'name': 'result',
               'required': None,
               'type': 'hidden',
               'value': ' '},
 'siteMedia': {   'class': 'field',
                  'data-xp-type': 'input.hidden',
                  'fieldType': 'HiddenField',
```

```
                                        'helpText': u'',
                                        'jsRequired': None,
                                        'label': None,
                                        'name': 'siteMedia',
                                        'required': None,
                                        'type': 'hidden',
                                        'value': '/static/media/'},
                            'viewNameSource': {   'class': 'field',
                                                  'data-xp-type': 'input.hidden',
                                                  'fieldType': 'HiddenField',
                                                  'helpText': u'',
                                                  'jsRequired': None,
                                                  'label': None,
                                                  'name': 'viewNameSource',
                                                  'required': None,
                                                  'type': 'hidden',
                                                  'value': ''},
                            'viewNameTarget': {   'class': 'field',
                                                  'data-xp-type': 'input.hidden',
                                                  'fieldType': 'HiddenField',
                                                  'helpText': u'',
                                                  'jsRequired': None,
                                                  'label': None,
                                                  'name': 'viewNameTarget',
                                                  'required': None,
                                                  'type': 'hidden',
                                                  'value': ' '}},
                'isDefaultApp': False,
                'isLogin': False,
                'menus': {   'main': [],
                             'service': [   {   'action': '',
                                                'app': u'ximpia_site',
                                                'appSlug': u'front',
                                                'description': u'Home',
                                                'icon': u'iconHome',
                                                'image': '',
                                                'isCurrent': True,
                                                'isDefaultApp': True,
                                                'items': [],
                                                'name': u'home',
                                                'params': {   },
                                                'sep': False,
                                                'service': u'Web',
                                                'title': u'Home',
                                                'view': u'home',
                                                'viewSlug': u'home',
                                                'winType': u'window',
                                                'zone': u'service'}],
                             'sys': [],
                             'view': []},
                'settings': {   u'NUMBER_RESULTS_LIST': 50,
                                u'SIGNUP_SOCIAL_NETWORK': False,
                                u'SIGNUP_USER_PASSWORD': True,
                                u'SITE_SIGNUP_INVITATION': False},
                'tmpl': {   u'home': u'home'},
                'view': 'home',
                'viewSlug': u'home',
                'winType': u'window'},
```

```
    'status': 'OK'}
```

4.4 Menu

Menu items get generated from the information you define in `components.py` configuration file.

Example:

```
from ximpia.xpcore.choices import Choices as _Ch
import ximpia.xpcore.constants as _K

self._reg.registerViewMenu(__name__, viewName=Views.HOME_LOGIN, menus=[
                {_K.ZONE: _Ch.MENU_ZONE_SYS, _K.MENU_NAME: Menus.SYS},
                {_K.ZONE: _Ch.MENU_ZONE_SYS, _K.GROUP: Menus.SYS,
                    _K.MENU_NAME: Menus.CHANGE_PASSWORD},
                {_K.ZONE: _Ch.MENU_ZONE_SYS, _K.GROUP: Menus.SYS,
                    _K.MENU_NAME: Menus.SIGN_OUT},
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.HOME_LOGIN}
            ])
self._reg.registerViewMenu(__name__, viewName=Views.ACTIVATION_USER, menus=[
                {_K.ZONE: _Ch.MENU_ZONE_VIEW, _K.MENU_NAME: Menus.HOME},
                {_K.ZONE: _Ch.MENU_ZONE_VIEW, _K.MENU_NAME: Menus.LOGIN}
            ])
self._reg.registerViewMenu(__name__, viewName=Views.SIGNUP, menus=[
                {_K.ZONE: _Ch.MENU_ZONE_VIEW, _K.MENU_NAME: Menus.HOME},
                {_K.ZONE: _Ch.MENU_ZONE_VIEW, _K.MENU_NAME: Menus.LOGIN}
            ])
self._reg.registerServMenu(__name__, serviceName=Services.USERS, menus=[
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.LOGIN,
                    _K.CONDITIONS: 'notLogin:render:True'},
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.SIGNUP,
                    _K.CONDITIONS: 'notLogin:render:True'},
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.HOME_LOGIN,
                    _K.CONDITIONS: 'login:render:True'}
            ])
```

4.4.1 Zones

- `sys` : Entries displayed when you click on logo item and drop down displays.
- `main` : Entries right to search box, which generally would be without text (Small icons)
- `view` : Entries of views either associated to services or views related to currentl view.

4.4.2 Services

You can register views inside services in menu or you can use the `registerServMenu` which has some highlight for current view in a service. All views from services would be visible (when you register them) and users would know which view they are in. It is nice for SiteService and small sites, where you have 4 or 5 views and users know which site zone they are in.

Also nice with services that you want users to have some 'location' of where they are.

4.4.3 Linked Views

You can link related views to current one from the view zone. It is a nice way to tell users what related features exist on current view. These additional views will add value to information shown to user.

These views can be from other services within your app or services in another apps users have access.

Views would be popup or normal full views.

4.4.4 Conditions

You may associate conditions to your menu with conditions based on visual context:

```
self._reg.registerCondition(__name__, 'notLogin', 'isLogin == false')
self._reg.registerCondition(__name__, 'login', 'isLogin == true')

self._reg.registerServMenu(__name__, serviceName=Services.USERS, menus=[
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.LOGIN,
                    _K.CONDITIONS: 'notLogin:render:True'},
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.SIGNUP,
                    _K.CONDITIONS: 'notLogin:render:True'},
                {_K.ZONE: _Ch.MENU_ZONE_MAIN, _K.MENU_NAME: Menus.HOME_LOGIN,
                    _K.CONDITIONS: 'login:render:True'}
            ])
```

The home login button will only display when we are logged in. Otherwise, will not render.

4.5 Search

We provide a search box with auto-complete to have a shortcut place where call all views and actions.

You can register which views and actions show up in search auto-complete and also associate specific data to be searchable, like customer names for example.

```
self._reg.registerSearch(__name__, text='Change Password', viewName=Views.CHANGE_PASSWORD)
```

When you want to register data, you would do in your services:

```
search = SearchService(self._ctx)
search.add_index(customer_name, self._app, view_name='show_customer',
    action_name=None, params={customer_id=876}):
```

This would add to search index a particular customer. Would show `customer_name` in auto-complete, which would be linked to view `show_customer` with attributes `customer_id=876`.

CHAPTER 5

Release Notes

5.1 Release Notes

5.1.1 0.2.1

Adds changes to navigation system, default app, minor changes and bug fixes.

Workflow Changes

- **No workflow decorator for workflow views**
- **Decorators don't need to configure flow code**
- **Workflow views don't need flow decorator**
- **No event flow links**: Flow links can be triggered when flow variables match your criteria, with or without executing actions.
- **Flow META**: A new table for meta variable/value has been added. As new workflow configuration variables are added, we don't need to change model structure, just a data migration with new workflow meta variables. Currently we have meta variables for reset on start of flow, delete user data on end and jump to last view by user.

Default App

You can configure default app in `settings.py`. When building urls using slugs, for components related to your default app, we do not show app slug, all views come from root path, like `/contact-us`.

If you want to disable, just have default app to ''

Upgrading

You need to migrate the ximpia apps: `ximpia.xpcore` and `ximpia.xpsite`:

```
python manage.py migrate ximpia.xpcore ximpia.xpsite
```

Since we now add request property to services, we need to inyect request into site service at your app views.py file:

```
@context_view(__name__)
@view_tmpl(__name__)
def home(request, **args):
    # Instantiage SiteService.home and return result
    site = SiteService(args['ctx'])
    site.request = request
```

```
    result = site.viewHome()
    return result
```

5.1.2 0.2.0

First mayor relase of Ximpia, adding visual components, service oriented architecture and ximpia-app building app script.

CHAPTER 6

Contributing

Best way to contribute is to help us with visual components already identified and under development or provide your own visual components to be included in our releases.

You can check our visual components at GitHub: Visual Components

To contribute send us a message https://ximpia.com/contact-us

CHAPTER 7

Code

https://github.com/Ximpia/ximpia/

CHAPTER 8

Website

https://ximpia.com