# Ximpia Documentation

***Release 0.2***

**Jorge**

November 18, 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 Context

Instead of request and response, this framework uses context which contains session, cookies and it is shared in all layers: service, business and data. We have full contexts with all data from front-end like forms, flow, session and cookies and minimized context with common data for business and data layers. Service layer uses full context and other layers interchange minimized context.

You can extend our common context to include extended attributes needed for common parts of your services, useful when you need custom decorators for your applications.

You can write data into context from service and business layers to interchange domain layer information like user metadata, user profile information, etc...

### 1.1.3 Services

Use cases are materialized in the framework as services. Services will hold view methods, action methods and validation operations.

Think of Apps as collections of services, and services collections of views and operations:

```
import messages as _m

class SiteService(CommonService):

        @validation()
```

```
        def _validate_invitation_pending(self, invitation_code):
                """
                Validates that invitation is pending
                """
                setting = self._get_setting(K.SET_SITE_SIGNUP_INVITATION)
                (K.SET_SITE_SIGNUP_INVITATION, setting.is_checked()) )
                if setting.is_checked():
                        self._validate_exists([
                                        [self._dbInvitation,
                                        {'invitationCode': invitation_code,
                                        'status': K.PENDING},
                                        'invitationCode', _m.ERR_invitation_not_valid] ])

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

        @action(forms.HomeForm)
        def activateGroup(self):
                """Activate group"""
                groups = self._get_list_pk_values('groups')
```

In case vaidation is not checked, user will see message `ERR_invitation_not_valid`

In case you have use cases with a set of operations, you can choose to materialize those into a service class or have n services with ‘‘do‘‘operation or similar.

You register services to map them into database and you would write code by extending from `CommonService` and have methods for views, actions, workflow views, etc...(components.py):

```
self._reg.registerService(__name__, serviceName='My Service', className=SiteService)
```

### 1.1.4 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')
```

### 1.1.5 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.

### 1.1.6 Workflow

It allows you to glue together your views (navigation) without writing code, just defining your flow with views and actions. You define views, actions associated to views and flow variables that must met in order to satisfy flow. These variables will behave like conditions for your application flow.

Your layers may write parameters to flow as you do with sessions. Session data starts when user starts flow and end when flow ends. There is a set of parameters that control the way flows behave to adapt to your needs.

You would register flow parameters through components.py file:

```
self._reg.registerFlow(__name__, flowCode='login')
self._reg.registerFlowView(__name__, flowCode='login', viewNameSource='login',
        viewNameTarget='homeLogin', actionName='login', order=10)
```

### 1.1.7 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')
```

### 1.1.8 Templates

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

```
myproject/myapp/templates
```

You will find directories for templates. By default, you will find your app directory which would keep `window` and `popup` directories. You can define templates for other apps within your application extending their templates, like you would do for our `xpsite` app.

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.

You have base template code for your application at `myproject/myapp/templates/dir/myapp.html`:

```
<!DOCTYPE html>
<html>
<head>
        <!-- Place style sheets here ... -->
</head>
<body>
<footer>
</footer>
<!-- Your javascript here ... -->
</body>
</html>
```

You can link your visual components and apply style themes.

### 1.1.9 Visual Components

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.

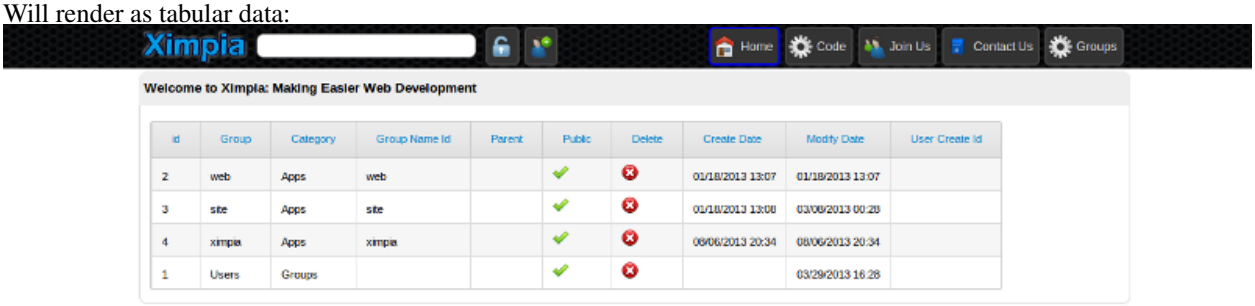
We provide a set of visual components tailored for most needs and you only need to parametrize them in HTML5 templates. Most of the time, you will not need to develop js code, simply configure the components with HTML5 `data-` attributes.

In case this is not enough for you, you can write your own components. We also provide a js rendering component: you define the js function to do render logic. This is useful for integrating external js code into your application.

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:



## 1.2 Example

Here goes the example for a change password popup. When user clicks on menu icon, a change password popup shows with current password and new password. The popup has button send to send new password to server.

### 1.2.1 Model

Since change password uses django User model, you have no changes in model for this example

### 1.2.2 Visual Components

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

```
<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>
```

All components are `field` type. You include attributes in `data-` html5 attrs.

Form would need to have `_XP_FORM_ID` equal to `changePassword`, the id for the `div` element.

### 1.2.3 Form

```
class ChangePasswordForm(XBaseForm):
        _XP_FORM_ID = 'changePassword'
        _dbUser = User()
        username = UserField(_dbUser, 'username', label='Username')
        newPassword = PasswordField(_dbUser, 'password', minLength=6,
                label='Password', helpText = _('Your New Password'))
        newPasswordConfirm = PasswordField(_dbUser, 'password', minLength=6,
                label='Confirm Password',
                helpText = _('Write again your password to make sure there are no errors'))
        errorMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['ERR_change_password']]))
        okMessages = HiddenField(initial=_jsf.buildMsgArray([_m, ['OK_PASSWORD_CHANGE']]))
        def clean(self):
                """Clean form"""
```

```
        self._validate_same_fields([('newPassword','newPasswordConfirm')])
        self._xp_clean()
        return self.cleaned_data
```

The form has cross validation for new password and confirmation for password. Template must reference `_XP_FORM_ID`.

Next step would be implement view for popup.

### 1.2.4 View

```
@view(forms.UserChangePasswordForm)
def view_change_password(self):
        """Change password form with current password and new password
        """
        self._put_form_value('username', self._ctx.user.username)
```

We need to insert user from context. In this case we have no database value since we have user in context. Otherwise, we would not need to call `_put_form_value` and method would only have `pass` command.

For database fields cases, form would populate your field data from database without need to code inside view.

### 1.2.5 Action

Code to run when Save button is clicked by user:

```
@validation()
def _validate_user(self):
        """Validate user: Check user password"""
        self._ctx.user = self._authenticate_user(self._ctx.user,
                self._f()['password'], 'password', _m.ERR_wrong_password)

@action(forms.UserChangePasswordForm)
def change_password(self):
        """Change password from user area
        """
        self._dbUser = self._instances(UserDAO)[0]
        self._validate_user()
        user = self._dbUser.get(username= self._ctx.user)
        user.set_password(self._f()['newPassword'])
        user.save()
```

1. Get user data instance with context inyected
2. Validate user: Will show message `ERR_wrong_password` in case user does not authenticate.
3. Call ‘‘set_password‘‘django method with new password and save changes

### 1.2.6 Registering

```
# view
self._reg.registerView(__name__, serviceName='Users', viewName='changePassword',
        slug='change-password', className=SiteService, method='view_change_password',
        hasAuth=True, winType='popup')
# template
self._reg.registerTemplate(__name__, viewName='changePassword', name='changePassword',
```

```
        winType='popup')
# action
self._reg.registerAction(__name__, serviceName='Users', actionName='changePassword',
        slug='change-password', className=SiteService, method='change_password',
        hasAuth=True)
# search
self._reg.registerSearch(__name__, text='Change Password', viewName='changePassword')
```

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 Setup Application

To start your application, type:

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

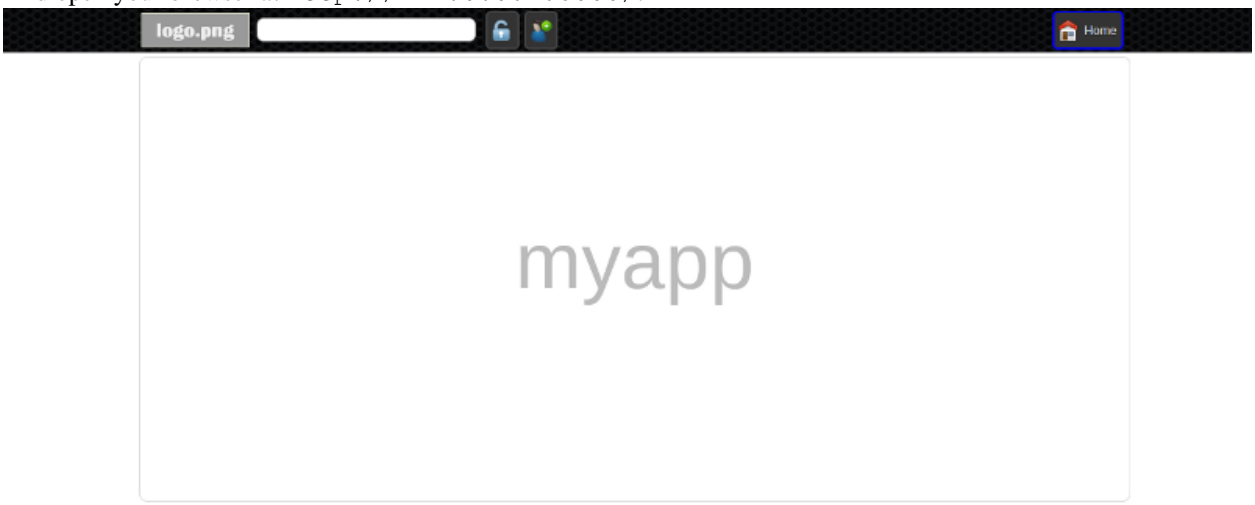
It will create folders and files needed for your application in ximpia. It will prompt for basic information like database connection user, admin name and password and locale.

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 develop your app: views, actions, etc...

CHAPTER 3

# 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 4

# Code

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

CHAPTER 5

# Website

https://ximpia.com