차례

Xpressengine 매뉴얼

이 문서는 Xpressengine을 사용하거나, Xpressengine(이하 XE)의 플러그인을 제작하고자 하는 개발자들을 위한 매뉴얼입니다.

이 문서는 아래와 같은 순서를 가지고 있습니다.

시작하기

우선 XE를 설치하는 방법을 설명합니다. XE를 설치하기 위한 서버 요구사항을 알아보고 리눅스와 윈도우 환경에서 다양한 방법으로 설치하는 방법을 안내합니다.

사이트 설정

XE를 설치한 후, 사이트의 다양한 기본 설정을 관리하는 방법을 알아봅니다. 또, 개발할 때 필요한 디버깅 환경을 설정해보고 디버깅하는 방법에 대해 알아봅니다.

XE 구조

XE를 구성하는 여러 디렉토리에 대해 알아보고, XE에서 중요한 개념인 컴포넌트, 플러그인, 서비스에 대하여 설명합니다.

기본 사용법

대부분의 웹어플리케이션은 브라우저로부터 요청(Request)을 받고, 이를 처리한 후 응답(Response)을 반환하는 라이프사이클을 가지고 있습니다. 라이프 사이클에서는 XE는 어떤 과정을 거쳐 요청을 처리한 후 응답을 보내는지에 대해 설명하고, 라이프 사이클을 거치면서 만나는 여러 구성요소에 대하여 자세히 설명합니다.

플러그인

XE를 확장하는 유일한 방법인 플러그인을 제작하는 방법과 플러그인의 구조에 대하여 설명합니다.

서비스

XE 플러그인을 개발할 때, 개발자는 XE에서 제공하는 서비스(여러가지 기능과 라이브러리)를 로드해서 사용해야 합니다. 이 장에서는 XE에서 제공하는 다양한 서비스를 사용하는 방법에 대하여 안내합니다.

컴포넌트 제작 가이드

XE의 여러가지 컴포넌트(테마, 스킨, 모듈 등)를 제작하는 방법에 대하여 자세히 설명합니다.

현재 누락된 매뉴얼

- laravel config 사용법
- 플러그인 캐싱 안내
- widget
- widgetbox
- theme - widgetbox

설치하기

서버 요구사항

XE를 설치하기 위해서는 아래의 요구사항이 만족되어야 합니다.

- 웹서버(apache, nginx 등)
- PHP 5.5.9 이상
  - OpenSSL PHP Extension
  - PDO PHP Extension
  - Mbstring PHP Extension
  - Tokenizer PHP Extension
  - GD PHP Extension
- MariaDB or MySQL 5.1 이상
- 터미널 접속 환경
- 디스크 300M 이상의 여유 공간
  - 500M 이상 권장

알려진 문제점

- Cafe24 **10G 광아우토반 FullSSD** 웹 호스팅 서비스에서 **UTF-8 (PHP7.0, mariadb-10.0.x)** 옵션만 지원합니다. **10G 광아우토반 FullSSD**의 다른 옵션을 신청했을 경우 XE3는 정상 동작하지 않을 수 있습니다.

인스톨러를 이용한 설치

Linux

터미널에서 아래와 같이 명령어를 실행합니다.

```
$ php -r "copy('http://start.xpressengine.io/installer', 'installer');" && php installer install
```

> **NOTE:** 일부 호스팅 환경에서 작동이 안 될 수 있습니다. 작동이 안 될 경우, 아래 명령어를 실행해보시기 바랍니다.
>
> ```
> $ php -d display_errors=1 -d error_reporting=-1 -d allow_url_fopen=1 -r "copy('http://start.xpressengine.io/in
> staller', 'installer');" && php installer install
> ```

Window

> Git 설치 터미널 환경을 위해 Git을 설치합니다. Git(준비중) 다운로드 및 설치를 참고하세요

Git-Bash를 실행하고 아래와 같이 명령어를 실행합니다.

```
$ php -r "copy('http://start.xpressengine.io/installer', 'installer');" && php installer install
```

위 명령어를 실행하면 설치가 시작됩니다. 안내에 따라 설치 정보를 입력하십시오.

Git 을 이용한 설치

> Git을 사용하면 업데이트및 현재 개발중인 코드를 손쉽게 적용할 수 있습니다. 코어 버전 업데이트할 때 FTP 없이 Git 을 통해 업데이트 할 수 있습니다.

Github 저장소 파일을 이용해 설치합니다

```
$ git clone https://github.com/xpressengine/xpressengine.git
$ cd xpressengine
$ composer install
...
$ php artisan xe:install
...
```

위 명령어를 실행하면 설치가 시작됩니다. 안내에 따라 설치 정보를 입력하십시오.

설치 정보 입력

1. 데이터베이스, 사이트 정보 입력

인스톨러 캡쳐 이미지(database, site 정보 입력 / 엔터) 인스톨러는 Database에 필요한 테이블을 생성하고, 기본적인 설정 파일을 생성합니다. 이 작업은 시간이 오래 걸릴 수 있습니다.

- Host [localhost] : Database 주소. 기본 `localhost`
- Port [3306] : Database prot. 기본 `3306`
- Database name : Database name
- UserId [root] : Database user id. 기본 `root`
- Password [] : Database user password
- site url [http://mysite.com] : 홈페이지 주소 입력.
  > 하위 디렉토리에 설치 할 경우 하위 디렉토리까지 입력해야 합니다.
- Timezone [Asia/Seoul] : 타임존 정보를 입력합니다. 기본 `Asia/Seoul`
  > 타임존 에서 원하는 지역의 시간대를 입력하세요.
- locale [] : 언어를 입력합니다. 영어, 한국어 두가지 언어를 지원합니다.
  > 다른 언어의 설치는 인스톨 후에 언어팩을 업로드해서 사용가능합니다. RC 버전에서 지원할 예정입니다.

2. 관리자 정보 입력

인스톨러 캡쳐 이미지(관리자 정보 입력)

- Email : 관리자 이메일
- Name [admin] : 관리자 이름. 기본 `admin`
- Password : 관리자 비밀번호
- Password again : 관리자 비밀번호 확인

3. 디렉토리 권한 및 서버 정보 수집 동의

인스톨러 캡쳐 이미지(설정)

- ./storage directory permission [0707] : /storage 디렉토리 권한 설정. 기본 `0707`
- ./bootstrap/cache directory permission [0707] : /bootstrap/cache 디렉토리 권한 설정. 기본 `0707`
- Do you agree to collect your system environmental information? [yes] : 서버 환경 정보 수집 동의. 기본 `yes`
  > 더나은 서비스 제공을 위해 설치된 서버의 환경을 수집하고 있습니다. 서버, 웹서버, PHP, Database 등의 정보를 수집합니다.

설정 파일을 이용한 설치

설정파일을 사용하면 더욱 쉽게 설치할 수 있습니다. 설치하기 전에 아래와 같이 커맨드를 실행하여 설정파일을 생성합니다.

```
$ php -r "copy('http://start.xpressengine.io/installer', 'installer');" && php installer make
```

xe_install_config.yaml 파일이 생성됩니다. 파일을 열고 설치 정보를 입력하세요. 설치 커맨드를 실행합니다. --config 및 --no-interact 옵션을 사용하십시오.

```
$ php installer install --config=.xe_install_config.yaml --no-interact
```

설치옵션

- --config=< configfile> 설정파일을 지정합니다.
- --no-interact 대화형입력을 사용하지 않고 설정파일의 정보를 사용하여 자동으로 설치합니다. 이 옵션을 --config옵션과 같이 사용해야 합니다.
- --install-dir 설치경로를 지정합니다. 지정하지 않을 경우 현재 디렉토리에 설치합니다.

웹 인스톨러로 설치하기

Composer나 Console 접속을 어려워하는 사용자를 위해 웹 브라우져에서 설치할 수 있도록 웹 인스톨러를 제공합니다.

웹 인스톨러로 설치 후 사이트 운영에서 권한관련 문제가 발생할 수 있습니다.

설치 동영상

이해를 돕기 위해 설치 영상을 마련했습니다. 바로가기

FileZila

FTP는 FileZila 를 사용해서 설명합니다. FileZila 는 무료로 사용이 가능한 프로그램 입니다. 다운로드

설치 파일 다운로드

- http://start.xpressengine.io/latest.zip 을 다운로드 합니다.
- 다운로드 받은 zip 파일의 압축을 풀고 서버에 업로드 합니다. (약 100MB)

디렉토리 권한 설정

웹 서버가 파일을 쓸 수 있도록 권한을 설정합니다.

> 권한 설정할 때 `하위 디렉터리로 이동`, `모든 파일과 디렉터리에 적용` 을 반드시 체크해 주세요.

1. bootstrap/cache 디렉토리 권한 설정
2. config/production 디렉토리 권한 설정
3. plugins 디렉토리 권한 설정
4. storage 디렉토리 권한 설정
5. vendor 디렉토리 권한 설정
6. composer.lock 파일 권한 설정

웹 인스톨러 실행

설치할 사이트에 접속하면 인스톨 화면으로 이동됩니다.

> 만약 하위 디렉토리에 설치할 경우는 해당 디렉토리로 접속해 주세요.

알려진 문제점

- FTP의 파일 업로드 오류

  > 파일 업로드 및 디렉토리 설정을 완료하고 웹 인스톨러 접근할 때 오류가 발생하는 경우가 있습니다. 이 문제는 FTP 파일 업로드 중 누락된 있어 발생할 수 있는 문제 입니다. 해결하기 위해서 FTP로 다시 업로드 해야합니다. 동일 조건을 업로드할 경우 비슷한 오류가 계속해서 발생할 수 잇으므로 중복파일 `건너뛰기` 옵션으로 업로드 해보는걸 권장합니다. 이미지

- 웹 서버 타임아웃

  > 서버 성능에 따라 웹 서버 타임아웃 설정에 의해 설치에 실패할 가능성이 있습니다. 이 문제는 웹서버 설정을 변경해야 하는 아주 복잡한 문제 입니다.

윈도우 개발환경

영상보기

서버 환경 설정

...

업데이트

XE 업데이트는 3.0.0-beta 버전부터 지원합니다. 만약 현재 설치된 XE가 그 이전 버전이면 새로 설치하셔야 합니다.

최신버전 다운로드

우선 XE 최신버전을 다운로드 받은 후, 압축을 풀어서 XE가 설치된 디렉토리에 덮어씌웁니다.

> 주의!
> 만약, XE소스코드나 플러그인 소스코드를 수정했다면 미리 백업해두시길 바랍니다.

업데이트 적용

아래 콘솔 커맨드를 사용하여 업데이트된 소스코드를 적용합니다.

```
php artisan xe:update --skip-download
```

실제 구동화면입니다.

```
$ php artisan xe:update --skip-download

 Update Xpressengine.

 Update version information:
   3.0.0-beta -> 3.0.0-beta.2

 Update the Xpressengine ver.3.0.0-beta. There is a maximum moisture is applied.
Do you want to update? (yes/no) [no]:
 > yes

Run the composer update. There is a maximum moisture is applied.
------------------------------------------------------------------

 composer update
 ...
 ...

 Run the migration of the Xpressengine.
--------------------------------------

 updating category.. [skipped]
 updating config.. [skipped]
 updating counter.. [skipped]
 updating document.. [skipped]
 updating dynamicField.. [skipped]
 updating editor.. [skipped]
 updating media.. [skipped]
 updating menu.. [skipped]
 updating permission.. [skipped]
 updating plugin.. [skipped]
 updating routing.. [skipped]
 updating settings.. [skipped]
 updating site.. [skipped]
 updating skin.. [skipped]
 updating storage.. [skipped]
 updating tag.. [skipped]
 updating temporary.. [skipped]
 updating theme.. [skipped]
 updating translation.. [success]
 updating update.. [skipped]
 updating user.. [skipped]


 [OK] Update the Xpressengine to ver.3.0.0-beta.2
```

설치를 완료한 후에는 `사이트 관리자 > 플러그인 > 플러그인 목록` 에서 플러그인들도 업데이트하시기 바랍니다.

config 디렉토리

`conifg` 디렉토리에는 XE에서 필요한 다양한 설정 정보가 포함된 파일들이 위치합니다. XE는 사용자들이 XE 코어 업데이트로부터 자유로워질 수 있도록 Cascading 방식의 `config` 확장 기능을 지원합니다.

Cascading config

이것은 사용자의 XE에 지정된 환경변수에 따라 서로 다른 설정을 사용할 수 있도록 하는 방법중 하나입니다. 이 방식은 지정된 환경변수와 같은 이름의 `config` 디렉토리 내 하위 디렉토리에 정의된 설정 값을 사용하도록 하고 있습니다.

```
config/
├── production
│   ├── app.php
│   ├── database.php
│   └── mail.php
├── app.php
├── auth.php
├── database.php
...
└── xe.php
```

`config` 디렉토리는 위와 같은 구조를 가지고 있습니다. 같은 이름을 가지는 `config/app.php` 와 `config/production/app.php` 파일을 열어보면 아래와 같은 내용을 확인할 수 있습니다.

```
// in config/app.php
return [
    'debug' => env('APP_DEBUG', false),
    ...
```

```
// in config/production/app.php
return [
    'debug' => false,
    ...
```

`config/app.php` 파일에서는 `env` 함수를 사용하여 `debug` 설정을 하고 있습니다. 환경변수 `APP_DEBUG` 가 정의되어 있는 경우 해당 값을, 그렇지 않은경우 `false` 가 지정되도록 되어있습니다. 그리고 `config/production/app.php` 파일에서는 `debug` 를 `false` 로 설정하고 있습니다.

이 상황에서 만약 사용자의 환경이 `production` 이고 `config/production/app.php` 의 `debug` 값을 `true` 로 지정했다면, XE는 `config/app.php` 의 설정을 무시하고 `true` 를 `debug` 설정으로 사용합니다.

환경변수의 지정

환경변수는 XE 루트 디렉토리에 위치하고 있는 `.env` 파일에서 지정할 수 있습니다.

```
// in .env file
APP_ENV=production
```

만약 사용자가 `local` 개발환경에 대한 설정을 별도로 지정하여 사용하고 싶은 경우, `.env` 파일의 `APP_ENV` 항목에 `local` 로 지정하고, config 디렉토리 내에 `local` 디렉토리를 생성하여 원하는 설정을 해당 디렉토리에 위치시키면 됩니다.

> XE를 처음 설치할 때, 환경변수는 `production` 으로 지정됩니다.

`.env` 에 대한 내용은 다음 문서에서 자세히 확인하실 수 있습니다.

config caching

`config` 파일에는 `env` , `storage_path` 와 같은 함수들을 사용하여 값을 지정하고 있습니다. 이는 config 로 부터 값을 얻고자 할 때, 해당 함수가 실행되어야 함을 의미합니다. 그래서 XE에서는 좀 더 빠르게 설정값을 불러올 수 있도록 cache 파일을 생성할 수 있는 커맨드를 제공하고 있습니다.

```
$ php artisan config:cache
```

콘솔에서 위 커맨드를 실행하면 `config` 디렉토리 내에 설정된 모든 값이 cache 파일로 생성되어집니다.

이미 생성된 config cache 파일을 제거하고 싶은 경우 아래 커맨드를 통해 제거할 수 있습니다.

```
$ php artisan config:clear
```

.env

.env 파일은 웹 애플리케이션이 위치한 곳이 어떤 환경인지를 지정하고 현재 환경에서 사용되어질 환경변수들이 작성된 파일입니다.

.env 파일은 다음과 같이 작성되어 집니다.

```
APP_ENV=local
APP_KEY=SomeRandomString
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
...
```

위 내용에서 `APP_ENV` 는 현재 환경을 가리키는 항목이고 기타 나머지는 config 에서 사용되어질 설정값들입니다.

환경변수 추가하기

.env 에서 정의되는 환경변수는 php 파일내에 `env` 함수를 사용하여 작성되어져야 합니다.

```
  'mysql' => [
      'driver'    => 'mysql',
      'host'      => env('DB_HOST', 'localhost'),
      'database'  => env('DB_DATABASE', 'forge'),
      'username'  => env('DB_USERNAME', 'forge'),
      'password'  => env('DB_PASSWORD', ''),
      'port'      => env('DB_PORT', '3306'),
      'charset'   => 'utf8',
      'collation' => 'utf8_unicode_ci',
      'prefix' => 'xe_',
      'strict'    => false,
  ],
```

위 설정에 있는 `host` , `database` , `username` , `password` , `port` 는 .env 파일에 작성되어 값을 지정할 수 있지만, 나머지 `charset` , `prefix` 등은 .env 파일을 통해 지정할 수 없습니다.

만약 기존에 존재하는 설정이외에 추가로 설정이 추가되고, 이 설정들이 .env 파일에 의해 값이 지정되길 원한다면, 해당 설정 값을 `env` 함수를 이용해 작성하면 됩니다.

```
    'new_setting' => env('NEW_SETTING', '');
```

그리고 .env 파일에 해당 값을 지정해 주면 됩니다.

```
NEW_SETTING=MySetting
```

> XE는 설치시 .env 파일이 존재하지 않습니다. 현재 시스템의 환경을 지정하거나 환경변수를 추가하고자 하는 경우 웹 애플리케이션 root 위치에 .env 파일을 생성하여 사용할 수 있습니다.

디버깅

XE를 운영하거나 개발하는 과정에서 오류가 발생하는 경우, 발생한 오류에 대한 자세한 정보를 필요로 합니다.

디버깅 모드 활성화

XE를 디버깅모드로 설정하면 오류가 발생한 경우, 브라우저에 오류의 자세한 정보가 바로 출력됩니다. 디버그 모드를 활성화 하기 위해서는 `config/production/app.php` 파일의 `debug` 값을 `true` 로 설정해야 합니다.

현재 환경에 맞는 config 파일을 열어 값을 변경합니다.

```
//in config/production/app.php
  ...
  'debug' => true,
  ...
```

`.env` 파일에 `debug` 모드를 지정할 수도 있습니다.

```
// in config/production/app.php
  ...
  'debug' => env('APP_DEBUG', true)
  ...
```

위와 같이 `config/production/app.php` 파일을 설정하고, XE의 루트디렉토리의 `.env` 파일에는 아래와 같이 작성합니다.

```
# in .env file
APP_DEBUG=true
```

> **주의!**
> 실제 서비스가 운영되는 환경에서는 디버깅 모드를 활성화하지 마십시오. 디버깅 모드를 활성화 해 놓으면, 일반 사용자도 오류가 발생할 경우, 브라우저에서 오류의 자세한 내용을 볼 수 있으므로, 보안상 문제가 될 수 있습니다.

변수값 확인하기(dump)

디버깅을 위해 특정시점에서 변수의 값이나 함수의 반환값 등을 확인할 필요가 있습니다. XE는 변수의 값을 브라우저에 곧바로 출력하거나 로그 파일에 기록해 놓을 수 있는 기능을 제공합니다.

브라우저에 출력하기

XE는 `dump` 와 `dd` 함수를 제공하고 있습니다. `dump` 는 php 의 내장함수인 `var_dump` 와 유사하지만 브라우저 상에서 좀 더 보기 좋은 형태로 표현해 줍니다. `dd` 는 dump & die 의 약어로 dump 처리 후 라이프사이클을 중단시킵니다.

```
dump($var);
dd($var);
```

여러가지 값을 동시에 확인하고자 한다면 해당 변수들을 인자로 나열하면 됩니다.

```
dump($foo, $bar, $baz);
dd($foo, $bar, $baz);
```

로그파일에 기록하기

Log 파사드를 이용해 로그 파일에 내용을 기록할 수 있습니다. 로그 파일의 위치는 `storage/log/laravel.log` 입니다.

```
$var = 'a';
$bar = 'b';
Log::info($var . ' ' . $bar); // [0000-00-00 00:00:00] production.INFO: a b
```

> log 파일에는 사용자의 호출에 의한 기록 이외에도 장애시 발생한 오류 정보 등이 기록되어 있습니다.

디렉토리 구조

루트 디렉토리에서 중요한 항목만 나열해 보았습니다. XE는 라라벨 프레임워크의 구조를 기반으로하여 커스터마이징되어 있습니다.

```
├── app/
├── assets/
├── bootstrap/
├── config/
├── core/
├── migrations/
├── plugins/
├── resources/
├── storage/
├── vendor/
├── web_installer/
├── artisan
├── composer.json
├── composer.user.json.example
└── index.php
```

루트 디렉토리

app

XE를 구성하는 컨트롤러나 커맨드와 같은 어플리케이션 레벨에 해당하는 코드가 들어있습니다.

core

XE의 핵심적인 서비스 및 컴포넌트 관리 기능을 위한 클래스 파일의 모음입니다.

migrations

XE를 설치하거나 업데이트할 때, 실행되는 코드가 들어있습니다.

config

XE의 설정을 저장하는 파일들이 들어있습니다.

assets

웹페이지에서 필요한 stylesheet, javascript나 이미지 파일과 같은 assets 파일들이 들어있습니다.

plugins

서드파티에서 작성한 플러그인이 설치되는 디렉토리입니다.

resources

XE에서 필요한 템플릿 파일이나 언어파일이 들어있습니다.

storage

이 디렉토리에는 XE가 실행되면서 생성되는 동적인 파일들이 저장되는 위치입니다. 회원 프로필 사진이나 게시판 첨부 파일, 그외 캐시 파일 등이 이 디렉토리에 생성됩니다.

vendor

composer를 통해 설치되는 외부 라이브러리들이 설치되는 디렉토리입니다.

web_installer

XE를 웹을 통해 설치할 때 필요한 파일들이 들어있습니다.

artisan

터미널에서 XE의 명령을 실행할 때 사용하는 파일입니다.

composer.json

XE는 PHP 패키지 관리툴인 composer를 사용합니다. 이 파일은 composer 툴에서 사용하는 설정 파일입니다.

composer.user.json.example

XE에서는 `composer.json` 파일을 직접 수정하는 것을 금지합니다. `composer.json` 파일은 XE의 소스코드에 포함된 파일이므로 XE를 업데이트할 때 덮어씌워집니다. `composer.json`을 수정하는 대신 `composer.user.json` 파일을 사용하여 패키지를 관리할 수 있습니다. `composer.user.json.example` 파일 이름을 `composer.user.json`로 바꿔서 사용하시면 됩니다.

index.php

웹브라우저를 통해 사이트에 접근할 때, 항상 실행되는 PHP파일입니다.

컴포넌트

컴포넌트는 **플러그인을 통해 등록할 수 있는 XE의 구성요소**를 통틀어 지칭하는 용어입니다. XE에는 여러가지 타입의 컴포넌트가 있으며 테마, 스킨, 에디터, 위젯 등이 대표적입니다. 각 타입의 컴포넌트들 간에 별다른 공통점은 없습니다. 오직 플러그인 개발자에 의해 등록됨으로써 XE에 추가될 수 있다는 점이 유일한 공통점입니다.

플러그인 개발자들은 컴포넌트를 제작하여 XE에 등록할 수 있습니다. 만약 자신만의 컴포넌트를 만들어 사용하고 싶다면 먼저 플러그인을 하나 생성한 다음 플러그인에 컴포넌트를 추가해 넣습니다. 그 다음, 플러그인을 활성화하면 자신이 제작한 컴포넌트를 XE에서 사용할 수 있습니다.

예를 들어 새로운 테마를 만들어서 자신의 사이트에 적용하고 싶다면, 먼저 빈 플러그인을 하나 생성합니다. 그 다음 플러그인 안에서 테마 컴포넌트를 만듭니다. 만약 테마와 함께 게시판 스킨과 위젯도 만들어서 사이트에 적용하고 싶다면, 이 두가지 컴포넌트도 같은 플러그인에 추가합니다. 플러그인을 활성화하면 플러그인에 포함된 컴포넌트들을 XE에서 모두 사용할 수 있습니다.

> XE1에서는 각 컴포넌트를 따로 XE에 추가해야 했습니다. XE3에서는 하나의 플러그인에 컴포넌트를 모두 포함하여 한번에 추가할 수 있습니다. 배포할 때에도 마찬가지로 각 컴포넌트를 따로 배포하지 않고, 플러그인 하나에 담아서 배포하면 됩니다.

컴포넌트 인터페이스

모든 컴포넌트는 `Xpressengine\Plugin\ComponentInterface` 인터페이스를 구현한 클래스 형태여야 합니다. 이 인터페이스는 컴포넌트를 XE에 등록할 때 필요한 규칙을 한정해주는 역할을 합니다.

컴포넌트 타입

XE는 기본적으로 10개 타입의 컴포넌트를 가지고 있습니다. 각 타입의 컴포넌트는 모두 `\Xpressengine\Plugin\ComponentInterface`를 구현한 추상클래스를 정의하고 있습니다. 여러분이 어떤 타입의 컴포넌트를 제작하고 싶다면, 그 타입의 추상클래스를 상속받는 클래스를 작성하면 됩니다.

만약, 테마 컴포넌트를 제작하려고 한다면 테마 타입의 추상클래스인 `\Xpressengine\Theme\AbstractTheme`를 상속받는 클래스를 작성하십시오.

```
\Xpressengine\Plugin\ComponentInterface
  ├── 테마 - \Xpressengine\Theme\AbstractTheme
  ├── 모듈 - \Xpressengine\Module\AbstractModule
  ├── 스킨 - \Xpressengine\Skin\AbstractSkin
  ├── 다이나믹필드 - \Xpressengine\DynamicField\AbstractType
  ├── 다이나믹필드 스킨 - \Xpressengine\DynamicField\AbstractSkin
  ├── 토글메뉴 - \Xpressengine\ToggleMenu\AbstractToggleMenu
  ├── UI오브젝트 - \Xpressengine\UIObject\AbstractUIObject
  ├── 위젯 - \Xpressengine\Widget\AbstractWidget
  ├── 에디터 - \Xpressengine\Editor\AbstractEditor
  └── 에디터툴 - \Xpressengine\Editor\AbstractTool
```

플러그인 개발자는 10개의 타입 이외에 플러그인에서 필요한 타입을 직접 생성하여 사용할 수도 있습니다. 플러그인에서 필요한 컴포넌트 타입을 정의해 놓으면, 다른 플러그인 개발자들이 정의한 타입의 컴포넌트를 제작하여 등록할 수 있습니다.

11개의 XE의 기본 컴포넌트 타입에 대해 간략히 설명해 보겠습니다.

테마

테마는 웹사이트의 디자인과 웹페이지의 전체적인 레이아웃을 결정합니다.

모든 웹사이트는 요청받은 URL에 해당하는 메인 콘텐츠를 출력합니다. 예를 들어 특정 포스트를 출력하는 웹페이지라면, 포스트의 제목, 내용, 댓글목록이 메인 콘텐츠에 해당됩니다. XE는 출력할 메인 콘텐츠를 작성한 후 웹브라우저로 보내기 전에 지정된 테마에게 메인 콘텐츠를 전달합니다. 테마는 웹사이트에서 공통적으로 필요한 헤더, 푸터, 사이드바 영역과 전달받은 메인 콘텐츠 영역을 조립하여, 완성된 웹페이지를 만듭니다. 이때 테마는 웹페이지에 디자인을 결정짓는 마크업을 생성하고, 스타일시트를 로드합니다.

테마 컴포넌트의 제작법은 이 문서를 참조하세요.

스킨

스킨의 목적은 출력되는 웹브라우저에서 출력되는 모든 html을 교체 가능하도록 만들기 위함입니다. 가령, 게시판 플러그인은 글 목록 보기, 글 보기 등의 페이지를 제공합니다. 만약 게시판 플러그인에 스킨을 적용하지 않았다면, 게시판 플러그인의 제작자가 처음 제작한 페이지 디자인을 바꿀 수 없습니다. 하지만 게시판에 스킨 시스템이 적용되어 있다면 사이트 관리자는 다른 개발자가 제작한 게시판 스킨을 선택하고, 사용할 수 있습니다. 자신이 만든 플러그인의 디자인을 사이트 관리자가 교체할 수 있도록 하려면 스킨 시스템을 적용하십시오.

스킨 컴포넌트의 제작법은 이 문서를 참조하세요.

다이나믹 필드

다이나믹 필드는 사용자가 관리자에서 입력필드를 추가해 사용할 수 있도록 해줍니다.

이는 코어의 패키지나 개별 플러그인 별로 지원여부를 결정하며 회원 패키지, Board 플러그인에서 지원하고 있습니다. 다이나믹 필드를 이용해 회원 가입할 때 나이를 입력 받거나 게시물을 작성할 때 주소를 입력받을 수 있습니다.

다이나믹 필드 스킨

다이나믹 필드 스킨은 다이나믹 필드를 출력할 때 사용하는 스킨 입니다.

라디오 버튼을 제공하는 다이나믹 필드를 사용할 때 테마나 스킨에 따라 스타일을 변경할 수 있도록 합니다.

UI오브젝트

XE에서는 화면에 자주 출력되는 UI 요소들이 있습니다. 텍스트 인풋박스나 셀렉트 박스(select)와 같은 기본적인 폼 요소들이 있고, 테마 선택기나 메뉴 선택기, 또는 권한 설정 UI와 같이 XE에서만 사용되는 특별한 요소들도 있습니다. UI오브젝트는 이렇게 자주 사용되는 UI 요소를 개발자들이 쉽게 출력할 수 있는 방법을 제공합니다.

직접 UI요소의 마크업을 작성하는 대신, `uio()` 함수를 사용하여 UI오브젝트를 출력하십시오. 선택된 UI오브젝트는 알아서 html 마크업을 생성하여 출력하고, 스타일시트와 스크립트 파일도 자동으로 로드합니다.

위젯

위젯은 다양한 컨텐츠를 사이트의 어느 곳에서나 반복적으로 출력할 수 있는 방법을 제공합니다. 위젯은 출력할 컨텐츠에 대한 설정 정보를 사용자로부터 입력받고, 입력받은 설정 정보를 이용하여 컨텐츠를 생성합니다. 그 다음, 미리 정해진 템플릿을 통해 컨텐츠를 출력합니다. 사용자는 화면에 위젯을 추가하기 위해 위젯 생성 UI를 사용할 수 있습니다. 위젯 생성 UI에 추가할 위젯의 종류와 스킨을 선택하고, 위젯의 설정정보를 입력하면 XML 형식의 위젯코드가 생성됩니다. 이 위젯코드를 테마나 페이지의 원하는 위치에 삽입하면, XE는 최종적으로 위젯코드를 HTML로 변환하여 화면에 출력합니다.

모듈

...

에디터

...

에디터툴

...

토글메뉴

...

플러그인

플러그인은 여러분이 XE를 확장할 수 있는 유일한 방법입니다. XE에 새로운 기능을 추가하거나 기본 기능을 변경하려고 할 때, 또는 테마나 스킨과 같은 컴포넌트를 XE에 추가하고 싶을 때, 플러그인을 사용하십시오.

XE는 오픈소스프로그램으로 자유롭게 코어 소스코드를 수정하여 사용할수 있지만, 업데이트 되는 소스코드를 계속 적용하려면 소스코드의 수정을 피해야 합니다. 대신 플러그인을 사용하면 XE 소스코드의 수정을 피할 수 있습니다. 또한 플러그인을 다른 XE 사용자들과 공유할 수도 있습니다.

각 플러그인은 고유한 이름을 가지는 하나의 디렉토리로 구성되며, `/plugins` 디렉토리에 등록됩니다.

번들 플러그인

XE는 사용자들이 자주 사용할 만한 플러그인을 XE에 포함하여 배포하고 있습니다.

- alice
- google_analytics
- orientator
- page
- board
- comment
- ckeditor
- claim
- external_page
- social_login
- news_client

플러그인 상태

각 플러그인은 활성화 또는 비활성화 상태를 가집니다. 어떤 플러그인을 `/plugins` 디렉토리에 추가한다고 해도 처음에는 비활성화 상태이기 때문에 바로 작동되지 않습니다. 플러그인을 활성화(activate)시켜야 비로소 플러그인이 작동합니다. 플러그인을 활성화시키려면 사이트 관리자 > 플러그인 > 플러그인 목록에서 원하는 플러그인을 활성화시키십시오.

개발 모드 플러그인

XE 자료실을 통해 설치하지 않은 플러그인을 `개발 모드 플러그인` 이라고 합니다.

사이트관리자는 XE 자료실을 통해 다른 개발자들이 배포한 플러그인을 다운로드 받을 수 있습니다. 물론, 다른 개발자의 플러그인 소스코드를 `/plugins`에 직접 추가하여 사용할 수도 있으며, 여러분이 직접 생성한 플러그인을 추가하여 사용할 수도 있습니다.

XE 자료실을 통해 설치하지 않은 플러그인은 터미널에서 반드시 플러그인 디렉토리로 이동후 `composer update` 명령을 실행해주어야 합니다. 이 명령을 실행하면, 플러그인 디렉토리에 `vendor` 디렉토리가 생성됩니다. 따라서 개발모드 플러그인은 `vendor` 디렉토리를 가집니다.

> 직접 설치한 플러그인에서 `composer update`를 실행하지 않으면 autoload가 등록되지 않아 제대로 작동하지 않을 수 있습니다. 또 XE에서는 자료실을 통해 설치한 플러그인으로 인식하여 오작동을 일으킬 수 있습니다.

서비스

소개

XE가 앞서 설명한 라이프사이클에 따라 요청을 처리하고 응답하는 과정을 수행하는 동안, 우리는 매우 다양한 작업을 실행해야 합니다. 때로는 데이터베이스에서 자료를 검색하거나 자료를 저장하기도 하고, 이메일을 전송하거나 세션이나 쿠키를 다뤄야 할 때도 있습니다. 개발자들이 이런 작업을 수행하는 코드를 모두 직접 구현한다면 매우 힘든 개발이 될 것입니다.

XE는 개발자들이 원하는 기능을 쉽게 구현할 수 있도록 많은 서비스를 제공합니다. 아래 목록은 라라벨이 기본으로 제공하는 프레임워크 레벨의 주요 서비스 목록입니다.

- Auth
- Cache
- Config
- Console
- Container
- Cookie
- Database
- Encryption
- Events
- Filesystem
- Hashing
- Log
- Mail
- Pagination
- Queue
- Redis
- Routing
- Session
- Translation
- Validation
- View

XE는 라라벨이 제공하는 위 서비스 이외에도 문서(document), 파일(storage), 회원(user) 관리와 같은 CMS 어플리케이션 레벨의 서비스를 많이 제공합니다. XE에서 제공하는 서비스은 아래 목록과 같습니다.

- Captcha
- Category
- Config
- Counter
- Database
- Document
- DynamicField
- Editor
- Frontend
- Interception
- Keygen
- Media
- Menu
- Module
- Permission
- Plugin

- Presenter
- Register
- Routing
- Seo
- Settings
- Site
- Skin
- Storage
- Tag
- Temporary
- Theme
- ToggleMenu
- Translation
- Trash
- UIObject
- User
- Widget

각각의 서비스가 제공하는 기능과 용도를 충분히 알고 있다면 훨씬 편하게 플러그인을 개발할 수 있습니다. 본 매뉴얼의 서비스 카테고리에서 각 서비스의 기능과 사용법을 자세히 안내하고 있습니다.

서비스 로드하기

파사드를 사용하여 로드하기

내 코드에서 필요한 서비스를 로드하는 방법으로 XE는 파사드를 제공합니다. 파사드는 어떤 서비스를 대신하는 프록시 역할하며, Static 클래스 형식으로 사용할 수 있습니다.

```
// User 서비스 사용
$user = \XeUser::find($userId);
```

위 예제에서 `XeUser` 는 파사드입니다. 이 파사드를 통해 User 서비스를 사용할 수 있습니다. 위 코드와 같이 파사드의 `find` 메소드를 호출하면 실제로는 `\Xpressengine\User\UserHandler` 클래스의 인스턴스의 `find` 메소드가 호출됩니다.

서비스컨테이너를 사용하여 로드하기

파사드를 사용하는 대신, 서비스 컨테이너로부터 직접 서비스를 로드할 수 있습니다. XE에서 제공하는 모든 서비스는 '서비스 컨테이너'에 등록됩니다. 내 코드에서 어떤 서비스를 로드해야할 때, 서비스 컨테이너에게 그 서비스를 달라고 요청하면, 서비스 컨테이너는 요청받은 서비스를 반환해 줍니다. 보통 `app()` 함수나 `App` 파사드를 사용하여 서비스 컨테이너로부터 필요한 서비스를 로드합니다.

```
// User 서비스 로드
$userHandler = app('xe.user');

// or
$userHandler = \App::make('xe.user');

// User 서비스 사용
$user = $userHandler->find($userId);
```

보통 서비스는 특정 클래스의 싱글톤 인스턴스입니다. 그 클래스는 개발자가 서비스를 이용할 때 편하게 사용할 수 있는 메소드(인터페이스)를 가지고 있습니다.

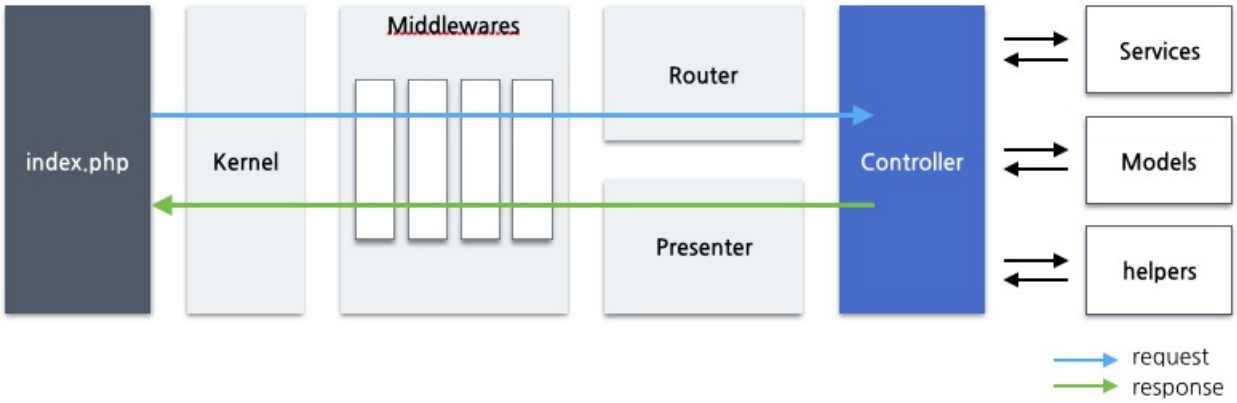
사실, 서비스 컨테이너는 서비스 관리에 국한된 역할을 하는 것은 아닙니다. 정확히 말하면, 서비스 컨테이너는 다양한 클래스의 의존성을 관리하는 강력한 도구 입니다. 서비스 컨테이너에 대하여 더 알고싶다면, 이 문서를 참고하세요.

라이프 사이클

여러분이 XE를 사용하거나 XE의 플러그인을 개발하려고 한다면, XE 사용법, 플러그인 제작법, 그리고 XE에서 제공하는 여러가지 서비스의 사용법을 숙지하는 것 만으로도 충분히 많은 것을 할 수 있습니다. 하지만 여러분이 좀 더 고도의 기능을 필요로 하는 플러그인을 만들거나 XE를 제대로 사용하고 싶다면, XE가 어떤 방식과 구조로 요청을 처리하는지에 대한 전체적인 흐름을 알고 있어야 합니다. 전체적인 흐름을 알고 있지 않다면 결국 한계에 도달할 것이고, 전체적인 흐름을 알기 위해 노력하는 시점이 올 것입니다.

XE의 라이프 사이클은 크게 두 가지 경우로 나눌 수 있습니다. 사용자들의 웹브라우저로 부터 http 요청을 받았을 때, 이를 처리하고 응답하는 일반적인 경우와, 사이트 관리자가 ssh와 같은 콘솔에 접근하여 php 명령을 실행시킬 경우가 있습니다. 이 문서에서는 더 일반적으로 생각할 수 있는 http 요청을 처리하는 경우에 대하여 살펴보겠습니다.

아래 다이어그램은 XE의 요청 처리 흐름을 개략적으로 보여줍니다.



index.php

사용자의 웹브라우저로부터 http 전송 요청이 들어올 경우, XE는 항상 `index.php` 파일을 실행시킵니다. `index.php` 파일이 그리 많은 코드를 가지고 있는 것은 아닙니다.

가장 먼저 `index.php` 는 composer를 통해 생성된 autoload 파일을 로드합니다. autoload 파일을 로드함으로써 XE는 php 파일을 include하지 않고 자동으로 로드할 수 있게됩니다.

그 다음으로 서비스 컨테이너를 생성합니다. 서비스 컨테이너는 생성되자마자 주요 서비스인 라우팅(routing) 서비스와 이벤트(events) 서비스를 등록합니다.

세번째로는 http 요청을 처리하기 위한 Http 커널을 생성합니다. 그리고 현재 http 요청에 대한 정보를 가지는 `Request` 인스턴스를 생성합니다.

마지막으로 `Request` 인스턴스를 커널에게 전달하여 http 요청의 본격적인 처리를 시작합니다.

Http 커널

Http 커널의 주목적은 단순하게 `Request` 를 처리하고, 브라우저로 돌려줄 응답(Http Response)를 만드는 것입니다.

Http 커널은 `Illuminate\Foundation\Http\Kernel` 를 상속받고 있으며, 생성된 다음에는 `Request` 를 처리할 준비, 즉 부팅(bootstrapping)을 합니다.

Http 커널은 부팅 과정에서 에러 처리, 로그 설정, 어플리케이션의 실행 환경의 검사 등 실제로 요청이 처리되기 전에 수행해야 되는 작업들을 합니다.

또, Http 커널은 부팅 과정에서 데이터베이스, 문서, 회원과 같은 XE에서 제공하는 대부분의 서비스를 앞서 생성된 서비스 컨테이너에 등록합니다.

> **NOTE:** XE에서 활성화되어 있는 플러그인들이 부팅되는 시점은 서비스들이 등록된 바로 다음입니다.

Http 미들웨어

Http 커널은 `Request` 처리할 준비가 완료되면, 커널에 등록된 미들웨어들에게 `Request` 를 전달합니다. `Request` 는 미들웨어들을 거친후 컨트롤러에게 전달됩니다.

Http 미들웨어는 글로벌 미들웨어(global middleware)와 라우트 미들웨어(route middleware)로 구분할 수 있습니다. 글로벌 미들웨어는 모든 요청에 대해 항상 작동하는 미들웨어이며, 라우트 미들웨어는 특정 라우트에 등록된 미들웨어입니다. `Request` 에 해당하는 라우트가 선택되면 그 라우트에 등록된 미들웨어들이 작동합니다.

미들웨어에 대한 자세한 정보는 라라벨 문서를 참고하시기 바랍니다.

라우터

미들웨어를 통과한 `Request` 는 라우터에게 전달됩니다. 라우터는 전달받은 `Request` 를 전담하여 처리할 컨트롤러가 누군지 찾고, 찾은 컨트롤러를 호출합니다.

컨트롤러

각각의 `Request` 는 모두 XE가 해주길 바라는 정확한 목적을 가지고 있습니다. 예를 들어, 어떤 요청은 게시판 글 출력이 목적일 수도 있고, 또 어떤 요청은 현재 사용자를 로그아웃 처리해 달라는 게 목적일 수도 있습니다. 컨트롤러는 `Request` 의 주목적을 처리하는 역할을 합니다. XE에는 매우 많은 컨트롤러가 이미 포함돼 있고, 플러그인을 통해서도 많이 추가될 수 있습니다.

컨트롤러는 `Request` 를 처리하기 위해 여러가지 서비스나 모델 그리고 헬퍼함수들을 가져다 사용합니다.

프리젠터

컨트롤러가 `Request` 를 처리하고 나면 `Response` 를 생성하기 위해 프리젠터를 호출합니다. 이 때 만약 화면 출력이 필요하다면 화면 출력에 필요한 데이터와 출력할 때 사용할 스킨정보를 프리젠터에게 전달합니다.

프리젠터는 컨트롤러가 전달한 데이터와 스킨정보, 그리고 지정된 테마 정보등을 조합해서 완성된 html 문서를 조립합니다. 이렇게 완성된 html 문서는 `Response` 인스턴스가 되어 다시 미들웨어를 거친후 브라우저로 전송됩니다.

프리젠터는 html 타입의 `Response` 뿐만 아니라, ajax 요청에 대한 응답과 같이 json 방식의 `Response` 도 처리합니다.

라우팅(routing)

라우터는 `Request` 의 URI를 판단하여 `Request` 를 처리할 담당 컨트롤러를 찾는 역할을 합니다. 이 장에서는 라우트를 정의하는 방법에 대하여 설명합니다.

기본적인 라우팅

이미 XE에는 매우 많은 라우트가 `app/Http/routes.php` 파일안에 정의되어 있습니다. 플러그인을 개발할 때에는 각 플러그인 클래스의 `boot` 메소드에 라우트 정의 코드를 작성하십시오. 플러그인이 boot될 때 라우트가 등록됩니다.

가장 기본적인 라우트는 URI와 `Closure` 하나로 지정할 수 있습니다.

기본적인 라우트

라우트를 등록할 때에는 `Route` 파사드를 사용합니다.

```
// GET Http 메소드로 요청될 경우 'Hello World'를 화면에 출력
Route::get('/', function()
{
    return 'Hello World';
});

Route::post('foo/bar', function()
{
    return 'Hello World';
});

Route::put('foo/bar', function()
{
    //
});

Route::delete('foo/bar', function()
{
    //
});
```

`Closure` 대신 컨트롤러를 사용할 수도 있습니다.

```
Route::get('user/profile', 'UserController@showProfile');
```

여러 HTTP 메소드에 라우트 등록하기

```
Route::match(['get', 'post'], '/', function()
{
    return 'Hello World';
});
```

`any` 메소드를 사용하면 모든 http 메소드에 응답하는 라우트를 등록할 수도 있습니다.

```
Route::any('foo', function()
{
    return 'Hello World';
});
```

라우트에 등록된 URL을 생성하려면 `url` 헬퍼함수를 사용하면 됩니다:

```
$url = url('foo');
```

라우트 파라미터

라우트에서 요청된 URI 세그먼트를 얻을 수 있습니다:

기본적인 라우트 파라미터

```
Route::get('user/{id}', function($id)
{
    return 'User '.$id;
});
```

> **주의:** 라우트 파라미터는 `-` 문자를 포함하면 안됩니다. (`_`)를 사용하십시오.

선택적인 라우트 파라미터

```
Route::get('user/{name?}', function($name = null)
{
    return $name;
});

Route::get('user/{name?}', function($name = 'John')
{
    return $name;
});
```

`name` 파라미터는 옵션입니다. `name` 파라미터가 URL에 포함되어 있지 않아도 위 라우트가 작동됩니다.

정규표현식으로 파라미터 제약하기

```
Route::get('user/{name}', function($name)
{
    //
})
->where('name', '[A-Za-z]+');
```

```
Route::get('user/{id}', function($id)
{
    //
})
->where('id', '[0-9]+');

Route::get('user/{id}/{name}', function($id, $name)
{
    //
})
->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

이름이 지정된 라우트

이름이 지정된 라우트는 지정된 라우트에 대한 URL을 생성하거나 Redirect를 할 때 편리함을 제공합니다. `as` 배열 키를 통해 라우트에 이름을 지정할 수 있습니다.

```
Route::get('user/profile', ['as' => 'profile', function()
{
    //
}]);
```

컨트롤러 액션에 대해서도 라우트 이름을 지정할 수 있습니다.

```
Route::get('user/profile', [
    'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

이제 URL을 생성하거나 Redirect를 하는 데 라우트 이름을 사용할 수 있습니다.

```
$url = route('profile');

$redirect = redirect()->route('profile');
```

라우트가 파라미터를 가지고 있다면, `route` 함수의 두번째 인자로 파라미터를 전달할 수 있습니다. 주어진 파라미터는 자동으로 URL에 추가됩니다.

```
Route::get('user/{id}/profile', ['as' => 'profile', function ($id) {
    //
}]);

$url = route('profile', ['id' => 1]);
```

라우트 그룹

때때로 많은 라우트들이 URL 세그먼트, 미들웨어, 네임스페이스 등과 같은 공통의 요구사항을 공유하고자 하는 경우가 있습니다. 이러한 옵션들을 모든 라우트에 개별로 각각 지정하는 대신에 라우트 그룹을 통해서 다수의 라우트에 속성을 지정할 수가 있습니다.

속성값들을 공유하는 것은 `Route::group` 메소드의 첫 번째 인자로 배열을 지정하면 됩니다.

라우트 미들웨어

라우트 그룹에 지정하는 배열의 `middleware` 값에 미들웨어의 목록을 정의함으로써 그룹 내의 모든 라우트에 미들웨어가 적용됩니다. 라우트 미들웨어는 배열에 정의된 순서대로 실행될것입니다:

> 주의! 라우트 미들웨어는 Http 커널의 미들웨어와는 다른 별개의 기능입니다.

```
Route::group(['middleware' => ['foo','bar']], function()
{
    Route::get('/', function()
    {
        // Has Foo And Bar Middleware
    });

    Route::get('user/profile', function()
    {
        // Has Foo And Bar Middleware
    });
});
```

네임스페이스

그룹의 속성 배열에 `namespace` 파라미터를 사용하여 가룹의 모든 컨트롤러에 네임스페이스를 지정할 수 있습니다:

```
Route::group(['namespace' => 'Admin'], function()
{
    // Controllers Within The "App\Http\Controllers\Admin" Namespace

    Route::group(['namespace' => 'User'], function()
    {
        // Controllers Within The "App\Http\Controllers\Admin\User" Namespace
    });
});
```

> **참고:** 기본적으로 `RouteServiceProvider`에서 포함하고 있는 `routes.php` 파일에는 라우트 컨트롤들을 위해서 네임스페이스가 지정되어 있습니다. 따라서 `App\Http\Controllers`의 전체 네임스페이스를 따로 지정할 필요는 없습니다.

라우트 접두어 지정하기

라우트 그룹의 접두어(prefix)는 그룹의 속성 배열에 `prefix` 옵션을 사용하여 지정합니다:

```
Route::group(['prefix' => 'admin'], function()
{
    Route::get('users', function()
    {
        // Matches The "/admin/users" URL
    });
});
```

또한, `prefix` 파라미터를 라우트들의 공통 파라미터로 지정할 수 있습니다:

라우트 prefix 안에서 URL 파라미터 등록하기

```
Route::group(['prefix' => 'accounts/{account_id}'], function()
{
    Route::get('detail', function($account_id)
    {
        //
    });
});
```

또한, 지정된 파라미터 변수의 제약 사항을 정의할 수도 있습니다:

```
Route::group([
    'prefix' => 'accounts/{account_id}',
    'where' => ['account_id' => '[0-9]+'],
], function() {

    // Define Routes Here
});
```

Fixed 라우트

각 플러그인들은 자유롭게 라우트를 추가하여 사용할 수 있습니다. 하지만 서로 다른 플러그인들이 동일한 규칙의 라우트를 등록하면 문제가 발생할 수 있습니다. XE는 플러그인간 라우트의 충돌을 방지하기 위하여 각 플러그인에게 별도의 라우트 공간(url 세그먼트)을 할당합니다.

각 플러그인에게 할당된 라우트 규칙을 사용하려면 `Route::fixed` 메소드를 사용하십시오.

```
Route::fixed(<plugin_id>, function() {

    // Define Routes Here
    Route::get('/', ...);

});
```

`Route::fixed` 메소드의 첫번째 파라미터는 플러그인의 아이디를 지정하면 됩니다. `Route::fixed` 를 사용할 경우, 접두어(prefix)가 `/plugin/<plugin_id>` 으로 자동으로 지정됩니다. 따라서 위의 코드는 아래의 `Route::group` 을 사용한 코드와 동일합니다.

```
Route::group(['prefix'=>'plugin/<plugin_id>'], function() {

    // Define Routes Here
    Route::get('/', ...);

});
```

하지만, 접두어에 자동으로 추가되는 `plugin` 은 사이트 관리자가 설정에서 변경할 수 있는 값이므로 반드시 `Route::fixed` 를 사용하십시오.

CSRF 보호하기

XE에서는 크로스 사이트 요청 위조(cross-site request forgeries)으로부터 응용 프로그램을 쉽게 보호할 수 있습니다. 크로스 사이트 요청 위조는 악의적인 공격의 하나이며 인증받은 사용자를 대신하여 허가 받지 않은 명령을 수행합니다.

XE는 사용자별 CSRF "토큰"을 자동으로 생성합니다. 이 토큰은 인증된 사용자가 실제로 XE에 요청을 보내고 있는지 식별하는 데 사용됩니다.

Form에 CSRF 토큰 삽입하기

```
<input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
```

다음처럼 Blade 템플릿에서 사용할 수 있습니다.

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

일일이 수동으로 POST, PUT 또는 DELETE 요청에 대한 CSRF 토큰을 확인할 필요가 없습니다. XE가 자동으로 요청중인 토큰을 세션에 저장되어 있는 토큰과 일치하는지 확인할 것입니다.

메소드 Spoofing-속이기

HTML form은 실제로 `PUT` , `PATCH` 와 `DELETE` 액션을 지원하지 않습니다. 따라서 `PUT` , `PATCH` 이나 `DELETE` 로 지정된 라우트를 호출하는 HTML form을 정의한다면 `_method` 의 숨겨진 필드를 지정해야 합니다.

`_method` 필드로 보내진 값은 HTTP 요청 메소드를 구분하는 데 사용됩니다. 다음 예를 참조하십시오:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
</form>
```

Http 컨트롤러(Http Controllers)

플러그인은 플러그인 클래스의 `boot` 메소드를 통해 라우트를 등록할 수 있습니다. 라우트를 등록할 때, 특정 URL의 요청을 처리할 로직을 클로저로 작성하는 대신, 별도의 컨트롤러 클래스에 작성할 수 있습니다. 성격이 비슷한 요청을 처리하기 위한 컨트롤러 클래스를 정의하십시오.

컨트롤러는 `App\Http\Controllers\Controller` 클래스를 상속받아 작성하십시오.

기본 컨트롤러

다음은 기본적인 컨트롤러 클래스의 예제입니다:

```
<?php namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class UserController extends Controller {

    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }

}
```

다음과 같이 컨트롤러의 액션에 라우트를 지정할 수 있습니다:

```
Route::get('user/{id}', 'App\Http\UserController@showProfile');
```

컨트롤러 & 네임스페이스

컨트롤러의 네임스페이스를 지정할 때에는 반드시 전체 네임스페이스를 다 써주어야 합니다.

```
Route::get('foo', 'Photos\AdminController@method');
```

`namespace`를 사용하면 반복되는 namespace를 생략할 수 있습니다.

```
Route::group(['prefix' => 'photos', 'namespace' => 'Photos'], function()
{
    Route::get('admin', 'AdminController@method');
});
```

이름이 지정된 컨트롤러 라우트

클로저 라우트와 같이 컨트롤러 라우트에 이름을 지정할 수 있습니다.

```
Route::get('foo', ['uses' => 'FooController@method', 'as' => 'name']);
```

컨트롤러 액션의 URL 구하기

컨트롤러 액션에 대한 URL을 생성하기 위해서 `action` 헬퍼함수를 사용합니다:

```
$url = action('App\Http\Controllers\FooController@method');
```

단순히 컨트롤러의 전체 네임스페이스의 대신 클래스명만으로 URL을 생성하고 싶은 경우에는 root 컨트롤러 네임스페이스를 URL 제너레이터에 등록하면 됩니다:

```
URL::setRootControllerNamespace('App\Http\Controllers');

$url = action('FooController@method');
```

실행중인 컨트롤러 액션의 이름을 찾고자 한다면 `currentRouteAction` 메소드를 사용하면 됩니다:

```
$action = Route::currentRouteAction();
```

컨트롤러 미들웨어

미들웨어는 다음과 같이 컨트롤러 라우트에 지정합니다.

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

덧붙여 미들웨어를 컨트롤러의 생성자에서 지정할수도 있습니다.

```
class UserController extends Controller {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }
}
```

묵시적 컨트롤러

XE에서는 한번의 라우팅 등록으로 컨트롤러를 통해 모든 액션들을 처리할 수 있는 손쉬운 방법을 제공합니다. 먼저 `Route::controller` 메소드를 사용하여 경로를 지정합니다:

```
Route::controller('users', 'UserController');
```

`controller` 메소드는 두 개의 인자를 넘겨 받도록 되어 있습니다. 첫 번째 인자는 컨트롤러로 제어할 URI이고, 두 번째는 컨트롤러의 클래스명을 의미합니다. 이어서 해당하는 HTTP 메소드 이름을 접두어로 (get, post..) 사용하는 형태로 컨트롤러의 메소드를 추가합니다:

```
class UserController extends Controller {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

위의 경우에 컨트롤러의 `index` 메소드는 `users` URI에 대한 루트 주소에 대한 결과를 반환합니다.

만약 컨트롤러의 메소드가 여러개의 단어로 구성되어 진 형태라면 "-"을 통해서 접속할 수 있는 URI를 제공하게 됩니다. 예를 들어, `UserController`에 다음과 같은 액션이 정의되었다면 URI는 `users/admin-profile`과 같이 구성됩니다:

```
public function getAdminProfile() {}
```

라우트에 이름 지정하기

컨트롤러 라우트에 어떤 “이름”을 지정하고자 한다면 `controller` 메소드의 세 번째 인자를 통해서 지정할 수 있습니다:

```
Route::controller('users', 'UserController', [
    'anyLogin' => 'user.login',
]);
```

의존성 주입 & 컨트롤러

생성자 주입

XE의 서비스 컨테이너는 모든 컨트롤러의 의존성을 해결하기 위해서 사용됩니다. 그 결과 컨트롤러가 필요로 하는 의존 객체들에 대해서 생성자에서 타입힌트로 지정할 수 있게 됩니다:

```php
<?php namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use App\Repositories\UserRepository;

class UserController extends Controller {

    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

}
```

서비스 컨테이너가 의존성을 해결을 할 수 있다면 타입 힌트에 지정할 수는 있습니다.

메소드 인젝션-주입

생성자 주입과 더불어 컨트롤러의 메소드에서도 타입힌트를 통한 의존성 주입을 할 수 있습니다. 예를 들어, 메소드에서 `Request` 인스턴스를 타입힌트를 통해서 주입할 수 있습니다:

```php
<?php namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }

}
```

컨트롤러 메소드가 라우트 인자로부터 입력값을 받아야 한다면 간단하게 의존성 지정 뒤에 인자를 지정하면 됩니다:

```php
<?php namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Update the specified user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

요청(Request)

XE는 웹브라우저로부터 요청을 받으면, 제일 먼저 `index.php` 가 실행되고 `index.php` 는 현재 요청에 대한 정보를 담고 있는 `Request` 인스턴스를 생성합니다. 이 `Request` 인스턴스는 XE가 실행되는 동안 매우 많은 곳에서 로드되어 현재 요청에 대한 정보를 참조할 수 있도록 합니다.

Request 인스턴스 획득하기

파사드를 이용한 방법

`Request` 파사드는 컨테이너와 결합된 현재의 Request에 엑세스 할 수 있도록 해줍니다. 예를 들면:

```
$name = Request::input('name');
```

만약 특정 네임스페이스 아래에서 `Request` 파사드를 사용하고자 한다면 클래스 상단부분에 `use Request;` 구문을 추가해야 된다는 것을 기억하십시오.

의존성 주입을 통한 방법

현재의 의존성 주입을 통해서 HTTP request을 획득하기 위해서는 여러분의 컨트롤러 생성자나 메소드에서 타입힌트를 지정해야 합니다. XE의 request 인스턴스는 `\Xpressengine\Http\Request` 클래스의 인스턴스입니다. 이 클래스는 `\Illuminate\Http\Request` 를 상속받고 있습니다.

현재의 request의 인스턴스는 서비스 컨테이너에 의해서 자동으로 주입될 것입니다:

```
<?php namespace App\Http\Controllers;

use Xpressengine\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }

}
```

만약 컨트롤러 메소드에서 라우트 파라미터를 입력값으로 받아야 한다면 의존성을 지정한 뒤에 라우트 파라미터를 나열하면 됩니다:

```php
<?php namespace App\Http\Controllers;

use Xpressengine\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Update the specified user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

입력값 검색하기

입력값 검색하기

간단한 메소드를 통해서 `Xpressengine\Http\Request` 인스턴스 모든 사용자 입력값에 엑세스 할 수 있습니다. request에서 어떤 HTTP 메소드를 사용했는지에 대해서는 걱정할 필요 없이 모든 HTTP 메소드에 대해서 같은 방법으로 입력값에 대해 엑세스가 가능합니다.

```php
$name = $request->input('name');
```

입력값이 없을 때 기본값 가져오기

```php
$name = $request->input('name', 'Sally');
```

입력값이 존재하는지 확인하기

```php
if ($request->has('name'))
{
    //
}
```

전체 입력값 가져오기

```php
$input = $request->all();
```

Request 입력 중에서 몇개의 값만 가져오기

```php
$input = $request->only('username', 'password');

$input = $request->except('credit_card');
```

입력폼에 배열로 값이 전달된다면 '점'으로 구분하여 입력값에 엑세스 할 수 있습니다:

```php
$input = $request->input('products.0.name');
```

이전 입력

XE는 현재 request의 입력값을 다음 request까지 유지하는 방법을 제공합니다. 예를 들어, 폼의 입력값 체크에서 에러가 발생하면 작성한 값들을 다시 채워줘야 할 필요가 있을 수 있습니다.

입력값들 세션에 저장하기

`flash` 메소드는 현재의 입력들을 세션에 저장하여 사용자가 다음번에 request를 보내도 사용가능하게 만들어 줍니다.

```
$request->flash();
```

몇개의 입력값만 세션에 저장하기

```
$request->flashOnly('username', 'email');

$request->flashExcept('password');
```

플래쉬 & 리다이렉트

대부분 이전 페이지로 리다이렉트 하면서 입력값을 플래쉥 하기를 원하는 데, 이 경우 리다이렉트와 함께 입력값 플래싱을 메소드 체이닝으로 사용할 수 있습니다.

```
return redirect('form')->withInput();

return redirect('form')->withInput(Request::except('password'));
```

이전 입력값 검색하기

이전 Request에 대해 저장된 입력값을 검색하기 위해서는 `Request` 인스턴스의 `old` 메소드를 사용하면 됩니다.

```
$username = Request::old('username');
```

블레이드 템플릿 안에서 지난 입력값을 보여주려면 `old` 헬퍼함수를 사용하는 것이 보다 편리합니다:

```
{{ old('username') }}
```

파일 처리

Request 인스턴스의 `file` 메소드를 사용하면 사용자가 업로드한 파일을 엑세스할 수 있습니다. `file` 메소드에 의해 반환되는 값은 `Symfony\Component\HttpFoundation\File\UploadedFile` 클래스의 인스턴스입니다. 이 인스턴스의 다양한 메소드를 사용하여 업로드된 파일에 대한 정보를 참조할 수 있습니다.

업로드한 파일 가져오기

```
$file = $request->file('photo');
```

파일이 업로드 되었는지 확인하기

```
if ($request->hasFile('photo'))
{
    //
}
```

업로드한 파일이 유효한지 판단하기

```
if ($request->file('photo')->isValid())
{
    //
}
```

업로드한 파일 이동하기

```
$request->file('photo')->move($destinationPath);

$request->file('photo')->move($destinationPath, $fileName);
```

기타 파일 메소드

그 밖에도 다양한 메소드들이 `UploadedFile` 인스턴스에 준비되어 있습니다. 추가적인 메소들에 대한 정보는 API 문서를 참고하십시오.

기타 Request에 대한 정보

`Request` 클래스는 `Symfony\Component\HttpFoundation\Request` 클래스를 상속하고 있으며 어플리케이션을 위한 HTTP request을 확인하는 많은 메소드를 제공하고 있습니다. 다음은 몇몇 예시들입니다.

Request URI 가져오기

```
$uri = $request->path();
```

Request 가 AJAX 요청인지 확인

```
if ($request->ajax())
{
    //
}
```

Request 메소드 확인하기

```
$method = $request->method();

if ($request->isMethod('post'))
{
    //
}
```

현재 request가 패턴에 일치하는지 확인하기

```
if ($request->is('admin/*'))
{
    //
}
```

현재 request URL 가져오기

```
$url = $request->url();
```

응답(Response)

Response 사용의 제한

XE의 기본 프레임워크인 라라벨에서는 대부분의 라우트나 컨트롤러 액션에서 `Illuminate\Http\Response` 의 인스턴스나 뷰를 반환하도록 합니다.

하지만 XE는 웹 브라우저로 html 형식의 응답을 보낼 때, 스킨과 테마를 적용한 후 보내야 합니다. 특별한 경우가 아니라면 컨트롤러에서 `Illuminate\Http\Response` 인스턴스나 뷰를 직접 반환(return)하지 마십시오. 대신, 프리젠터를 사용하여 반환하십시오. 반드시 프리젠터를 사용해야만 테마와 스킨이 적용되고 위젯 또한 정상적으로 출력됩니다.

리다이렉트

일반적으로 리다이렉트 Response는 `Illuminate\Http\RedirectResponse` 클래스의 인스턴스이며, 사용자를 다른 URL로 리다이렉트하는 데 필요한 적절한 헤더를 포함하고 있습니다.

리다이렉트 반환하기

`RedirectResponse` 인스턴스를 생성하는 데는 몇 가지 방법이 있습니다. 가장 간단한 방법은 `redirect` 헬퍼 함수를 사용하는 것입니다. 테스트를 진행할 때 리다이렉트 Response를 생성하는 모킹(Mock)은 일반적으로 잘 하지 않기 때문에, 대부분의 경우에 헬퍼 함수를 사용하게 됩니다.

```
return redirect('user/login');
```

리다이렉트에 플래시 데이터와 함께 반환하기

새로운 URL로 리다이렉트 이동하고 플래시 데이터를 세션에 저장 하는 것은 일반적으로 동시에 진행됩니다. 따라서 편의성을 높이기 위해 `RedirectResponse` 인스턴스를 생성하고 **동시에** 메소드 체인을 통해 플래시 데이터를 세션에 저장할 수 있습니다:

```
return redirect('user/login')->with('message', 'Login Failed');
```

이전 URL로 리다이렉트

예를 들어, 폼 전송 후에, 사용자를 이전 URL로 리다이렉트 시키고자 하는 경우가 있을 수 있습니다. 이런 경우에는 `back` 메소드를 사용하면 됩니다:

```
return redirect()->back();

return redirect()->back()->withInput();
```

이름이 지정된 라우트로 리다이렉트 하기

전달 인자 없이 `redirect` 헬퍼 함수를 호출할 때에는 `Illuminate\Routing\Redirector` 의 인스턴스가 반환됩니다. 따라서 `Redirector` 인스턴스의 메소드를 사용할 수 있습니다. 예를 들어, 이름지 지정된 라우트로 이동하는 `RedirectResponse` 를 생성하고자 한다면 `route` 메소드를 사용할 수 있습니다:

```
return redirect()->route('login');
```

이름이 지정된 라우트로 파라미터와 함께 리다이렉트 하기

라우트에 전달해야 할 파라미터가 있다면 `route` 메소드의 두 번째 인자로 전달하면 됩니다.

```
// For a route with the following URI: profile/{id}

return redirect()->route('profile', [1]);
```

이름지 지정된 라우트로 파라미터 이름과 함께 리다이렉트 하기

```
// For a route with the following URI: profile/{user}

return redirect()->route('profile', ['user' => 1]);
```

컨트롤러 액션으로 리다이렉트 하기

이름이 지정된 라우트로 이동하는 `RedirectResponse` 인스턴스를 생성하는것과 비슷하게 컨트롤러 액션으로 리다이렉션 할 수 있습니다.

```
return redirect()->action('App\Http\Controllers\HomeController@index');
```

> **주의:** `URL:setRootControllerNamespace` 를 통해서 컨트롤러의 루트 네임스페이스가 지정되었다면, 전체 네임 스페이스를 지정할 필요가 없습니다.

컨트롤러 액션으로 파라미터와 함께 리다이렉트 하기

```
return redirect()->action('App\Http\Controllers\UserController@profile', [1]);
```

컨트롤러 액션으로 파라미터 이름과 함께 리다이렉트 하기

```
return redirect()->action('App\Http\Controllers\UserController@profile', ['user' => 1]);
```

기타 Response

`response` 헬퍼 함수를 사용하여 편리하게 다른 타입의 response 인스턴스를 생성할 수도 있습니다. `response` 헬퍼함수를 인자 없이 호출하게 되면 `Illuminate\Contracts\Routing\ResponseFactory` contract를 반환합니다. 이 contract는 response를 생성하기 위한 다양한 메소드를 제공합니다.

JSON response 생성하기

`json` 메소드는 헤더의 `Content-Type` 을 자동으로 `application/json` 으로 지정합니다:

```
return response()->json(['name' => 'Abigail', 'state' => 'CA']);
```

JSONP Response 생성하기

```
return response()->json(['name' => 'Abigail', 'state' => 'CA'])
                 ->setCallback($request->input('callback'));
```

파일 다운로드 Response 생성하기

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);

return response()->download($pathToFile)->deleteFileAfterSend(true);
```

> **참고:** 파일 다운로드를 관리하는 Symfony의 HttpFoundation에서 다운로드 할 파일의 이름이 ASCII 파일 이름임을 필요로 하고 있습니다.

프리젠터(Presenter)

- 3.0.0-beta6 변경 사항
  > 화면 출력을 담당하는 renderer 의 명칭을 presentable 로 변경하고 이 인터페이스를 따르는 구현체의 이름을 변경함
- 3.0.0-beta7 변경 사항
  > 프리젠터의 본래 목적인 html, api 형식의 유연한 처리를 위해서 XePresenter::make() 을 변경합니다. 이전에는 make() 가 html에 대한 지원만 처리하였지만 이후 부터 make 를 사용할 경우 html, api 모두를 지원하도록 변경합니다.

XE에서는 웹 브라우저로 응답을 보낼 때 뷰를 대신해서 프리젠터를 사용합니다.

프리젠터는 일반적인 HTML 응답뿐만 아니라 API 요청에 대한 Json 응답을 하나의 메소드로 처리할 수 있습니다. 프리젠터는 요청(Request) 정보에 포함된 응답 포멧을 검사하고, 데이터를 해당 포멧에 맞게 반환합니다.

XE 설계 과정에서, 하나의 컨트롤러를 이용해서 HTML 형식과 API 형식(Json)의 응답을 모두를 처리하여 유지보수의 비용을 줄이고자 하는 요구사항이 있었습니다.

프리젠터는 응답 포멧에 따라 presetable 한 구현체를 선택합니다. 기본으로 `HtmlPresenter` , `JsonPresenter` 두개의 구현체를 포함하고 있습니다. 만약 API 요청의 응답을 Json이 아닌 XML 형식으로 받고싶다면, 플러그인을 통해 presentable 한 XML 구현체를 추가하고 요청에서 반환 포멧을 xml로 하면 됩니다.

프리젠터는 XePresenter 파사드를 제공합니다.

HTML 형식으로 응답하기

Html 응답을 처리할 경우, `HtmlPresenter` 는 테마, 스킨을 처리합니다. `HtmlPresenter` 는 테마 핸들러에게 테마 컴포넌트를 받아 처리합니다. 스킨은 `XePresenter::setSkinTargetId()` 으로 외부에서 스킨 타겟 아이디를 입력받아 스킨 핸들러를 사용해 설정된 스킨을 사용합니다.

XE에서 HTML 형식으로 응답할 때에는, 컨트롤러가 생성한 데이터를 스킨을 사용해 HTML로 변환합니다. 또, 컴파일된 HTML에 테마를 적용하여 반환합니다. 프리젠터에서 스킨을 선택할 수 있도록 응답할 때 어떤 종류의 스킨을 사용해야 하는지 설정하고 사용해야 합니다.

> HTML 형식만 지원하려고 싶을 경우 `XePresenter::makeHtml()` 을 사용합니다.

```
// in app/Html/Controllers/ProfileController.php

// 스킨 타겟 지정
XePresenter::setSkinTargetId('member/profile');

...

// HTML 형식으로 반환, 'index' 뷰를 사용
return XePresenter::makeHtml('index', compact('user', 'grant'));
```

스킨을 사용하지 않고, 템플릿 파일의 뷰 이름을 바로 지정할 수도 있습니다.

```
// in app/Html/Controllers/SeoController.php

// 뷰 이름을 직접 지정
return XePresenter::makeHtml('seo.setting');
```

스킨 타겟 아이디

스킨은 특정 컴포넌트나 집단 밑에 하나의 그룹을 형성합니다.

코어에서 제공하는 프로필 컨트롤러는 `member/profile` 아이디를 스킨 타겟 아이디로 사용하고 있습니다. 플러그인으로 프로필 스킨을 만드려고 할 때 제작되는 스킨 컴포넌트는 `member/profile` 를 대상 아이디로 컴포넌트 아이디를 생성해야 하며 스킨 패키지는 이를 기준으로 스킨을 설정할 수 있는 관리자를 제공합니다.

다른 예로 Board 플러그의 Board 모듈 컴포너트는 `module/board@board` 스킨 타겟 아이디를 사용합니다.

스킨을 사용하는 컨트롤러는 `XePresenter::setSkinTargetId()` 에 지정된 스킨 타겟 아이디를 설정하여 `HtmlRenderer::render()` 가 처리될 때 설정된 스킨을 찾아 처리할 수 있도록 합니다.

HtmlPresenter

프리젠터에서 Html 응답처리할 `XePresenter::make()` 할 경우 기본으로 `/resources/views/` 를 참고합니다. 이것은 `HtmlPresenter::renderSkin()` 에서 스킨의 타겟 아이디가 지정되지 않았을 경우 뷰를 직접 사용하기 때문입니다.

정상적인 사용 과정으로 스킨의 타겟 아이디를 프리젠터에게 전달한 경우 HtmlPresenter는 사용될 스킨을 찾아 Renderable 인터페이스의 `render()` 를 실행시키며 스킨 컴포넌트는 뷰를 사용해서 블레이드 파일을 처리합니다.

전체 프레임 구성 파일

`resources/views/common/base.blade.php` 으로 전체 프레임을 구성하며 `$content` 에 테마를 전달받아 출력합니다.

`HtmlPresenter::render()` 는 SEO, 스킨, 테마 순서로 처리되고 마지막에 `self::$commonHtmlWrapper` 으로 감싸서 반환합니다. `self::$commonHtmlWrapper` 는 `app/Providers/PresenterServiceProvider.php` 에서 `config/xe.php` 의 `HtmlWrapper` 로 설정합니다.

`base.blade.php` 는 프론트엔드에 등록된 js, css등 여러 요소들을 어떤 위치에 출력할지 결정하고 있습니다.

API 형식으로 응답하기

XE는 json 형식을 지원하며 `JsonPresenter` 가 사용됩니다.

```
// in app/Html/Controllers/DynamicFieldController.php

  ...

  // $list를 json 형식으로 변환하여 반환
  return XePresenter::makeApi(['list' => $list]);
```

모든 형식 지원

HTML, API 모든 형식을 지원하기 위해서 `XePresenter::make()` 를 사용합니다.

```
// in src/Controllers/UserController.php

  ...

  // 응답 형식에 따라 $data를 json 형식으로 반환하거나,
  // $data를 index 템플릿(스킨)에 적용한 후, HTML 형식으로 반환
  return XePresenter::make('index', $data);
```

> XE의 API를 이용한 개발 케이스가 많지 않아 API 지원에 대한 부분은 계속 개선해야 합니다.

뷰(View)

컨트롤러는 `Request` 를 처리한 다음 다양한 형태의 `Response` 를 반환합니다. `GET` 요청일 경우 대부분 html 형식으로 `Response` 를 반환할 것이며, 때로는 json 형식으로 반환할 것입니다. `POST` 요청일 경우에는 대부분 리다이렉트(redirect) 형식으로 반환할 것입니다.

뷰는 컨트롤러가 html 형식의 `Response` 를 만들 때 사용됩니다. 컨트롤러는 html 형식의 `Response` 를 반환하기 위해 우선 필요한 데이터를 생성합니다. 그 다음, 생성된 데이터를 미리 정의된 템플릿에 적용하여 html 형식의 `Response` 를 만들어야 합니다. 이때 뷰에 템플릿 정보와 데이터를 전달하면 뷰는 이를 html로 생성해줍니다.

뷰는 프리젠테이션 로직을 컨트롤러 및 어플리케이션 로직과 분리해주는 역할을 합니다.

뷰 사용의 제한

XE는 웹 브라우저로 html 형식의 응답을 보낼 때, 스킨과 테마를 적용한 후 보내야 합니다. 특별한 경우가 아니라면 컨트롤러에서 뷰를 직접 반환(return)하지 마십시오. 대신 프리젠터를 사용하여 반환하십시오. 반드시 프리젠터를 사용해야만 테마와 스킨이 적용되고 위젯 또한 정상적으로 출력됩니다.

단, 컨트롤러가 아닌 다른 구성요소이거나, 컨트롤러에서라도 반환하는 값이 아니라면, html 스트링을 생성할 때 뷰를 자유롭게 사용하십시오.

기본 사용법

간단한 뷰는 다음과 같습니다:

```
// plugins/my_plugin/views/greeting.php
<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

뷰는 다음과 같이 브라우저로 보내집니다:

```
Route::get('/', function()
{
    return view('my_plugin::views.greeting', ['name' => 'James']);
});
```

위와 같이 `view` 헬퍼함수에 전달하는 첫 번째 인자는 템플릿 파일의 이름이 됩니다. 두 번째 전달 인자는 템플릿에서 사용하기 위한 데이터의 배열입니다.

템플릿 파일을 지정할 때, `<플러그인아이디>::` 형태의 전치사(prefix)를 사용하면 플러그인 디렉토리를 기준으로 하는 상대경로로 파일 경로를 지정할 수 있습니다. 또, 점(.)을 사용하여 중첩된 서브 디렉토리에 있는 파일을 지정할 수도 있습니다.

뷰에 데이터 전달하기

```
// Using conventional approach
$view = view('greeting')->with('name', 'Victoria');

// Using Magic Methods
$view = view('greeting')->withName('Victoria');
```

위 예제의 경우 뷰에서는 `$name` 변수에 `Victoria` 라는 값을 확인할 수 있습니다.

필요한 경우에 `view` 헬퍼함수에 두 번째 인자로 데이터 배열을 전달할 수도 있습니다:

```
$view = view('greetings', $data);
```

이러한 방식으로 정보를 전달할 때, `$data` 는 키/값으로 구성된 배열이어야 합니다. 뷰 안에서 여러분은 `{{ $key }}` 와 같이 각각의 키에 해당하는 값에 엑세스 할 수 있습니다. (`$data['$key']` 는 존재한다고 가정합니다.)

파일 패스로부터 view 반환

필요하다면 절대경로를 기반으로 뷰를 생성할 수도 있습니다:

```
return view()->file($pathToFile, $data);
```

템플릿(Blade Template)

소개

XE에서 제공하고 있는 블레이드는 간결하고 강력한 템플릿 엔진입니다. 다른 인기있는 PHP 템플릿 엔진들과는 다르게, 블레이드는 여러분이 순수한 PHP 코드를 뷰에 사용하는 것을 제한하지 않습니다. 모든 블레이드 뷰는 순수한 PHP 코드로 컴파일된 후 캐싱되고, 파일이 수정되지 않을 때까지 캐싱된 파일을 사용합니다. 이는 블레이드가 본질적으로 성능상의 부담이 없음을 의미합니다. 블레이드 뷰 파일은 확장자로 `.blade.php`를 사용합니다.

템플릿의 상속

레이아웃 정의하기

블레이드를 사용할 때의 주된 장점 두가지는 템플릿 상속과 섹션입니다. 시작하기 전에 간단한 예제를 살펴보겠습니다. 첫번째로, 우리는 "master" 페이지 레이아웃을 보겠습니다. 대부분의 웹사이트는 여러 페이지에 걸쳐 동일한 레이아웃을 사용하기 때문에, 이 레이아웃을 하나의 블레이드 뷰로 정의하는 것이 편합니다.

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

보는 바와 같이, 이 파일은 일반적인 HTML 마크업을 가지고 있습니다. 그런데 `@section` 과 `@yield` 지시어에 주목해 주십시오. `@section` 지시어는 이름에서도 알 수 있듯이 컨텐츠의 섹션을 정의하고 있고. 반대로 `@yield` 지시어는 주어진 섹션의 컨텐츠를 출력하고 있습니다.

이제 레이아웃은 정의했고, 이 레이아웃을 상속받을 자식 페이지를 정의해 보겠습니다.

레이아웃 확장하기

자식페이지를 정의할 때, `@extends` 지시어를 사용하여 자식 페이지에서 "상속"받을 레이아웃을 지정할 수 있습니다. 레이아웃을 상속(`@extends`)받는 뷰들은 `@section` 지시어를 사용해서 그 레이아웃의 섹션에 들어갈 컨텐츠를 주입해야 합니다. 앞의 예에서 보았듯이, 자식 페이지에서 주입한 섹션의 컨텐츠는 레이아웃의 `@yield` 부분에 출력될 것입니다.

```
<!-- Stored in resources/views/child.blade.php -->

@extends('layouts.master')

@section('title', 'Page Title')

@section('sidebar')
    @@parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

In this example, the `sidebar` section is utilizing the `@@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@@parent` directive will be replaced by the content of the layout when the view is rendered.

위의 예에서, `sidebar` 섹션은 레이아웃의 사이드바에 컨텐츠를 붙이기 위해 `@@parent` 지시어를 사용하고 있습니다. `@@parent` 지시어는 뷰가 렌더링 될 때, 그 레이아웃의 `sidebar` 섹션이 가지고 있는 컨텐츠로 대체될 것입니다.

Of course, just like plain PHP views, Blade views may be returned from routes using the global `view` helper function:

블레이드 템플릿은 순수한 PHP 뷰와 마찬가지로 `view` 헬퍼를 사용하여 곧바로 반환될 수 있습니다.

```
Route::get('blade', function () {
    return view('child');
});
```

데이터 출력하기

You may display data passed to your Blade views by wrapping the variable in "curly" braces. For example, given the following route:

중괄호(culry brace)로 변수를 감싸면, 블레이드 뷰로 전달된 데이터를 출력할 수 있습니다. 예를 들어, 주어진 라우트가 있을 때:

```
Route::get('greeting', function () {
    return view('welcome', ['name' => 'Samantha']);
});
```

`name` 변수의 값을 다음과 같이 출력할 수 있습니다:

```
Hello, {{ $name }}.
```

뷰로 전달된 변수들의 값을 출력하는 것에 별다른 제한은 없습니다. PHP 함수의 결과 값을 출력할 수도 있습니다. 블레이드의 데이터 출력 구문 안에는 어떤 PHP 코드도 들어갈 수 있습니다.

```
The current UNIX timestamp is {{ time() }}.
```

> **주의:** `{{ }}` 구문은 자동으로 PHP의 htmlentities` 함수로 감싼 후 출력합니다. XSS 공격을 방어하기 위함입니다.

블레이드 & 자바스크립트 프레임워크

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the `@` symbol to inform the Blade rendering engine an expression should remain untouched. For example:

많은 자바스크립트 프레임웍에서도 브라우저에 출력되어야 할 데이터를 표시하기 위해 중괄호("curly" braces)를 사용하고 있습니다. 이럴 경우, `@` 기호를 사용하면 블레이드 랜더링 엔진은 이 구문을 변환하지 않고 그대로 출력할 것입니다. 예를 들어:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

위의 예에서 `@` 기호는 블레이드에 의해서 제거될 것이고, 나머지 `{{ name }}` 구문은 블레이드 엔진에 의해 해석되지 않고 그대로 남아있게 되어, 자바스크립트 프레임워크에 의해 변환되어 집니다.

데이터가 존재하는지 확인후 출력하기

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set. We can express this in verbose PHP code like so:

가끔 변수를 출력할 때, 그 변수가 실제로 존재하는지 확신하지 못할 때도 있습니다. 이때 다음과 같이 PHP 코드를 사용할 수 있습니다.

```
{{ isset($name) ? $name : 'Default' }}
```

이렇게 3항연산자를 사용하는 대신, 블레이드는 다음과 같은 편리한 구문을 제공합니다:

```
{{ $name or 'Default' }}
```

위의 예에서, 만약 `$name` 변수가 존재하면 그 값이 출력될 것이고, 존재하지 않는다면 대신 `Default` 라는 문자가 출력될 것입니다.

이스케이프하지 않고 데이터 출력하기

By default, Blade `{{ }}` statements are automatically sent through PHP's `htmlentities` function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

기본적으로, `{{ }}` 구문은 XSS 공격을 방어하기 위해 자동으로 PHP의 `htmlentities` 함수를 실행후 반환합니다. 만약 데이터를 `htmlentities` 함수를 거치지 않은채 출력하고 싶다면, 다음과 같은 구문을 사용하십시오.

```
Hello, {!! $name !!}.
```

> **주의:** 사용자로부터 입력 된 내용을 표시 할 때에는 escape에 대한 매우 세심한 주의가 필요합니다. 컨텐츠의 HTML 엔티티를 escape 하기위해 항상 이중 중괄호 표기법을 사용하십시오.

제어문

블레이드는 템플릿 상속 및 데이터 출력과 더불어, 조건문이나 반복문과 같이 일반적인 PHP 제어문의 실행을 위해 편리하고 간결한 구문을 제공합니다. 이 구문들은 매우 깔끔하고 간단하면서도, 대응되는 PHP 제어문들과 비슷한 모습을 띄고 있습니다.

If문

`if` 문은 `@if`, `@elseif`, `@else` 와 `@endif` 지시어를 사용하여 구성할 수 있습니다. 이 지시어들은 각각 대응되는 PHP 구문과 동일하게 작동합니다.

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

편의성을 위해, 블레이드는 `@unless` 지시어를 제공합니다.

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

반복문

In addition to conditional statements, Blade provides simple directives for working with PHP's supported loop structures. Again, each of these directives functions identically to their PHP counterparts:

조건문과 더불어, 블레이드는 PHP가 지원하는 반복문 기능을 하는 간단한 지시어를 제공합니다. 다시 한번 말하지만, 각각의 지시어들은 대응하는 PHP 구문과 동일한 기능을 합니다.

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

서브 뷰 포함하기

`@include` 지시어는 손쉽게 블레이드 뷰를 현재의 뷰에 포함(include)시킬수 있도록 도와줍니다. 부모 뷰에서 사용할 수 있는 모든 변수는 포함된 서브 뷰에서도 사용할 수 있습니다.

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of extra data to the included view:

서브 뷰는 부모 뷰의 모든 데이터를 상속받겠지만, 그 외에 더 많은 데이터를 배열 형식으로 전달할 수 있습니다.

```
@include('view.name', ['some' => 'data'])
```

> **주의:** 블레이드 뷰 안에서 `__DIR__` 과 `__FILE__` 과 같은 상수를 사용하지 마십시오. 블레이드 뷰는 캐싱된 뷰의 위치를 참조하기 때문입니다.

컬렉션을 뷰에서 렌더링하기

블레이드의 `@each` 지시어를 사용하면 반복문(loop)과 `include` 구문을 한 줄로 합칠 수 있습니다.

```
@each('view.name', $jobs, 'job')
```

첫번째 인자는 배열이나 컬렉션의 각 요소를 렌더링하기 위한 서브 뷰의 이름입니다. 두번째 인자는 반복 처리하는 배열이나 컬렉션이며 세번째 인수는 뷰에서의 반복값이 대입되는 변수의 이름입니다. 예를 들어 `jobs` 배열을 반복 처리하려면 보통 서브 뷰에서 각 과제를 `job` 변수로 접근해야 할 것입니다.

You may also pass a fourth argument to the `@each` directive. This argument determines the view that will be rendered if the given array is empty.

또한 `@each` 지시어로 네번째 인수를 전달할 수도 있습니다. 이 인자는 특정 배열이 비었을 경우 렌더링될 뷰를 결정합니다.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

주석

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

블레이드는 또한 뷰에 주석을 정의할 수 있습니다. 하지만 HTML 주석과는 다르게, 블레이드 주석은 브라우저로 전송되는 HTML에 포함되어 있지 않습니다:

```
{{-- This comment will not be present in the rendered HTML --}}
```

서비스 주입하기

The `@inject` directive may be used to retrieve a service from the Laravel service container. The first argument passed to `@inject` is the name of the variable the service will be placed into, while the second argument is the class / interface name of the service you wish to resolve:

`@inject` 지시어는 서비스 컨테이너로부터 서비스를 조회하는 데에 사용될 수 있습니다. `@inject` 에 전달된 첫번째 인자는 서비스가 할당될 변수의 이름이며 두번째 인자는 주입하려는 서비스의 클래스/인터페이스의 이름입니다:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

플러그인 개발 시작하기

플러그인 생성 커맨드

처음 플러그인 개발을 시작할 때 부딪히는 난관은 플러그인에 필요한 기본적인 디렉토리와 파일들을 직접 생성하는 것입니다. 만약 `my_plugin` 이라는 플러그인을 하나 만들기 시작한다면, 우선 `plugins` 디렉토리에 `my_plugin` 이라는 이름의 디렉토리를 만들고, 그 안에 `plugin.php` , `composer.json` 파일을 만드는 것부터 시작해야 합니다. 이러한 수고를 줄이기 위하여 XE는 플러그인 생성 커맨드를 제공합니다.

플러그인 생성 커맨드를 사용해서 만든 플러그인은 웹페이지를 출력하는 기본적인 기능을 샘플로 포함하고 있습니다. 원치 않을 경우 샘플 웹페이지 출력 기능을 삭제하시고, 플러그인 개발을 시작하시기 바랍니다.

터미널에서 아래와 같이 명령어를 실행하십시오.

```
$ php artisan make:plugin <name> <namespace> <title>
```

`name` 파라메터는 플러그인의 고유 id입니다. 플러그인의 디렉토리 이름으로도 사용됩니다.

`namespace` 파라메터에는 플러그인 클래스의 네임스페이스를 지정합니다. 지정한 네임스페이스는 플러그인 클래스 뿐만 아니라 플러그인 내에 존재하는 모든 PHP 클래스의 네임스페이스로도 사용됩니다. 이 네임스페이스는 다른 개발자가 작성한 클래스와 클래스명이 동일할 때 서로 구분하기 위해 사용됩니다. 다른 사람과 중복되지 않는 자신만의 고유한 네임스페이스를 지정하십시오. 가능하면 자신의 이름이나 소속회사명을 사용하시길 권장합니다.

예를 들어, 본인의 이름이 'SungbumHong'이고 'foo' 플러그인이라면 `SungbumHong\XePlugins\Foo` 또는 `SungbumHong\Foo` 를 네임스페이스로 사용하십시오. 또다른 플러그인 'bar'가 있다면, bar 플러그인에는 `SungbumHong\XePlugins\Bar` 를 네임스페이스로 사용할 수 있습니다.

플러그인 구조

XE 플러그인은 하나의 디렉토리로 구성되며, 디렉토리 안에는 플러그인에 필요한 파일들이 포함돼 있습니다. 아래의 파일 목록은 각 플러그인이 포함해야 할 필수 파일들입니다.

```
plugins/
└── myplugin/
    ├── composer.json
    └── plugin.php
```

> 플러그인을 생성할 때에는 직접 모든 파일을 작성하지 마시고, 플러그인 생성 명령(`php artisan make:plugin`)을 사용하시기 바랍니다. 몇가지 설정만 입력해주면 플러그인에 필요한 파일을 자동으로 작성해주고, 권장하는 디렉토리 구조를 생성해 줍니다.

composer.json

XE 플러그인은 하나의 composer 패키지이기도 합니다. 다른 플러그인이나 라이브러리 패키지에 대한 의존성 처리, 오토로드(autoload)와 같은 composer의 장점을 활용하기 위해, XE에서는 composer를 사용하여 플러그인을 관리합니다. `composer.json` 파일은 composer가 패키지의 정보를 담을 때 사용하는 파일입니다. 또한 XE는 XE가 자체적으로 필요로 하는 플러그인 정보를 담기 위해 이 파일을 같이 사용합니다.

아래 코드는 `alice` 번들 플러그인의 `composer.json` 파일입니다.

```
{
  "name": "xpressengine-plugin/alice",
  "description": "This Package is Xpressengine Plugin - Alice Theme.",
  "keywords": ["xpressengine", "plugin", "theme", "alice"],
  "license": "LGPL-2.1",
  "version": "0.9.0",
  "type": "xpressengine-plugin",
  "authors": [{
      "name": "xpressengine"
  }],
  "extra": {
    "xpressengine": {
      "title": "Alice Theme",
      "screenshots": [
        "screenshots/main.png",
        "screenshots/sub.png",
        "screenshots/site.png"
      ],
      "icon": "icon.png",
      "component": {
        "theme/alice@alice": {
          "class": "Xpressengine\\Plugins\\Alice\\Theme\\Alice",
          "name": "Alice",
          "description": "The First Theme for XpressEngine3",
          "screenshot": "/plugins/alice/screenshots/main.png",
        }
      }
    }
  },
  "autoload": {
    "psr-4": {
      "Xpressengine\\Plugins\\Alice\\": "src/"
    }
  }
}
```

`composer.json` 파일은 json 형식으로 정보를 담고 있습니다. 대부분 composer가 필요로 하는 정보들입니다. 여러분이 눈여겨 보아야 할 정보는 `extra > xpressengine` 에 기록된 정보입니다. `extra > xpressengine` 에는 이 플러그인의 제목(title), 스크린샷 목록(screenshots), 아이콘(icon), 그리고 이 플러그인에서 등록하는 XE 컴포넌트에 대한 정보(component)를 담고 있습니다.

plugin.php

`composer.json` 파일이 플러그인에 대한 정보를 담고 있는 파일이라면, `plugin.php` 파일은 플러그인의 실제 작동을 하는데 필요한 코드가 기술되는 파일입니다.

XE는 플러그인이 설치(install), 업데이트(update), 활성화(activate), 비활성화(deactivate), 삭제(uninstall), 부트(boot)될 때, 이 파일에 기술된 코드를 실행하여 플러그인이 필요로 하는 작업을 할 수 있도록 합니다.

`plugin.php` 에는 반드시 하나의 PHP 클래스가 기술되어야 합니다. 이 플러그인 클래스는 반드시 `\Xpressengine\Plugin\AbstractPlugin` 클래스를 `extends` 해야 합니다.

가장 간단한 형태의 `plugin.php` 파일입니다.

```
<?php
namespace MyPlugin\Sample;

use Xpressengine\Plugin\AbstractPlugin;

class Plugin extends AbstractPlugin
{
    public function boot()
    {
        // implement code
    }
}
```

플러그인 클래스는 아래의 메소드를 구현할 수 있습니다.

메소드	설명
activate	플러그인이 활성화될 때 필요한 코드를 작성하십시오.
boot	플러그인이 부트될 때 필요한 코드를 작성하십시오.
checkInstalled	플러그인의 설치 여부를 체크한 결과를 반환하십시오. 이 메소드가 `false` 를 반환하면 플러그인이 활성화 될 때 `install` 메소드가 실행됩니다.
install	데이터베이스 테이블 생성코드와 같이 플러그인을 설치할 때 필요한 코드를 작성하십시오.
checkUpdated	플러그인의 업데이트 여부를 체크한 결과를 반환하십시오. 이 메소드가 `false` 를 반환하면 플러그인이 활성화 될 때 `update` 메소드가 실행됩니다.
update	플러그인이 변경되었고, 데이터베이스 테이블의 컬럼 추가와 같은 작업이 필요하다면, 이 메소드에 필요한 코드를 작성하십시오.
deactivate	플러그인이 비활성화될 때 필요한 코드를 작성하십시오.
uninstall	플러그인이 삭제될 때 필요한 코드를 작성하십시오.
getSettingsURI	만약 플러그인이 '플러그인 설정' 페이지를 가진다면, 이 메소드에서 페이지의 url을 반환하십시오. 플러그인 관리페이지에서 반환한 링크가 노출됩니다.

권장하는 플러그인 구조

플러그인은 위에서 설명한 두개의 파일(`plugin.php` , `composer.json`)만으로도 구현이 가능합니다. 하지만 규모가 큰 플러그인을 제작할 때에는 더욱더 많은 파일을 필요로 합니다. 또, 많은 파일을 제대로 관리하기 위해서는 디렉토리 구조를 잘 설계해야 합니다. XE는 플러그인의 구조를 아래와 같이 권장하고 있습니다.

```
plugins/
└── myplugin
    ├── assets/
    ├── screenshots/
    ├── src/
    ├── views/
    ├── icon.png
    ├── composer.json
    └── plugin.php
```

`assets` 디렉토리는 stylesheet, script, 이미지 파일과 같은 asset 파일들을 담는 디렉토리입니다. 웹 브라우저에서 직접적으로 요청하는 파일들을 이 디렉토리에 넣으십시오.

`src` 디렉토리는 PHP 클래스나 함수와 같이 php로 작성된 파일들을 담는 디렉토리입니다. 컨트롤러 및 컴포넌트 파일들을 이 디렉토리에 넣으십시오.

`views` 디렉토리는 템플릿 파일을 담는 디렉토리입니다. XE에서는 blade 템플릿 엔진을 사용합니다. blade 템플릿으로 작성된 PHP 파일, 또는 순수한 PHP 작성된 템플릿 파일을 이 디렉토리에 넣으십시오.

`screenshots` 디렉토리에는 플러그인의 스크린샷 이미지를 넣으십시오.

`icon.png` 와 같이 플러그인의 아이콘 이미지 파일은 플러그인 디렉토리에 넣으십시오. 파일명과 확장자는 다를 수 있습니다.

스크린샷 이미지나 아이콘 이미지는 `composer.json` 의 `extra > xpressengine` 에 파일경로를 지정해 주어야 합니다.

플러그인 버전 관리

플러그인은 처음 설치될 때, 필요에 따라 플러그인에서 사용할 데이터베이스 테이블을 생성하거나, 설정을 저장하기도 하고, 필요한 파일을 미리 생성해 놓기도 합니다. 이런 작업들은 플러그인이 실행될 때마다 매번 필요한 작업이 아니라 처음 설치될 때 단 한번만 실행돼야 하는 작업입니다. 플러그인이 업데이트 되었을 경우에도 마찬가지입니다. 플러그인에 새로운 기능이 플러그인에 추가되었다면 데이터베이스 테이블을 변경해야 할 수도 있습니다.

이렇게 플러그인이 설치되고 업데이트될 때, 또는 플러그인이 삭제될 때 실행되어야 하는 코드는 `plugin.php` 파일의 플러그인 클래스에 작성하십시오.

플러그인 설치 과정

플러그인 클래스는 설치와 관련된 두개의 메소드를 가지고 있습니다. `checkInstalled` 와 `install` 메소드입니다.

```
<?php
namespace MyPlugin;

use Xpressengine\Plugin\AbstractPlugin;
use Schema;

class Plugin extends AbstractPlugin
{
  public function checkInstalled($installedVersion = null)
  {
      // 플러그인이 설치된 상태인지 체크하는 코드를 작성합니다.
  }

  public function install()
  {
      // 플러그인이 설치될 때 필요한 코드를 작성합니다.
  }

}
```

`checkInstalled` 메소드는 플러그인이 활성화될 때마다 호출됩니다. 만약 이 메소드가 `true` 를 반환하면 XE는 이 플러그인이 이미 XE에 설치된 상태로 간주하고 곧바로 플러그인을 활성화시킵니다. 반대로 이 메소드가 `false` 를 반환하면 XE는 이 플러그인이 아직 XE에 설치가 안 된 상태라고 판단합니다.

만약 필요한 테이블이 생성되어 있는지 검사하고 싶다면 아래와 같이 작성하면 됩니다.

```
public function checkInstalled($installedVersion = null)
{
    // 테이블이 존재하는지 검사, 없으면 false를 반환
    return Schema::hasTable('table_name');
}
```

XE는 `checkInstalled` 메소드의 리턴 값이 `false` 일 경우, `install` 메소드를 호출합니다.

```
public function install()
{
    // 플러그인이 설치될 때 필요한 코드를 작성합니다.
    Schema::create('table_name', function ($table) {
      $table->engine = 'InnoDB';
      $table->increments('id');
      $table->string('name', 200);
    });
}
```

플러그인 업데이트 과정

플러그인 클래스는 업데이트와 관련된 두개의 메소드를 가지고 있습니다. `checkUpdated` 와 `update` 메소드입니다. 두 메소드는 앞서 설명한 `checkInstalled` 와 `install` 메소드와 비슷한 작동과정을 가집니다.

```php
<?php
namespace MyPlugin;

use Xpressengine\Plugin\AbstractPlugin;
use Schema;

class Plugin extends AbstractPlugin
{
  public function checkUpdated($installedVersion = null)
  {
      // 최신버전이 적용된 상태인지 체크합니다.
  }

  public function update()
  {
      // 플러그인의 최신버전을 적용하기 위한 코드를 작성합니다.
  }

}
```

`checkUpdated` 메소드는 현재 XE에 적용된 플러그인의 버전을 파라메터로 받습니다.

사이트 관리 페이지 추가하기

XE는 사이트 관리자 또는 관리 등급을 가진 회원만 접근할 수 있는 '사이트 관리 영역'을 가지고 있습니다. 사이트 관리 영역은 사이트 관리에 필요한 다양한 '사이트 관리 페이지'들로 구성되어 있습니다. 사이트 관리 영역은 일반적으로 `http://<도메인>/settings` 로 접근할 수 있습니다.

플러그인도 사이트 관리 영역에 플러그인을 위한 관리페이지를 추가할 수 있습니다.

사이트 관리 페이지를 추가할 때에는 크게 세가지 작업이 필요합니다.

- 페이지(라우트) 등록
- 사이트 관리 메뉴 등록 및 페이지와 연결
- 사이트 관리 권한 등록 및 페이지와 연결

세가지 작업에 대하여 자세히 살펴보겠습니다.

페이지 등록(라우트 등록)

관리 페이지를 등록하는 것은 결국 라우트를 등록하는 것을 뜻합니다. 관리 페이지를 위한 라우트를 등록하는 방법은 일반 웹페이지의 라우트를 등록하는 방법과 크게 다르지 않습니다. 다만, `Route::group` 메소드 대신 `Route::settings` 메소드를 사용하면 됩니다.

```
Route::settings($uri, function() {
  Route::get('/', ...);
  Route::post('/', ...);
});
```

사이트 관리 영역에 속하는 모든 페이지의 URL은 모두 첫번째 세그먼트로 `settings` 를 가지게 됩니다. 라우트를 등록하는 코드는 플러그인 클래스의 `boot` 메소드에 등록하십시오.

```
<?php
// plugins/my_plugin/plugin.php
namespace MyPlugin;

use Xpressengine\Plugin\AbstractPlugin;
use Route;
use Presenter;

class Plugin extends AbstractPlugin
{
  public function boot()
  {
      // 사이트 관리 페이지 추가
      // http://<domain>/settings/my_plugin url로 접근 가능
      Route::settings(static::getId(), function() {
        Route::get('/', function(){
          return Presenter::make(static::view('views.settings');
        }
      });
  }

}
```

사이트 관리 페이지들이 출력될 때에는 사이트 관리 영역용 테마를 적용한 후 출력돼야 합니다. `Presenter` 를 사용하여 결과를 반환하십시오. 자동으로 사이트 관리 영역용 테마가 적용됩니다.

메뉴 등록

사이트 관리 영역의 화면 좌측에는 관리 메뉴 트리가 출력됩니다. 플러그인은 이 트리에 메뉴를 추가할 수 있습니다. `XeRegister` 파사드의 `push` 메소드를 사용하여 `settings/menu`에 메뉴 정보를 등록합니다.

```
\XeRegister::push('settings/menu', $menuId, $menuInfo);
```

아래 코드는 사이트 관리 메뉴의 `컨텐츠` 메뉴 하위에 서브 메뉴로 `게시판` 메뉴를 추가하는 예제입니다.

```
\XeRegister::push('settings/menu', 'contents.board', [
    'title' => '게시판',
    'display' => true,
    'description' => '',
    'ordering' => 2000,
]);
```

첫번째 파라미터는 항상 `settings/menu`를 지정하십시오.

두번째 파라미터는 메뉴의 아이디인 동시에 메뉴의 부모 메뉴를 지정하는 역할을 합니다. 부모메뉴의 아이디와 현재 메뉴의 아이디를 점(.)을 사용하여 연결해주십시오. 위 예제의 경우 `contents` 메뉴 하위에 `board` 메뉴를 추가합니다.

세번째 파라미터는 메뉴의 상세정보를 담은 배열입니다.

- `title`은 메뉴가 출력될 때 사용하는 메뉴의 이름입니다.
- `display`가 `true`이면 메뉴가 메뉴 트리에 출력됩니다. `false`의 경우 메뉴 트리에 출력되지 않습니다. 예를 들어 `회원 정보 수정` 과 같은 메뉴는 메뉴 트리에는 출력되지 않는 숨김 메뉴입니다. 사이트 관리페이지의 상단에는 빵조각(breadcrumb)으로 현재 관리페이지의 위치를 표시해주는데, 이때 숨김 메뉴를 등록해 놓으면 유용합니다.
- `description`은 메뉴에 대한 설명입니다. 사이트 관리페이지의 상단에 출력됩니다.
- `ordering`은 메뉴가 출력되는 순서를 지정할 때 사용됩니다. 같은 레벨의 메뉴들이 출력될 때 `ordering`이 작은 메뉴부터 출력됩니다.

페이지에 메뉴 연결하기

사이트 관리 페이지(라우트)를 등록할 때, 페이지에 해당하는 메뉴를 지정할 수 있습니다. 아래 코드는 회원 추가 페이지에 메뉴를 지정하는 예제입니다.

```
// member.create 등록
\XeRegister::push('settings/menu', 'user.create', [
    'title' => '새 회원 추가',
    'description' => '신규회원을 추가합니다.',
    'display' => false,
    'ordering' => 200
]);
```

```
// settings_menu 지정
Route::settings('user', function () {
    Route::get('create', [
            'as' => 'settings.member.create',
            'uses' => 'Member\Settings\UserController@create',
            'settings_menu' => 'user.create'
    ]);
});
```

라우트를 등록할 때, 두번째 파라미터 배열의 `settings_menu` 필드에 메뉴 아이디를 지정하면 됩니다. 위 예제의 경우 `member.create` 메뉴를 라우트와 연결하고 있습니다.

관리 권한

기본적으로 사이트의 최고관리자(super)는 모든 관리 페이지에 접근할 수 있습니다. 하지만, 최고관리자가 아닌 관리자(manager) 등급의 회원이더라도 선택적으로 사이트 관리 페이지에 접근할 수 있도록 지정할 수 있습니다.

권한 등록

권한을 등록하는 방법은 앞서 설명한 메뉴 등록 방법과 유사합니다. `XeRegister` 파사드의 `push` 메소드를 사용합니다.

```
\XeRegister::push('settings/permission', $permissionId, $permissionInfo);
```

아래 코드는 '회원 생성'이라는 권한을 등록하는 예제입니다.

```
\XeRegister::push('settings/permission', 'user.create', [
    'title' => '회원 생성',
    'tab' => '회원관리'
]);
```

첫번째 파라미터는 항상 `settings/permission`을 지정하십시오.

두번째 파라미터는 권한의 아이디입니다.

세번째 파라미터는 권한의 상세정보를 담은 배열입니다.

- `title`은 권한의 이름입니다. 권한 관리 페이지에서 권한을 표시할 때 사용됩니다.
- `tab`은 권한이 속할 그룹을 이름을 지정합니다. 권한 관리 페이지에서는 동일한 `tab`을 가진 권한들을 그룹지어 출력합니다.

위와 같이 권한을 등록해 놓으면, '사이트 관리 > 설정 > 관리페이지 권한 설정' 페이지에 등록한 권한이 표시됩니다. 사이트 관리자는 이 페이지에서 특정 사용자에게 권한을 부여할 수 있습니다.

페이지에 권한 지정하기

등록한 권한을 관리 페이지(라우트)에 연결하는 과정도 사이트 관리 메뉴를 지정했던 방식과 유사합니다. `settings_permission` 필드에 권한의 아이디를 지정하십시오.

```
Route::settings('user', function () {
    Route::get('create', [
           'as' => 'settings.member.create',
           'uses' => 'Member\Settings\UserController@create',
           'settings_menu' => 'user.create',
           'settings_permission' => 'user.create'
    ]);
});
```

위 예제에서는 '회원 생성' 권한을 회원 추가 페이지에 지정하고 있습니다.

어떤 회원이 회원 추가 페이지에 접근할 경우, XE는 먼저 회원 추가 페이지에 지정된 권한을 찾습니다. 위의 경우 회원 추가 (`user.create`) 권한을 찾게 됩니다. 그 다음 XE는 로그인한 회원에 권한이 부여되어 있는지 검사합니다. 만약 권한이 부여되어 있지 않은 회원이라면 오류를 출력하게 됩니다.

컴포넌트 등록

XE는 다양한 타입의 컴포넌트가 있습니다. 여러분이 컴포넌트를 가지는 플러그인 개발을 시작한다면, 먼저 뼈대만 갖춘 컴포넌트를 작성하고, 플러그인을 통해 작성한 컴포넌트를 XE에 등록해야 합니다. 타입이 다른 컴포넌트라도 XE에 등록하는 방법은 모두 동일합니다.

컴포넌트 아이디

각각의 컴포넌트는 모두 고유의 아이디를 가지고 있어야 합니다. XE는 컴포넌트의 아이디를 통해 등록된 컴포넌트를 효과적으로 관리합니다. 또, 컴포넌트 아이디는 컴포넌트 타입을 구분짓는 역할도 병행합니다. 각 컴포넌트 타입마다 아이디를 지정하는 규칙이 아래와 같이 정해져있습니다.

컴포넌트 타입	아이디 규칙	예제
테마	`theme/<plugin_name>@<pure_id>`	`theme/alice@alice`
모듈	`module/<plugin_name>@<pure_id>`	`module/myplugin@board`
위젯	`widget/<plugin_name>@<pure_id>`	`widget/myplugin@content`

스킨	`<skin_target_id>/skin/<plugin_name>@<pure_id>`	`user/profile/skin/social_login@default`
		`module/myplugin@board/skin/board@gallery`
		`widget/xpressengine@content/skin/myplugin@content`

UI 오브젝트	`uiobject/<plugin_name>@<pure_id>`	`uiobject/myplugin@formSelect`
다이나믹필드	`FieldType/<plugin_name>@<pure_id>`	`FieldType/myplugin@Text`
다이나믹필드스킨	`<dynamic_field_id>/FieldSkin/<plugin_name>@<pure_id>`	`FieldType/myplugin@Text/FieldSkin/fooplugin@TextDefa`
에디터	...	...
에디터툴	...	...

composer.json을 사용하여 등록하기

플러그인의 필수 구성 파일인 `composer.json` 을 통해 컴포넌트를 등록할 수 있습니다.

`composer.json` 항목의 `extra > xpressengine > component` 항목에 아래의 형식으로 컴포넌트 정보를 기입합니다.

```
"component": {
  "<컴포넌트 아이디>": {
    "class": "<컴포넌트 클래스명>",
    "name": "<컴포넌트 제목>",
    "description": "<컴포넌트 설명>",
    "screenshot": "<스크린샷 경로>",
  }
}
```

`<컴포넌트 아이디>` 에는 앞서 설명한 컴포넌트 아이디를 적습니다.

`class` 에는 컴포넌트 클래스의 풀네임을 적습니다. 컴포넌트 클래스는 반드시 autoload를 통해 로드 가능한 위치이어야 합니다.

`name` 에는 컴포넌트의 제목을 적습니다.

`description` 에는 컴포넌트에 대한 간략한 설명을 적습니다.

`screenshot` 에는 컴포넌트의 스크린샷 이미지 경로를 적습니다. 스크린샷 이미지는 이미 플러그인 디렉토리 안에 저장되어 있어야 합니다.

Alice 플러그인의 `composer.json` 파일 예제입니다. Alice 플러그인은 하나의 테마 컴포넌트를 등록하고 있습니다.

```
{
  "name": "xpressengine-plugin/alice",
  "description": "This Package is Xpressengine Plugin - Alice Theme.",

  ...

  "extra": {
    "xpressengine": {
      "title": "Alice Theme",
      "icon": "icon.png",
      "component": {
        "theme/alice@alice": {
          "class": "Xpressengine\\Plugins\\Alice\\Theme\\Alice",
          "name": "Alice",
          "description": "The First Theme for XpressEngine3",
          "screenshot": "/plugins/alice/screenshots/main.png",
        }
      }
    }
  },

  ...

}
```

plugin.php를 사용하여 등록하기

`component.json` 을 사용하지 않고, PHP 코드를 사용하여 컴포넌트를 등록할 수도 있습니다. 만약 컴포넌트가 항상 등록될 필요가 없다면, 즉, 특정 조건에 의해 동적으로 등록하고 싶다면 이 방법을 사용할 수 있습니다.

활성화 된 플러그인는 XE가 실행될 때마다 그 플러그인의 플러그인 클래스(`plugin.php` 파일에 존재)의 `boot` 메소드가 실행됩니다. `boot` 메소드에서 플러그인이 가지고 있는 컴포넌트를 등록시킬 수 있습니다. 아래 코드를 사용하십시오.

```
// myplugin/plugin.php

  ...

  public fucntion boot()
  {
    // 컴포넌트의 클래스명을 파라미터로 전달
    XePlugin::addComponent($componentClass);

  }

  ...
```

`component.json` 을 사용하지 않고 직접 컴포넌트를 등록하는 경우, 컴포넌트 클래스는 반드시 컴포넌트 아이디를 직접 지정하고 있어야 합니다. 아래와 같이 `$id` 프로퍼티를 직접 지정하십시오.

```
public static $id = 'component_type/my@component';
```

설정(config)

XE에서 제공하는 config는 laravel에서 제공되는 config와는 다르게 데이터베이스에 정보를 담으며 계층을 가지고 상위 정보를 참조합니다. 이때 각 계층은 "."(dot) 으로 구분합니다.

등록

config 는 해당 설정을 나타내는 이름과 그에 매칭되는 배열을 통해 등록됩니다.

```
XeConfig::add('foo', ['var1' => 'a', 'var2' => 'b']);
XeConfig::add('foo.bar', ['var1' => 'A']);
XeConfig::set('foo.baz', ['var2' => 'B']);
```

`add` 는 새로운 설정정보를 등록하는 메서드입니다. 만약 같은 이름의 설정이 이미 등록되어져 있는 경우 Exception이 발생하게 됩니다.

`set` 은 기존 설정의 존재 유무와 상관 없이 값을 등록합니다. `set` 의 경우 기존에 같은 이름의 설정이 존재하는 경우 메서드 호출 시 전달된 배열의 키에 해당하는 값만을 갱신해 줍니다.

```
XeConfig::add('foo', ['var1' => 'a', 'var2' => 'b']);
XeConfig::set('foo', ['var1' => 'A']);
echo XeConfig::getVal('foo.var1'); // A
echo XeConfig::getVal('foo.var2'); // b
```

설정을 등록하는 방법은 `add` , `set` 이외에도 `put` , `modify` , `setVal` 이 있습니다.

`put` 은 기존에 같은 이름을 가지는 설정이 먼저 등록되어 있어야 정상적으로 동작합니다. 또한 `set` 과는 다르게 기존의 설정들을 메서드 호출시 전달된 배열 값으로 모두 대체합니다.

```
XeConfig::add('foo', ['var1' => 'a', 'var2' => 'b']);
XeConfig::put('foo', ['var1' => 'A']);
echo XeConfig::getVal('foo.var1'); // A
echo XeConfig::getVal('foo.var2'); // null
```

`modify` 는 config 객체를 통해 수정하는 기능입니다.

```
$config = XeConfig::get('foo');
$config->set('var1', 'A');
XeConfig::modify($config);
```

`setVal` 은 설정의 특정 항목의 값만 등록하는 메서드입니다.

```
XeConfig::setVal('foo.var1', 'A');
```

조회

등록된 설정을 조회할때는 특정 설정키에 해당하는 값을 받거나, config 객체를 반환 받은후 값을 조회하는 방법이 있습니다.

```
XeConfig::add('foo', ['var1' => 'a', 'var2' => 'b']);
echo XeConfig::getVal('foo.var1'); // a
// 또는
$config = XeConfig::get('foo');
echo $config->get('var1'); // a
```

만약 등록된 설정이 없는 경우 특정값을 반환 받고 싶으면 다음 인자에 값을 전달하면 됩니다.

```
XeConfig::add('foo', ['var1' => 'a', 'var2' => 'b']);
echo XeConfig::getVal('foo.var3', 'c'); // c
// 또는
$config = XeConfig::get('foo');
echo $config->get('var3', 'c'); // c
```

계층조회

config는 계층을 가지고 있고 이를 이용하여 존재하지 않는 설정은 부모에 해당하는 설정을 조회하여 결과를 반환합니다.

```
XeConfig::add('foo', ['var1' => 'a', 'var2' => 'b']);
XeConfig::add('foo.bar', ['var1' => 'A']);
echo XeConfig::getVal('foo.bar.var2'); // b
```

> 설정을 조회할때 두번째 인자로 default 값을 전달하는 경우 최상위 설정에서도 해당하는 결과가 없는 경우에 default 값이 반환되어 집니다.

문서(document)

사이트에서 생산되는 게시판, 블로그, 댓글등 다양한 형태의 컨텐츠를 저장할 저장소를 제공합니다. DocumentHandler 는 Document, Revision 모델을 이용해서 컨텐츠 변경이력을 관리할 수 있는 기능을 제공하고 저장소의 부하분산을 위한 Division(저장소 분할) 저장소를 제공하합니다. 이러한 기능은 instanceId 에 따라 설정을 추가해서 관리할 수 있습니다.

또한 Document 모델은 DynamicField 를 지원합니다.

DocumentHandler, InstanceManager, ConfigHandler 는 Interception Proxy로 만들어 사용되므로 인터셉트 할 수 있습니다. (인터셉트하기)

기본 사용법

`XeDocument` 파사드로 DocumentHandler 를 사용합니다. 문서 등록, 수정, 삭제는 DocumentHandler를 이용해서 처리하고 문서 조회는 Document 모델을 직적 사용합니다. 단, 인스턴스를 생성해 사용할 경우 Document 모델에 대한 설정을 모델에 포함시키기 위해 `XeDocument::getModel($instnaceId)` 를 사용하기를 권장합니다.

모든 문서 가져오기

```
$doc = Document::all();
```

Primary Key를 통해서 하나의 레코드 가져오기

```
$doc = Document::find($documentId);
var_dump($doc->content);
```

인스턴스 생성을 통해 등록된 문서 가져오기

메뉴를 통해 인스턴스가 생성되었거나 또는 어떤 방식으로든 `InstanceManager` 를 이용해 인스턴스를 만들어 설정을 사용하는 경우에는 모델을 직접 사용하지 말고 `DocumentHandler` 를 통해 획득한 모델을 사용하도록 해야합니다.

```
$model = XeDocument::getModel($instanceId);
// 10개의 문서를 가져옵니다.
$model->paginate(10);
```

이렇게 획득한 모델은 설정에 대한 정보를 포함하고 있으며 `revision` , `division` 에 대한 처리 및 Database Proxy를 이용하는 기능인 DynamiField 에 대한 처리가 가능합니다.

문서 등록

```
$params['instanceId'] = $instanceId;
$params['title'] = '제목';
$params['content'] = '내용';
$inputs['userId'] = Auth::user()->getId();
$inputs['writer'] = Auth::user()->getDisplayName();

XeDocument::add($params);
```

문서 등록은 Document 모델을 직접 사용하지 않고 DocumentHandler 를 통해서 처리합니다. 이는 문서 저장할 때 인터셉트할 수 있는 포인트를 제공하며 Document 가 입력값 검사 및 `revision` 을 처리합니다.

수정하기

Document를 통해 문서를 직접가져오는 경우는 아래와 같이 `XeDocument::setModelConfig()` 를 이용해서 모델에 설정을 삽입해야 `DocumentHandler` 에서 설정에 따른 처리를 이상없이 수행할 수 있습니다.

```
$doc = Document::find($documentId);
$doc->title = '제목 수정';

// Document 모델($doc)에 설정 삽입
XeDocument::setModelConfig($doc, $doc->instanceId);
XeDocument::put($doc);
```

일반적인 상태에서 문서를 찾을 때 라이프 사이클 상에서 instanceId 를 확보할 수 있다고 가정한다면 아래 형식의 코드를 사용할 수 있습니다.

```
$doc = XeDocument::getModel($instanceId)->find($documentId);
$doc->title = '제목 수정';

XeDocument::put($doc);
```

삭제하기

```
$doc = Document::find($documentId);

// Document 모델($doc)에 설정 삽입
XeDocument::setModelConfig($doc, $doc->instanceId);
XeDocument::remove($doc);
```

```
// '937c2ec7' 은 인스턴스 아이디
$doc = XeDocument::getModel($instanceId)->find($documentId);
XeDocument::remove($doc);
```

인스턴스

관리자 > 사이트 메뉴에서 게시판을 만들 때 게시판 플러그인은 Document 서비스 를 사용하기 위해 Document 인스턴스를 만듭니다. Document 를 사용하기 위한 설정 및 그에 필요한 주변 요소를 생성해서 실제 사용가능한 형태로 만드는 것입니다.

인스턴스 생성

```
$params['revision'] = false;
$params['division'] = true;
XeDocument::createInstance($instanceId, $params);
```

Document 에 설정 요소를 `$params` 로 전달합니다.

인스턴스 제거

```
XeDocument::destroyInstance($instanceId);
```

생성된 인스턴스를 제거하며 인스턴스 아이디로 저장된 문서를 삭제합니다. 이것은 게시판 삭제와 같은 처리를 할 때 사용됩니다.

Document 모델

데이터를 저장할 때 사용하는 이것은 DynamicQuery 를 처리하기 위해 Config를 등록하는 기능과 문서를 처리할 때 유용한 다양한 메소드를 지원합니다.

테이블 부하분산

`Document::setConfig()` 하는 과정에서 테이블 부하분산 처리를 위해 실제 사용할 테이블 이름을 변경할 수 있도록 지원합니다. 이 기능은 `DocumentHandler::setModelConfig()` 할 때 처리됩니다.

다양한 상태값

상태를 표시하기 위한 다양한 컬럼을 갖고 있습니다.

- status : 문서의 상태를 저장합니다. (휴지통, 임시저장, 비공개, 공개, 공지)
- approved : 승인에 대한 설정을 저장합니다. (거절됨, 기다림, 허용됨)
- published : 발행에 대한 설정을 저장합니다. (거절됨, 기다림, 예약발행, 발행됨)
- display : 보여지는 상태에 대한 설정을 저장합니다. (숨김, 비밀, 보여짐)

이 상태값들을 복합적으로 사용하여 플러그인 자체적으로 다양한 방식의 상태를 표현할 수 있습니다. 그 중에 일반적으로 사용되는 `보여짐` , `숨김` , `발행` 등 다양한 형태의 상태값들을 제안하기 위한 메소드들이 추가되어 있습니다.

각 설정은 숫자로 정의되어 있으며 between 쿼리를 이용해서 더 다양한 설정을 사용할 수 있기를 바라고 있습니다. 서드파티 플러그인에서 Document 모델을 확장하는 `ExtendDocument` 모델을 만들고 상태를 추가하여 더 다양한 형태의 문서 상태 처리 방법이 제공될 수 있기를 바랍니다.

format

문서를 출력할 때 어떤 형태의 포멧으로 처리해야할 지 알려주기 위해서 글 저장할 때 format 을 등록하도록 하고 있습니다. 이것은 스마트 에디터를 사용하거나 혹은 마크다운으로 작성된 문서를 구분하고 저장된 내용을 출력할 때 어떤 방식으로 처리해야할 지 결정할 때 유용하게 사용될 것입니다. 현재의 포멧은 HTML 형식만 정의하고 있습니다.

reply

```
// 상위 글
$doc = XeDocument::getModel($instanceId)->find($documentId);

// parentId 등록
$params['parentId'] = $doc->id;
$params['instanceId'] = '937c2ec7';
$params['title'] = '제목';
$params['content'] = '내용';
$inputs['userId'] = Auth::user()->getId();
$inputs['writer'] = Auth::user()->getDisplayName();

XeDocument::add($params);
```

답글 처리를 위해 `Document::setReply()` 를 제공합니다. 어떤 문서의 하위로 글을 작성하려고 한다면 해당 모델에 `parentId` 를 놓고 저장하면 됩니다. `Document::setReply()` 의 실행은 `/app/Providers/DocumentServiceProvider.php` 의 `boot()` 에서 `Illuminate\Database\Eloquent\Model` 의 `creating` 이벤트 리스너를 등록해서 자동으로 처리됩니다.

Division 지원

테이블 분리 기능을 처리하기 위해 Document 모델은 문서 등록, 수정, 삭제할 때 설정 값을 확인하고 추가적인 처리를 진행합니다. `Illuminate\Database\Eloquent\Model` 에서 제공하는 `performInsert()` , `performUpdate()` , `performDeleteOnModel()` 기능을 이용해 데이터베이스에 처리될 때 추가적인 코드를 처리합니다.

관계

Document 모델은 User(회원)에 대한 관계만 제공합니다. 더 많은 릴레이션은 제공하지 않습니다. 더 많은 모델의 릴레이션을 사용하기 위해서는 게시판 플러그인의 Board 모델과 같이 Document 모델을 확장해서 사용하는것을 권장합니다.

키생성기(keygen)

XE는 단순하고 유추하기 쉬운 기존의 데이터베이스를 이용한 `auto_increment`를 벗어나, 유일성을 보장하면서 유추하기 어려운 키를 생성할 수 있는 기능을 제공합니다. 키의 생성은 UUID 방식으로 `ramsey/uuid`를 이용해 사용자가 간편하게 고유키를 얻을 수 있도록 해줍니다.

설정

키생성기의 설정은 `config/xe.php`의 `uid` 항목에 있습니다.

```
'uid' => [
  'version' => 4,
  'namespace' => 'xe'
]
```

`version`은 사용할 UUID의 버전을 말합니다. `1, 3, 4, 5` 네가지 값 중 하나를 입력할 수 있고, 기본은 `4` 버전을 사용합니다. `namespace`는 `3, 5` 버전에서 사용될 네임스페이스를 가리킵니다.

> 각 버전의 차이는 위키등에서 확인하실 수 있습니다.

사용

키의 생성은 `generate` 메서드를 통해 이루어 집니다.

```
$keygen = app('xe.keygen');
$id = $keygen->generate();
```

생성된 키는 `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`와 같은 형식을 가집니다.

모델

XE에서는 `Illuminate\Database\Eloquent\Model`을 상속한 `DynamicModel` 모델을 이용하는 경우 키생성기를 통해 고유키를 제공합니다. 키생성기에서 제공하는 키를 사용하고자 한다면 해당 모델의 `incrementnig`을 비활성해야 합니다.

```
use Xpressengine\Database\Eloquent\DynamicModel;

class MyModel extends DynamicModel
{
  public $incrementing = false;

  ...
}
```

메뉴/모듈(menu, module)

이벤트 및 인터셉션

플러그인 개발자가 플러그인에서 필요한 기능을 구현하기 위해서는 XE가 실행되는 여러 시점에 끼어들어 XE의 행동을 바꾸거나 추가적인 행동을 할 수 있어야 합니다.

예를 들어, 사이트에 새로운 회원이 가입할 때, 가입한 회원에게 가입축하 메일을 보내는 기능을 플러그인으로 만들 수 있습니다. 이 기능을 구현하려면 XE가 회원가입을 처리할 때, 플러그인이 끼어들어 메일을 전송하는 코드를 실행할 수 있어야 합니다.

이러한 '끼어들기'를 일반적으로 'hook' 또는 'event'라고 칭합니다. XE에서는 라라벨에서 기본적으로 제공하는 'event' 방식과 XE에서 새롭게 제공하는 'interception' 방식으로 '끼어들기'를 지원합니다.

이벤트

이벤트는 옵저버 패턴으로 구현됩니다. 특정 이벤트에 대하여 리스너를 등록해 놓으면, 이벤트가 발생했을 때, 등록한 리스너가 차례대로 실행됩니다.

이벤트 리스너 등록

이벤트가 발생했을 때 실행될 리스너는 `Event` 파사드나 `Illuminate\Contracts\Events\Dispatcher` contract의 구현을 이용해 등록할 수 있습니다

```
/**
 * Register any other events for your plugin
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    $events->listen('event.name', function ($foo, $bar) {
        //
    });
}

// or

public function boot()
{
    \Event::listen('event.name', function ($foo, $bar) {
        //
    });
}
```

이벤트 리스너를 클래스로 정의

클로저 형식의 이벤트 리스너 대신 클래스 형식으로 이벤트 리스너를 정의할 수 있습니다.

```
Event::listen('auth.login', 'LoginHandler');
```

기본적으로 이벤트가 발생했을 경우, 이벤트 리스너 클래스의 `handle` 메소드가 호출됩니다.

```
class LoginHandler {

    public function handle($data)
    {
        //
    }
}
```

`handle` 메소드 대신 다른 메소드를 사용할 수도 있습니다. 리스너 등록시 메소드명을 아래와 같이 기입하십시오.

```
Event::listen('auth.login', 'LoginHandler@onLogin');
```

와일드카드 이벤트 리스너

`*` 를 와일드카드로 사용하여 리스너를 등록하면 동일한 리스너에서 여러 개의 이벤트에 대응 할 수 있습니다. 와일드카드 리스너는 전체 이벤트 데이터 배열을 하나의 인자로 받아들입니다:

```
$events->listen('event.*', function (array $data) {
    //
});
```

리스너 우선순위 지정

리스너의 우선순위를 지정할 수 있습니다. 우선순위가 높은 리스너가 먼저 실행되며, 동일한 우선순위일 경우, 먼저 등록된 리스너가 먼저 실행됩니다.

```
Event::listen('auth.login', 'LoginHandler', 10);

Event::listen('auth.login', 'OtherHandler', 5);
```

이벤트 발생시키기

이벤트를 발생시키기려면 `Event` 파사드의 `fire` 메소드를 사용하십시오.

```
$args = [...];
Event::fire('event.name', $args)
```

이벤트 전달 중단하기

경우에 따라서 이벤트가 다른 리스너에게 전달되는 것을 중단 하기를 원할 수도 있습니다. 이러한 경우에는 리스너가 `false` 를 반환하면 됩니다.

```
Event::listen('auth.login', function($event)
{
    // Handle the event...

    return false;
});
```

XE 이벤트 목록

아래 목록은 XE에서 제공하는 다양한 코어 레벨의 이벤트의 목록입니다. 플러그인의 `boot` 메소드에서 아래 이벤트가 발생할 때 실행할 리스너를 등록하여 쓸 수 있습니다.

Event	Parameter(s)
auth.attempt	$credentials, $remember, $login
auth.login	$user, $remember
auth.logout	$user
cache.missed	$key
cache.hit	$key, $value
cache.write	$key, $value, $minutes
cache.delete	$key
connection.{name}.beginTransaction	$connection
connection.{name}.committed	$connection
connection.{name}.rollingBack	$connection
illuminate.query	$query, $bindings, $time, $connectionName
illuminate.queue.after	$connection, $job, $data
illuminate.queue.failed	$connection, $job, $data
illuminate.queue.stopping	null
mailer.sending	$message
router.matched	$route, $request
composing:{view name}	$view
creating:{view name}	$view
xe.editor.option.building	$editor
xe.editor.option.builded	$editor
xe.editor.render'	$editor
xe.editor.compile'	$editor

인터셉션

이벤트 방식을 사용하는 경우, 특정 시점에 이벤트를 발생시키기 위해 이벤트를 fire하는 코드를 일일이 작성해주어야 합니다. 만약 게시판 플러그인에서 게시판에 글이 처음 등록될 때, 또는 수정, 삭제 될 때마다 다른 플러그인에서 '끼어들기'를 할 수 있도록 해주려면 게시글의 등록, 수정, 삭제에 대한 이벤트를 fire하는 코드를 3군데에 작성해주어야 합니다. 이벤트 방식 대신 인터셉션을 사용하면 이벤트를 fire하는 코드를 매번 작성하는 번거로움을 줄여줄 수 있습니다.

XE3는 인터셉션을 구현하기 위하여 AOP를 사용하였습니다. AOP는 Aspect Oriented Programming의 약어입니다.

어드바이저(리스너) 등록

인터셉션 방식에서 어드바이저는 이벤트 방식의 리스너와 비슷합니다. 만약 사이트에 회원이 가입할 때, 메일 전송 코드가 실행되도록 하려면 `intercept` 함수를 사용하여 어드바이저를 등록할 수 있습니다.

```
// 가입 축하 메일 보내기 등록
intercept('XeUser@create', 'welcome_mail::send_mail', function($createUser, array $data) use ($mailer) {

    // 회원가입 코드를 실행
    $member = $createUser($data);

    // 메일 전송
    $mailer->sendWelcomeMail($member->email, $member->getDisplayName);

    // 회원가입 처리 결과 반환
    return $member;
});
```

`intercept` 함수의 원형은 아래와 같습니다.

```
intercept($pointCut, $name, Closure $advice)
```

첫번째 파라메터 `$pointCut` 은 이벤트 서비스의 이벤트명과 동일한 역할을 합니다. 즉, '끼어들기'를 할 대상 메소드를 칭하며, 위의 예에서는 `XeUser@create` 에 해당합니다. 이는 `XeUser` 의 `create` 메소드를 뜻합니다.

AOP에서 '끼어들기'를 하는 주체를 어드바이저(Advisor)라고 합니다. 이벤트 서비스의 리스너와 같은 개념입니다. 두번째 파라메터인 `$name` 은 이 어드바이저의 '이름'입니다. 원하는 이름을 직접 지정하십시오. 단 이 이름은 다른 어드바이저의 이름과 중복되지 않아야 합니다. 중복을 회피하기 위하여 해당 플러그인의 아이디(디렉토리명)를 이름의 접두사로 사용하십시오. `welcome_mail` 이 플러그인 아이디에 해당합니다. ::을 사용하여 접두사를 연결하십시오.

세번째 파라메터 `$advice` '끼어들기'를 한 후 실행될 코드입니다. 클로저 형식으로 입력해야 합니다. 이 클로저 내부에서는 반드시 타겟 메소드(회원가입메소드)를 호출해 주어야 합니다. 타겟 메소드는 클로저의 첫번째 파라메터(`$createUser`)를 사용하여 호출할 수 있습니다. 타겟 메소드를 호출하기 전이나 후에 원하는 코드를 추가하여 실행시킬 수 있습니다. 위의 예에서는 회원가입 처리를 한 후에 해당 회원에게 메일을 전송하는 코드가 추가되어 있습니다.

이 클로저에 대해 자세히 설명하면, 이 클로저의 첫번째 파라메터는 타겟 메소드입니다. 위의 예에서 `$createUser` 가 이에 해당합니다. 클로저 내부에서 항상 이 타겟 메소드를 호출해주어야 합니다. 또, 클로저는 타겟 메소드의 리턴값을 다시 리턴해야 합니다. 물론 필요에 따라 리턴값을 변경해도 됩니다. 두번째 파라메터부터는 대상 메소드가 호출될 때 받은 파라메터를 그대로 전달받습니다. 위의 예에서는 `$data` 에 해당하며, 가입할 회원의 정보가 담겨있습니다. 타겟 메소드에 따라 파라메터의 수와 내용이 달라집니다.

```
function($createUser, array $data) {

  // 항상 타겟 메소드(첫번째 파라메터)를 호출해주어야 한다.
  $member = $createMember($data);

  // 항상 타겟 메소드의 반환값을 다시 반환해야 한다.
  return $member;
}
```

어드바이저 간의 우선순위 지정

다수의 어드바이저가 등록돼 있다면, 어드바이저 간의 실행순서가 중요할 수 있습니다. `intercept` 함수를 호출시 다른 어드바이저와의 우선순위를 지정할 수 있습니다.

```
// email_checker::check가 실행된 후 실행.
intercept('XeUser@create', ['welcome_mail::send_mail' => 'foo@bar'], $closure);
```

우선순위는 두번째 파라미터에 배열형태로 지정할 수 있습니다. 위의 `intercept` 함수는 배열의 키 이름인 `welcome_mail::send_mail` 어드바이저를 등록하고 있습니다. 동시에 배열의 값(value)을 이용해여 선행되어야 하는 어드바이저 `foo@bar` 를 지정하고 있습니다.

위 코드에서 우선순위 관계 더욱더 명시적으로 작성할 수 있습니다. 중첩된 배열의 키에 'before'를 사용하십시오.

```
// 명시적으로 before 사용
intercept(
  'XeUser@create',
  ['welcome_mail::send_mail' => ['before' => 'foo@bar'],
  $closure
);

// 다수의 어드바이저와 우선순위 지정
intercept(
  'XeUser@create',
  ['welcome_mail::send_mail' => ['before' => ['foo@bar', 'foo@baz']],
  $closure
);
```

반대로 `after`를 사용하면, 현재 어드바이저보다 나중에 실행되어야 할 어드바이저를 등록할 수도 있습니다.

```
// after 사용
intercept(
  'XeUser@create',
  ['welcome_mail::send_mail' => ['after' => ['foo@bar']],
  $closure
);
```

인터셉션 서비스는 `intercept` 함수를 통해 등록받은 어드바이저들의 우선순위를 파악한 후, 순서대로 호출해 줍니다. 어드바이저들과 타겟 메소드는 데코레이션 패턴으로 실행됩니다. 이는 미들웨어와 거의 동일한 방식입니다.

프록시 생성

위의 회원가입 예제에서 보면, `XeUser@create`를 포인트컷으로 지정하고 있습니다. 위 예제의 코드는 `XeUser` 파사드 (`\Xpressengine\User\UserHandler`)의 `create` 메소드가 실행될 때, 자동으로 호출됩니다.

이렇게 어떤 클래스의 메소드(public 메소드만 해당)가 실행될 때, 자동으로 등록된 어드바이저들이 호출되도록 하려면, 그 클래스 프록시 클래스를 생성하고, 프록시 클래스를 대신 사용해야 합니다.

프록시 클래스는 `XeInterception` 파사드의 `proxy` 메소드를 사용하여 생성할 수 있습니다.

```
// 프록시 클래스 생성
$proxyClass = XeInterception::proxy('\Xpressengine\User\UserHandler', 'XeUser');

// 원본 클래스 대신 프록시 클래스 사용
$userHandler = new $proxyClass();
```

위 코드에서는 원본 클래스인 `\Xpressengine\User\UserHandler` 클래스의 프록시 클래스를 생성합니다. 그리고 원본 클래스 대신 프록시 클래스의 인스턴스를 생성하여 사용합니다. 마지막 파라미터에는 원본 클래스의 별칭(alias)를 지정할 수 있습니다. `UserHandler` 클래스는 `XeUser` 파사드를 통해 많이 사용됩니다. 위의 경우 파사드명을 별칭으로 등록했습니다. `intercept` 함수를 사용할 때, 원본클래스명 뿐만 아니라 별칭을 사용할 수도 있습니다.

아래의 두 코드 모두 사용할 수 있습니다.

```
// 별칭을 사용
intercept('XeUser@create', ...);

// 원본클래스명을 그대로 사용
intercept('Xpressengine\User\UserHandler@create', ...);
```

파일/스토리지(file, storage)

XE의 스토리지는 라라벨이 제공하는 파일 시스템을 기반으로 사용자의 파일을 관리할 수 있는 기능을 제공합니다. 또한 분산저장 기능을 갖추어 필요에 따라 여러개의 파일 저장소에 파일들을 나누어 저장할 수 있습니다.

설정

스토리지와 관련된 설정은 `config/filesystems.php` 에 있습니다. 기본으로는 `local` 드라이버가 지정되어 소스코드가 존재하는 위치의 `storage` 디렉토리에 파일이 저장됩니다.

클라우드 스토리지를 파일 저장소로 사용하고자 하는 경우에는 별도의 패키지가 설치되어야 합니다. S3나 Rackspace를 사용하는 경우 아래의 패키지가 컴포저를 통해 설치되어야 합니다.

- Amazon S3: league/flysystem-aws-s3-v3 ~1.0
- Racspace: league/flysystem-rackspace ~1.0

분산저장

분산저장 기능을 사용하기 위해서는 설정을 변경해야 합니다. 설정파일의 `division` 항목에 `enable` 값을 `true` 로 설정하고, 파일을 저장할 파일 저장소명을 작성해야 합니다.

```
'division' => [
  'enable' => true,
  'disk' => ['local', 's3', 'rackspace']
]
```

설정이 변경되면 스토리지 패키지는 파일이 업로드 될때마다 자동으로 각 파일 저장소로 분산하여 저장하게 됩니다.

> 분산저장의 대상이 되는 파일저장소는 분산저장을 설정하기 이전에 사용가능한 값으로 설정되어야 합니다.

사용

업로드

파일을 서버로 업로드하고자 할 때는 간단하게 `upload` 메서드만 호출하면 됩니다.

```
use XeStorage;
use Xpressengine\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
  public function uploadFile(Request $request)
  {
    XeStorage::upload($request->file('attached'), 'path/to/dir');
  }
}
```

저장하는 파일이 서버에서 특별한 이름을 가지게 하고 싶은 경우 이름을 지정할 수 있습니다.

```
XeStorage::upload($request->file('attached'), 'path/to/dir', 'custom_file_name');
```

그리고 만약 특정 파일저장소에 파일을 저장하고자 한다면 해당 파일저장소를 지정할 수 있습니다.

```
XeStorage::upload($request->file('attached'), 'path/to/dir', null, 's3');
```

단순 업로드를 통한 저장이 아닌 편집등을 거쳐 생성된 결과물을 파일로 저장하려는 경우 `create` 메서드를 사용하면 됩니다.

```
$content = file_get_content('path/to/file');
$content = /* 편집처리 (이미지 섬네일 생성등과 같은) */;
XeStorage::create($content, 'path/to/dir', 'somefile');
```

다운로드

파일 객체를 `download` 메서드를 통해 전달하세요.

```
$file = File::find($id);
XeStorage::download($file);
```

관계와 재사용

XE 스토리지는 파일과 파일을 이용하는 대상과의 관계를 필요로 합니다. 그래서 업로드 후 관계를 맺어주는 작업을 수행해야 합니다.

```
$file = XeStorage::upload($request->file('attached'), 'path/to/dir');
XeStorage::bind($targetId, $file);
```

그리고 이미 업로드되어진 파일은 다른 대상과도 관계를 맺을 수 있습니다.

```
$file = File::find($id);
XeStorage::bind($otherTargetId, $file);
```

만약 파일을 사용하던 대상이 더 이상 해당 파일을 사용하지 않게 된 경우 관계를 끊을 수 있습니다.

```
$file = File::find($id);
XeStorage::unBind($targetId, $file);
```

이때 어떤 대상과도 연결되어지지 않게 될 경우 해당 파일을 삭제할 수 있습니다.

```
XeStorage::unBind($targetId, $file, true);
```

여러개의 파일을 동시에 처리하는 경우 `sync` 메서드를 사용하면 파일을 사용하는 대상과 파일들의 관계를 간편하게 정의할 수 있습니다.

```
$fileIds = [];
foreach ($uploadedFiles as $uploadedFile) {
  $file = XeStorage::upload($request->file('attached'), 'path/to/dir');
  $fileIds[] = $file->id;
}
XeStorage::sync($targetId, $fileIds);
```

위와 같이 관계가 형성되면 대상의 아이디로 대상에 연결된 파일들을 조회하거나 관계를 끊을 수 있습니다.

```
// 대상의 파일 목록 조회
$files = File::getByFileable($targetId);

// 대상의 삭제등의 처리시 파일들의 연결해제 처리
XeStorage::unBindAll($targetId);
```

회원 및 인증 서비스

XE는 회원정보를 저장하고 관리하는 기능을 제공합니다. 일반적인 회원 관리 기능으로는 신규가입, 회원정보 수정, 회원 삭제와 같은 기능이 있습니다. 또, 인증 시도, 로그인, 로그아웃과 같은 인증 관련 기능도 함께 제공합니다.

회원 관리 기능을 정확히 사용하기 위해서는 회원 및 회원의 부가정보들의 데이터 구조를 정확히 파악해야 합니다.

데이터 구조

회원

회원은 사이트에서 가입 및 인증의 대상이 되는 개념으로, 일반적으로 한명의 사람이라고 생각할 수 있습니다. 회원은 사이트 내에서 권한(permission)을 부여받을 수 있는 대상이 될 수 있습니다.

XE는 회원정보를 `\Xpressengine\User\Models\User` 엘로퀀트 모델로 표현하며 `user` 테이블에 저장합니다.

계정

일반적인 회원서비스에서는 회원(user)과 계정(account)을 동일 개념으로 처리합니다. 하지만 XE는 소셜로그인과 같은 외부 계정을 통한 가입/인증을 유연하게 적용하기 위하여 회원과 계정의 개념을 분리했습니다. 회원은 기본계정 이외에 다수의 외부 계정을 가질 수 있으며, 다수의 계정 중 하나로 회원을 인증(authentication)하고 로그인할 수 있습니다.

만약 소셜로그인과 같이 외부 사이트에서 제공하는 회원서비스를 사용하는 회원가입/로그인 기능을 구현하고 싶다면, 회원 계정을 잘 활용하시길 바랍니다.

XE는 계정정보를 `\Xpressengine\User\Models\UserAccount` 엘로퀀트 모델로 표현하며 `user_account` 테이블에 저장합니다.

> XE에서는 회원의 계정정보를 등록(저장)하고 조회, 삭제하는 기능만 제공합니다. 계정정보를 이용한 가입 및 인증은 별도의 플러그인(social_login)을 통해 제공합니다.

이메일

한명의 회원은 다수의 '승인된 이메일'을 소유할 수 있습니다. 만약 회원이 다수의 이메일을 소유하고 있다면, 소유한 이메일 중 하나와 비밀번호를 사용하여 로그인 할 수 있습니다.

다수의 이메일 중 하나는 회원의 대표 이메일로 지정됩니다. 대표 이메일은 회원에게 이메일을 전송하거나 외부에 공개될 때 사용되는 이메일입니다.

XE는 이메일 정보를 `\Xpressengine\User\Models\UserAccount` 엘로퀀트 모델로 표현하며 `user_email` 테이블에 저장합니다.

승인대기 이메일

회원은 하나의 '승인대기 이메일'을 가질 수 있습니다. 신규회원이 가입할 때 입력한 이메일은 승인대기 이메일로 등록됩니다. 승인대기 이메일은 승인된 이메일과 별도의 테이블에 저장되며, 이메일 인증과정을 거친 후에 승인된 이메일로 등록됩니다.

XE는 승인대기 이메일 정보를 `\Xpressengine\User\Models\UserAccount` 엘로퀀트 모델로 표현하며 `user_pending_email` 테이블에 저장합니다.

> 사이트 관리자가 '이메일 인증' 기능을 사용하도록 설정했을 경우, 가입시 등록한 이메일 승인대기 이메일에 등록됩니다. 만약 이메일 인증 기능을 사용하지 않는다면, 가입과 동시에 일반 이메일로 등록됩니다.

회원등급

한명의 회원은 3개의 회원등급(최고관리자, 관리자, 일반회원) 중 하나를 부여받습니다. `최고관리자(super)` 는 사이트 내에서 모든 권한을 가지는 운영자입니다. `관리자(manager)` 는 일반적으로 사이트를 함께 관리하는 부운영자라고 생각할 수 있습니다. `일반회원(member)` 는 사이트에 사용자인 일반적인 회원입니다.

회원등급은 별도의 테이블에 저장되지 않으며, 회원(user) 테이블의 `rating` 컬럼에 저장됩니다.

그룹

회원그룹은 회원을 자유롭게 그루핑할 때 사용합니다. 사이트관리자는 자유롭게 회원그룹을 생성할 수 있고, 회원을 다수의 회원그룹에 소속시킬 수 있습니다. 회원그룹은 권한을 부여받을 수 있는 대상이 될 수 있습니다.

XE는 그룹 정보를 `\Xpressengine\User\Models\UserGroup` 엘로퀀트 모델로 표현하며 `user_group` 테이블에 저장합니다. 또, 회원과 회원이 소속된 그룹의 관계는 `user_group_user` 테이블에 저장됩니다.

회원 관리

회원 및 부가 정보(그룹, 계정, 이메일 등)를 조회하거나 처리할 때에는 `XeUser` 파사드를 사용하십시오. `XeUser` 파사드의 실제 구현은 `\Xpressengine\User\UserHandler` 에 정의되어 있습니다.

회원 조회

회원 아이디로 회원을 조회할 때에는 `find` 메소드를 사용하십시오.

```
$user = XeUser::find($id);
$username = $user->getDisplayName();
```

여러 회원을 조회할 수도 있습니다.

```
$ids = [1,2,3];
$users = XeUser::find($ids);

foreach($users as $user) {
  ...
}
```

다양하고 복잡한 조건(쿼리)으로 회원을 조회할 수도 있습니다. 자세한 사용법은 라라벨 문서를 참조하십시오.

```
// displayName이 'foo'인 회원 조회
$user = XeUser::where('displayName', 'foo')->first();
```

```
// displayName이 foo로 시작하는 모든 회원 조회
$user = XeUser::where('displayName', 'like', '%foo')->get();
```

신규 회원 추가

`XeUser` 는 복잡한 회원 생성 과정을 한번에 처리해주는 `create` 메소드를 제공합니다. `create` 메소드는 입력한 신규회원 정보에 대한 유효성 검사후 신규회원을 생성합니다. 또한 회원의 계정(account), 이메일(email) 정보도 자동으로 추가되며, 소속될 그룹에 대한 정보가 전달되었을 경우, 그룹에 추가시켜주기도 합니다.

```
$data = [
  'displayName' => 'foo',
  'email' => 'foo@email.com',
  //...
];

$newUser = XeUser::create($data);
```

신규 회원의 계정(account) 정보를 같이 등록할 수도 있습니다.

```
$data = [
  'displayName' => 'foo',
  'email' => 'foo@email.com',
  //...
]

$data['account'] = [
  'provider' => 'facebook',
  'token' => '3DIfdkwwfdsie...',
  'id' => 'id of facebook user',
  'data' => '...'
]

$newUser = XeUser::create($data);
```

신규회원이 소속될 회원그룹을 지정할 수도 있습니다.

```
  $data = [
  'displayName' => 'foo',
  'email' => 'foo@email.com',
  'groupId' => [21,23] // 그룹아이디가 21, 23인 그룹에 회원을 소속시킴
]

$newUser = XeUser::create($data);
```

만약, 유효성검사를 생략하거나 부가 정보(account, email, group)를 등록을 하지 않고, 순수하게 회원 정보만 추가하고 싶다면 `XeUser` 파사드 대신, `UserRepository` 를 사용하십시오.

`UserRepository` 는 `XeUser` 파사드를 사용하여 로드할 수 있습니다.

```
// 순수 회원정보만 등록
$data = [
  'displayName' => 'foo',
  'email' => 'foo@email.com',
  'password' => '...'
]

$newUser = XeUser::users()->create($data);
```

> `UserRepository` 의 `create` 메소드를 곧바로 사용할 때, `password` 를 암호화하지 않고 바로 저장합니다. 먼저 `password` 필드를 직접 암호화하십시오.

회원정보 수정

`XeUser::update` 메소드를 사용하면 회원정보를 변경할 수 있습니다. 유효성 검사 및 프로필 이미지 처리, 소속그룹의 변경도 동시에 처리합니다.

```
$user = XeUser::find(20);
XeUser::update($user, ['displayName' => 'bar']);
```

회원 삭제

`XeUser::leave` 메소드를 사용하면 회원을 삭제(탈퇴)할 수 있습니다. 삭제할 회원의 부가정보(account, email)도 같이 삭제됩니다.

```
$ids = [12,23,34];
XeUser::leave($ids); // 3명의 회원을 탈퇴시킴
```

`UserRepository` 의 `delete` 메소드를 사용하여 회원을 삭제할 수도 있습니다. 단, 이 메소드를 사용하면 삭제될 회원의 부 가 정보는 함께 삭제되지 않습니다.

```
$user = XeUser::find(12);
XeUser::users()->delete($user);
```

회원 계정 관리

계정 조회

회원계정을 조회할 때에는 `UserAccountRepository` 를 사용하십시오. `UserAccountRepository` 는 `XeUser` 파사드를 사용하여 로드할 수 있습니다.

```
$userAccountRepository = XeUser::accounts();
```

회원이 소유한 계정 목록을 조회할 수 있습니다.

```
// 회원 아이디로 계정 정보 조회
$userId = '123';
$accounts = XeUser::accounts()->findByUserId($accountId);
```

위 코드는 아래 코드로 대체할 수도 있습니다.

```
$user = XeUser::find('123');
$accounts = $user->accounts;
```

`User` 의 `getAccountByProvider` 메소드를 사용하면 특정 프로바이더(소셜로그인 벤더)의 계정을 조회할 수 있습니다.

```
$user = XeUser::find('123');
$facebookAccount = $user->getAccountByProvider('facebook');
```

좀 더 복잡한 조건으로 계정을 조회하고 싶다면, `query` 메소드를 사용하십시오.

```
$query = XeUser::accounts()->query();
$account = $query->where('email', 'foo@facebook.com')->first();
```

계정 생성

회원계정을 생성할 때에는 `XeUser` 파사드를 사용할 수 있습니다. `createAccount` 메소드를 사용하십시오.

```
// 기존 회원에 계정정보 추가하기
$user = XeUser::find('123');

$accountData = [
  'email' => 'foo@facebook.com',
  'accountId' => 'facebookId',
  'provider' => 'facebook',
  'token' => '39238432893,
  'data' => '...'
];

$newAccount = XeUser::createAccount($user, $accountData);
```

계정 수정

회원계정을 수정할 때에도 `XeUser` 파사드를 사용할 수 있습니다. `updateAccount` 메소드를 사용하십시오.

```
$user = XeUser::find('123');
$facebookAccount = $user->getAccountByProvider('facebook');

XeUser::updateAccount($facebookAccount, [token => '2197548']);
```

계정 삭제

회원계정을 수정할 때에는 `XeUser::deleteAccount` 메소드를 사용하십시오.

```
XeUser::deleteAccount($facebookAccount);
```

회원 이메일 관리

이메일 조회

회원계정을 조회할 때에는 `UserEmailRepository` 를 사용하십시오. `UserEmailRepository` 는 `XeUser` 파사드를 사용하여 로드할 수 있습니다.

```
$userEmailRepository = XeUser::emails();
```

이메일 주소로 이메일 정보를 조회할 수 있습니다.

```
$email = XeUser::emails()->findByAddress('foo@bar.com');
```

특정 회원이 소유한 모든 이메일을 조회할 수 있습니다.

```
$userId = '123';
$emails = XeUser::emails()->findByUserId($userId);
```

좀 더 복잡한 조건으로 이메일을 조회하고 싶다면, `query` 메소드를 사용하십시오.

```
// 'foo@'로 시작되는 이메일 검색
$query = XeUser::emails()->query();
$emails = $query->where('address', 'like', 'foo@%')->get();
```

승인대기 이메일을 조회할 때에는 `UserEmailRepository` 대신 'PendingEmailRepository'를 사용하십시오.
`PendingEmailRepository` 는 `XeUser` 파사드를 사용하여 로드할 수 있습니다. `UserEmailRepository` 와 동일하게 사용할 수 있습니다.

```
$userEmailRepository = XeUser::pendingEmails();
```

이메일 생성

회원 이메일을 생성할 때에는 `XeUser` 파사드의 `createEmail` 메소드를 사용하십시오.

```
$user = XeUser::find('123');

$emailData = [
  'address' => 'foo@bar.com'
];

$email = XeUser::createEmail($user, $emailData, true);
```

`createEmail` 메소드의 마지막 메소드는 생성하는 이메일의 승인 여부를 지정합니다. `true` 이면 승인된 이메일로 저장되며, `false` 이면 승인대기 이메일로 저장됩니다.

이메일 수정

기등록 된 이메일(또는 승인대기 이메일)을 수정할 수 있습니다. `XeUser::updateEmail` 을 사용하십시오.

```
$email->address = 'foo@bar.com';
XeUser::updateEmail($email);

// or

XeUser::updateEmail($email, ['address'=>'foo@bar.com']);
```

이메일 삭제

이메일(또는 승인대기 이메일)을 삭제할 수 있습니다. `XeUser::deleteEmail` 을 사용하십시오.

```
XeUser::deleteEmail($email);
```

회원 그룹 관리

그룹 조회

...

그룹 생성

```
// 그룹정보 생성
$groupData = [
  'name' => '정회원',
  'description' => '기본회원',
];

$group = XeUser::createGroup($groupData);
```

그룹 수정

```
XeUser::updateGroup($group, ['name' => '기본회원']);
```

그룹 삭제

```
XeUser::deleteGroup($group);
```

인증

등록된 회원은 사이트에 로그인할 수 있습니다. 사이트에 로그인할 때 XE는 입력된 회원정보에 대한 인증(authentication) 과정을 거친후, 세션(session)에 로그인 한 회원의 정보를 등록합니다. 로그아웃을 할 때에는 반대로 세션에 등록했던 회원 정보를 삭제합니다.

`Auth` 파사드를 사용하면 현재 접속한 회원의 로그인, 로그아웃을 비롯한 인증과 관련된 기능을 수행할 수 있습니다.

현재 로그인한 사용자 조회

현재 로그인한 사용자를 조회할 때에는 `user` 메소드를 사용하십시오.

```
$user = Auth::user();
```

또는 request 인스턴스를 사용할 수도 있습니다.

```
$user = request()->user();
```

로그인한 사용자의 아이디(id)만 조회할 수도 있습니다.

```
$loggedUserId = Auth::id();
```

로그인 상태 조회

`check` 메소드를 사용하면 접속 중인 사용자가 로그인을 한 상태인지 조회할 수 있습니다.

```
if (Auth::check()) {
    // The user is logged in...
}
```

로그인

등록된 회원의 인스턴스를 획득하고 있다면, `login` 메소드를 사용하여 해당 회원을 로그인시킬 수 있습니다.

```
$user = XeUser::find('123');

Auth::login($user);
```

해당 회원이 현재와 동일한 환경(단말기의 브라우저)에서 다음에 다시 사이트에 접속했을 때, 자동으로 로그인 처리를 하는 '로그인 유지' 기능을 사용할 수 있습니다. `login` 메소드의 두번째 파라메터로 `true` 를 사용하면, 해당 회원에게 '로그인 유지' 기능을 활성화시킵니다.

```
// Login and "remember" the given user...
Auth::login($user, true);
```

로그인 시도

`login` 메소드는 회원정보의 인증 과정을 거치지 않고, 회원 인스턴스를 통해 바로 로그인 처리를 합니다. 만약 회원의 입력한 정보를 인증한 후 로그인까지 시도하고 싶을 때에는 `attempt` 메소드를 사용하십시오.

`attempt` 메소드를 실행하면 입력된 정보가 등록된 회원정보와 일치하는지 검사합니다. 회원정보가 일치한다면 해당 회원을 로그인시킵니다.

```
if (Auth::attempt(['email' => $email, 'password' => $password])) {
    // Authentication passed...
    return redirect()->intended('dashboard');
}
```

만약 '로그인 유지' 기능을 사용하고싶을 때에는 두번째 파라메터로 `true` 를 전달하십시오.

```
if (Auth::attempt(['email' => $email, 'password' => $password], true)) {
    // The user is being remembered...
}
```

로그아웃

`logout` 메소드를 사용하십시오.

```
Auth::logout();
```

모바일(mobile)

플러그인을 개발할 때, 현재 접근한 기기가 데스크탑인지 모바일 기기인지 알아야 할 때가 있습니다. XE는 자체적으로 접근한 기기의 타입을 판단하고, 이를 사용할 수 있도록 인터페이스를 제공합니다.

기기 타입 강제 지정

XE는 우선 브라우저의 user-agent 정보를 바탕으로 현재 접근한 기기의 타입이 데스크탑인지 모바일인지 검사합니다.

만약 사용자가 접근한 기기의 타입을 강제로 지정하고 싶다면, 요청 URL의 쿼리스트링을 사용해 지정할 수 있습니다. 요청하는 URL에 `?_m=1` 을 추가하면 XE가 현재 접속한 기기를 모바일 기기로 판단하도록 강제 지정할 수 있습니다. 반대로 `?_m=0` 을 추가하면 데스크탑 기기로 판단하도록 강제 지정됩니다. 강제로 지정한 정보는 쿠키에 추가되어 차후 요청시에도 당분간 유지됩니다.

코드상에서 기기 타입을 지정하고 싶다면 직접 쿠키를 저장하면 됩니다. 쿠키 이름을 `mobile` 에 값을 '0' 또는 '1'로 저장하십시오.

```
// 현재 접속 기기를 120분간 모바일 기기로 강제 지정,
\Cookie::queue('mobile', '1', 120);
```

현재 접근한 기기의 타입 검사

XE는 현재 접근한 기기의 타입은 `Request` 인스턴스를 통해 알 수 있습니다.

```
$isMobile = request()->isMobile();
```

`isMobile` 메소드를 사용하면, 현재 설정된 기기의 타입이 모바일인지 검사할 수 있습니다. 만약 접근한 기기 타입이 강제로 지정돼 있다면, 강제로 지정된 타입을 반환합니다.

강제로 지정된 타입과 관계없이 순수하게 브라우저의 user-agent 정보만으로 판단된 기기 타입을 알고싶다면 대신 `isMobileByAgent` 메소드를 사용하십시오.

```
$isMobile = request()->isMobileByAgent();
```

권한(permission)

XE는 관리자 시스템을 통한 동적인 권한검사를 위해 laravel 의 Authorization 기능을 확장하여 제공하고 있습니다.

권한 검사 유형

permission 에서는 권한을 검사하기 위해 5가지 유형의 데이터를 가지고 있습니다.

- `RATING_TYPE` : 사용자의 등급을 나타냅니다. 등급은 `GUEST` , `MEMBER` , `MANAGER` , `SUPER` 로 구분합니다.
- `GROUP_TYPE` : 권한을 가지는 그룹들을 지정합니다.
- `USER_TYPE` : 권한을 가지는 특정 사용자들을 지정합니다.
- `EXCEPT_TYPE` : 권한을 가지지 않는 특정 사용자들을 지정합니다.
- `VGROUP_TYPE` : 권한을 가지는 가상 그룹을 지정합니다.

위 다섯가지 권한중 `EXCEPT_TYPE` 이 가장 우선됩니다. 사용자가 `EXCEPT_TYPE` 에 속하지 않는경우 나머지 중 단 하나의 유형에만 속하여도 권한을 가지는 것으로 판단합니다.

등록

권한을 등록할때는 `Grant` 객체를 이용합니다. `Grant` 객체에 검사하고자 하는 특정 행위를 나타내는 키워드와 각 유형에 맞는 값들을 지정하고, 권한을 구분하는 대상의 이름과 함께 등록하게 됩니다.

```
$grant = new Grant();
$grant->set('show', Rating::MEMBER);
$grant->set('create', Grant::GROUP_TYPE, ['{groupId}']);
$grant->set('create', Grant::EXCEPT_TYPE, ['{userId}']);

app('xe.permission')->register('foo.bar', $grant);
```

하나의 행위에 해당하는 다양한 유형의 데이터를 배열을 이용하여 동시에 설정할 수 있습니다.

```
$grant->set('create', [
  Grant::RATING_TYPE => Rating::MEMBER,
  Grant::GROUP_TYPE => ['{groupId}'],
  Grant::EXCEPT_TYPE => ['{userId}']
]);
```

폼

위와 같은 데이터를 브라우저를 통해 전달 받기위해 ui object 가 제공됩니다.

```
<div>
  {!! uio('permission', $arguments) !!}
</div>
```

이때 전달되는 인자는 배열로 다음과 같은 구조를 필요로 합니다.

```
array(
  'mode' => 'manual', // manual or inherit
  'grant' => [],      // 특정 행위에 해당하는 Grant 객체 등록 값
  'title' => 'create' // 특정 행위를 나타내는 키워드
  'groups' => []      // 전체 그룹
);
```

ui object 는 지정된 행위를 나타내는 키워드와 유형을 조합하여 input 의 이름을 제공합니다. 예를 들어 `create` 라는 행위에 대한 회원등급의 이름은 `createRating` 이 됩니다.

권한 등록을 위한 편의제공

관리자 페이지로부터 전달되는 권한 정보를 형식에 맞게 정제하고 등록하기에 불편함을 느낄 수 있습니다. 그래서 이런 부분에서 자유로워 질 수 있도록 클래스에 삽입되어지는 `PermissionSupport` trait 을 제공합니다.

```
class SomeController
{
  use PermissionSupport;

  public function getSetting()
  {
    $args = $this->getPermArguments('foo.bar', ['create', 'show']);
    ...
  }

  public function postSetting(Request $request)
  {
    $this->permissionRegister($request, 'foo.bar', ['create', 'show']);
    ...
  }

  ...
}
```

권한검사

권한을 검사할때는 laravel 의 Authorization 기능을 이용합니다. 첫번째 인자값에 행위에 대한 키워드를 전달하고 두번째 인자에 권한등록시 사용한 이름으로 생성된 `Instance` 객체를 전달합니다.

```
if (Gate::denies('create', new Instance('foo.bar'))) {
  // throw exception
}
```

카테고리(category)

웹 애플리케이션을 구성하다보면 컨텐츠를 분류하고자 할때가 있습니다. 이때 필요한게 카테고리 기능입니다.

등록

카테고리를 사용하기 위해선 먼저 카테고리의 그룹을 생성해야 합니다. 그리고 해당 카테고리 그룹에 아이템들을 등록시킵니다.

```
// 신규 카테고리 그룹 생성
$category = XeCategory::create();
// 카테고리 그룹에 아이템 추가
$item = XeCategory::createItem($category, ['word' => 'foo']);
```

만약 특정 카테고리 아이템의 자식에 해당하는 아이템을 등록하고자 한다면, 전달하는 값에 `parentId` 항목에 부모에 해당하는 아이디값을 포함하면 됩니다.

```
$item = XeCategory::createItem($category, ['word' => 'foo', 'parentId' => '{parent id}']);
```

트리

카테고리는 트리구조를 가지고 있습니다. 그래서 특정 카테고리 아이템의 부모 또는 자식에 해당하는 아이템들을 조회할 수 있습니다.

```
$item = CategoryItem::find($id);
// 자식에 해당하는 아이템
$children = $item->getChildren();
// 부모 아이템
$parent = $item->getParent();
```

또한 트리형태로 아이템들을 얻을 수 있습니다.

```
$category = Cagetory::find($id);
$tree = $category->getTree();     // 카테고리 전체 트리

$item = CategoryItem::find($id);
$tree = $item->getDescendantTree();  // 특정 아이템의 자손들로 이루어진 트리
```

데이터베이스(database)

XE가 자체적으로 부하분산을 처리할 수 있을까? 어떻게 하면 회원과 문서의 데이터베이스 사용을 분리시킬 수 있을까?

이 고민에서 시작이었습니다. 이미 Laravel은 훌륭한 데이터베이스 처리 방식을 제안하고 있습니다. 다만 우리는 앞서 얘기한 고민을 해결하기 위해 새로운 데이터베이스를 만들기 보다 우리가 하고자 하는 일을 해결할 수 있는 기능을 추가한 전혀 새롭지 않은 (Laravel 의 데이터베이스와 같은) 데이터베이스를 만들고 싶었습니다.

기본적인 사용 방법은 Laravel 과 같으며 단지 `XeDB` 파사드를 사용하면 됩니다.

> 기본적인 데이터베이스 사용, 쿼리빌더 의 내용을 숙지하시기 바랍니다.

설정

config/xe.php 의 `database` 에 가상 연결 설정을 등록합니다.

```
'database' => [
        'default' => [
            'slave' => ['default'],
        ],
        'document' => [
            'slave' => ['default'],
        ],
        'user' => [
            'slave' => ['default'],
        ],
    ],
```

XE에서는 회원과 문서의 데이터베이스를 분리해서 사용할 수 있도록 가상 연결을 분리해서 사용하고 있습니다. `app/Providers/` 의 UserServiceProvider.php, DocumentServiceProvider.php 에서 각각 `user` , `document` 커넥션을 사용하는 코드를 확인할 수 있습니다.

가상 연결

여러개의 논리적인 데이터베이스 연결을 사용하여 다중 커넥션 사용을 가능하도록 하고 서버에 트래픽이 증가할 경우 패키지별 데이터베이스 연결 설정을 변경하여 부하분산을 빠르게 처리할 수 있도록 합니다. 또한 하나의 논리적인 데이터베이스 연결에 여러개의 데이터베이스 연결 설정을 할 수 있도록 하고 Query 처리할 때 랜덤하게 커넥션을 사용하도록 구현하여 부하분산을 처리합니다.

또한 물리적으로 다른 여러개의 커넥션에 대해서 트랜젝션을 사용할 수 있습니다. 여러개의 커넥션에서 각각 발생하는 트랜젝션을 관리하여 하나의 커넥션에서 처리되는 것과 같이 동작합니다.

다이나믹 쿼리를 이용해서 ProxyManager 에 등록된 Proxy 들을 사용할 수 있습니다. 다이나믹 쿼리는 데이터베이스 CRUD 처리 시 발생하여 쿼리를 조작할 수 있도록 인터페이스를 제공합니다. ProxyInterface 의 인터페이스를 QueryBuilder 에서 각 메소드에서 필요한 인터페이스를 사용합니다.

Proxy

데이터베이스 Proxy는 쿼리 실행 실행에 앞서 쿼리를 수정할 수 있도록 인터페이스를 지원합니다. `DynamicQuery` 에서 `ProxyManager` 를 사용하도록 해서 inster(), update(), delete(), get(), first(), find(), paginate(), simplePaginate() 이 사용될 때 쿼리를 수정합니다. DynamicField 는 Proxy 기능을 이용해서 데이터베이스에 쿼리할 때 설정에 따라 추가된 확장 필드의 처리를 위한 쿼리를 실행하도록 구현했습니다.

Proxy 사용

```
$users = XeDB::dynamic('user')->get();
```

Proxy를 사용하기 위해서 `table()` 이 아닌 `dynamic()` 메소드를 사용합니다. `dynamic()` 을 사용해 반환된 `DynamicQuery` 인스턴스는 Proxy를 사용하기 위한 상태가 됩니다.

실제 데이터베이스 연결 설정은 기존의 Laravel 설정 방법과 동일합니다. config/database.php 에 데이터베이스 연결을 수정하거나 추가하면 됩니다.

트랜잭션

```
XeDB::beginTransaction();
... query ...
XeDB::commit(); // or XeDB::rollback();
```

XE3 데이터베이스는 여러개의 연결에 대해서 트랜잭션을 처리합니다. 전체 커넥션 정보를 참고하고 있는 TransactionHandler 는 트랜잭션이 시작될 때 연결되어 있는 모든 가상연결에 트랙잭션을 시작하고 또한 새로 맺는 연결도 트랜잭션이 시작되도록 처리합니다.

가상연결로 처리되는 하나 이상의 데이터베이스 연결에 대해서 트랜젝션 을 처리합니다. DB1, DB2 에 insert 처리 할 때 물리적으로 다른 두개의 데이터베이스에 트랜젝션이 처리될 수 있도록 지원합니다.

메일(email)

설정

메일에 대한 설정 파일은 `config/mail.php` 입니다. SMTP 호스트, 포트, 인증 및 라이브러리를 통해 송신되는 메세지들의 글로벌 `form` 주소를 설정할 수 있는 옵션들을 제공합니다. 원하는 경우 어떤 SMTP 서버라도 사용할 수 있습니다. 메일을 보낼 때 PHP의 `mail` 함수를 사용하려 한다면 설정 파일에서 `driver` 를 `mail` 로 변경하면 됩니다. 또한, `sendmail` 드라이버도 사용할 수 있습니다.

API 드라이버

또한, Mailgun과 Mandrill의 HTTP API 드라이버를 사용할 수 있습니다. HTTP API를 사용하려면 라라벨 문서를 참고하십시오.

로그 드라이버

`config/mail.php` 설정 파일에서 `driver` 옵션을 `log` 로 설정한다면 실제로 이메일을 수신자에게 보내지 않고 로그 파일에 기록하게 됩니다. 이 설정은 주로 로컬에서 빠르게 디버깅을 해야하거나 내용을 확인하고자 할때 유용합니다.

기본 사용법

`Mail::send` 메소드를 통해서 이메일을 보낼 수 있습니다:

```
Mail::send('emails.welcome', ['key' => 'value'], function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

`send` 메소드의 첫 번째 인자로 이메일의 본문에 사용되는 뷰(템플릿 파일)의 이름을 전달합니다. 두 번째는 뷰로 전달되는 인자인데 대부분 배열의 형태로 구성되며 `$key` 를 통해서 뷰에서 사용됩니다. 세 번째로 전달되는 클로저는 이메일 메세지에 대한 다양한 옵션을 지정하는 데 사용됩니다.

> **주의:** `$message` 라는 이름의 변수가 항상 이메일 뷰에 전달되고, 인라인 첨부를 사용가능하게 합니다. 따라서 `message` 라는 이름의 변수를 메일 뷰에서 사용하는 것은 피하는게 좋습니다.

만약 이메일 본문에서 사용할 템플릿 디자인이 준비되지 않았다면, XE에서 제공하는 템플릿 디자인을 사용할 수도 있습니다. XE에서 제공하는 템플릿 디자인은 `resources/views/emails/common.blade.php` 파일로 제공됩니다.

이메일 본문을 담고 있는 blade 파일에서 `emails.common` 을 extends 하십시오.

```
<!-- plugins/my_plugin/views/emails/welcome.blade.php -->

@extends('emails.common')

@section('content')

이메일 내용...

@endsection
```

HTML 뷰에 추가로 플레인 텍스트 뷰를 지정할 수도 있습니다:

```
Mail::send(['html.view', 'text.view'], $data, $callback);
```

또는 `html` 또는 `text` 키를 사용하고 한 종류의 뷰를 지정할 수 있습니다:

```
Mail::send(['text' => 'view'], $data, $callback);
```

이메일 메세지에 대해 참조나 첨부파일과 같은 다른 옵션들을 지정할 수도 있습니다:

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');

    $message->attach($pathToFile);
});
```

이메일에 첨부파일을 추가하고자 할때는 파일의 MIME 타입 또는 첨부파일이 표시되는 이름을 지정할 수 있습니다:

```
$message->attach($pathToFile, ['as' => $display, 'mime' => $mime]);
```

이메일을 보내기 위한 뷰 대신에 간단한 문자열을 사용하고자 한다면 `raw` 메소드를 사용하면 됩니다:

```
Mail::raw('Text to e-mail', function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');
});
```

> **주의:** `Mail::send` 클로저에 전달되는 메세지 인스턴스는 SwiftMailer 메세지 클래스를 확장하므로 이메일을 작성하는 데 필요한 클래스의 메소드들를 사용할 수 있습니다.

인라인 첨부

이메일에 인라인 이미지를 포함시키는 것은 번거로운 일입니다. XE는 이메일에 이미지를 첨부하고 최적의 CID를 얻을 수 있는 편리한 방법을 제공합니다.

이메일 뷰에서 이미지를 첨부하는 방법

```
<body>
    Here is an image:

    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

이메일 뷰에서 Raw 데이터를 첨부하는 방법

```
<body>
    Here is an image from raw data:

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

`$message` 변수는 항상 `Mail` 클래스에 의해서 뷰에 전달된다는 것에 주의하십시오.

로컬 개발환경에서의 메일

이메일을 전송하는 플러그인을 개발하는 경우에 로컬 또는 개발 환경이라면 메세지 전송을 비활성화 하는것이 바람직할 것입니다. 이를 위해서 `config/mail.php` 설정 파일에 `pretend` 옵션을 `true` 로 설정하거나 `Mail::pretend` 메소드를 호출하면 됩니다. 메일러가 `pretend` 모드인 경우에는 이메일 메세지는 수신자에게 송신되는 대신에 어플리케이션의 로그 파일에 기록됩니다.

만약 실제로 이메일이 어떻게 보여지는지 확인하고자 한다면 MailTrap과 같은 서비스를 이용하는 것도 고려해보시기 바랍니다.

프론트앤드(frontend)

브라우저에서 어떤 페이지를 요청하면 XE는 보통 모듈 스킨, 테마, 그리고 스킨과 테마에서 사용한 위젯이나 UI오브젝트에서 생성한 html 조각들을 조합하여 하나의 html 문서를 만듭니다. 그리고 이 html 문서에는 다양한 스타일시트와 스크립트 파일이 로드되고 meta 태그 같은 태그들이 html에 추가됩니다. Frontend 서비스는 이렇게 하나의 html 문서를 출력할 때 필요한 다양한 태그를 추가해주고 관리하는 역할을 합니다.

XE에서는 요청을 처리할 때마다 매우 다양한 플러그인이 실행되며, 각각의 플러그인들은 자신이 필요한 스크립트나 스타일시트 파일을 로드하게 됩니다. 만약 Frontend와 같은 관리 주체가 없다면, 플러그인들에 의해 동일한 스크립트 파일이 중복으로 로드될 수 있습니다.

Frontend 서비스는 아래와 같은 역할을 합니다.

- html문서의 타이틀을 지정한다.
- body 태그에 특정 class를 추가한다.
- js파일을 html 특정영역(head 또는 body의 상,하단)에 로드한다
- css파일을 html에 로드한다.
- 이미 다른 컴포넌트에서 로드된 js, css파일을 언로드(unload) 한다.
- meta 태그를 html에 로드한다.
- custom 태그(자유로운 형식의 text)를 html에 추가한다.
- form validation을 위한 rule을 지정한다.(브라우저에서 script를 통해 실행되는 validation)
- script에서 사용할 언어팩을 로드한다.

위의 역할을 수행할 때, Frontend 서비스는 여러 곳에서 로드된 js파일이나 css파일의 중복을 처리해줍니다. 또, js파일이나 css파일은 로드되는 순서도 매우 중요합니다. Frontend 패키지는 파일이 로드되는 순서를 지정하는 기능도 제공합니다.

브라우저 타이틀 태그 지정하기

`title` 메소드를 사용하십시오.

```
XeFrontend::title('브라우저 타이틀입니다');
```

body 태그의 class 지정하기

`bodyClass` 메소드를 사용하십시오.

```
// body에 'profile' class지정
XeFrontend::bodyClass('profile');
```

js 파일 로드하기

기본 사용법

`js` 메소드를 사용하여 스크립트 파일을 로드할 수 있습니다. 반드시 마지막에는 `load` 메소드를 사용해야 합니다.

```
// xe.js파일을 body의 하단에 로드함.
XeFrontend::js('assets/core/common/js/xe.js')->load();
```

이때, `appendTo` , `prependTo` 메소드를 사용하면, html 상에 스크립트 파일이 로드되는 위치를 지정할 수 있습니다. 지정하지 않을 경우 `<body>` 태그 하단에 로드합니다.

```
// xe.js파일을 head의 하단에 로드함.
XeFrontend::js('assets/core/common/js/xe.js')->appendTo('head')->load();

// xe.js파일을 body의 상단에 로드함.
XeFrontend::js('assets/core/common/js/xe.js')->prependTo('body')->load();
```

배열을 사용하여 다수의 파일을 동시에 로드할 수도 있습니다.

```
XeFrontend::js([
  'assets/vendor/jquery/jquery.min.js',
  'assets/core/common/js/xe.js',
  'plugin/board/assets/js/my.js'
)->load();
```

우선순위 지정

만약 스크립트 파일을 로드할 때, 반드시 먼저 로드돼야 하는 다른 스크립트 파일이 있다면 `before` 메소드를 사용하여 지정할 수 있습니다. 반대의 경우, `after` 메소드를 사용하십시오.

```
// bootstrap.js이 로드된 이후에 xe.js파일이 로드되도록 우선순위 지정
XeFrontend::js('assets/core/common/js/xe.js')
->before('assets/vendor/bootstrap/js/bootstrap.js')
->appendTo('body')->load();
```

다수의 파일을 동시에 로드한다면, 배열에 기입된 순서대로 우선순위가 정해집니다.

```
// 3 파일의 우선순위가 자동으로 지정됨
XeFrontend::js([
  'assets/vendor/jquery/jquery.min.js',
  'assets/core/common/js/xe.js',
  'plugin/board/assets/js/my.js'
)->load();
```

> **주의!** 만약 위의 방법을 사용하여 명시적으로 우선순위를 지정하지 않았다면, 로드된 파일들의 우선순위를 보장할 수 없습니다. 먼저 로드된 파일이라 하더라도 html상에서는 나중에 로드된 파일보다 늦게 로드될 수 있습니다.

언로드

이미 로드된 스크립트 파일이라도 `unload` 메소드를 사용하여 언로드할 수 있습니다.

```
// 로드된 xe.js파일을 언로드함.
XeFrontend::js('assets/core/common/js/xe.js')->unload();
```

css 파일 로드하기

css 파일도 js 스크립트 파일과 사용법이 동일합니다. 단 `appendTo`, `prependTo` 메소드를 지원하지 않습니다.

```
// xe.css파일을 로드함. 반드시 bootstrap.css가 로드된 다음에 로드되도록 우선순위를 지정
XeFrontend::js('assets/core/common/css/xe.css')->appendTo('body')->before('assets/vendor/bootstrap.css')->load();
```

meta 태그 추가

`meta` 메소드를 사용하여 meta 태그를 지정할 수 있습니다. meta 태그의 attribute를 지정하기 위해 `name`, `charset`, `property`, `httpEquiv`, `content` 를 지원합니다.

```
// 등록하려는 meta 태그의 별칭 등록.
$alias = 'my.viewport';

XeFrontend::meta($alias)->name('viewport')
->content('width=device-width, initial-scale=1.0')->load();
```

> `alias` 는 다른 곳에서 내가 입력한 meta 태그를 언로드할 때 key로 사용합니다.

custom html 태그 추가하기

`html` 메소드를 사용하면 자유롭게 원하는 코드를 추가할 수 있습니다.

```
$alias = 'myscript';

// script 코드를 `<body>` 하단에 추가
XeFrontend::html($alias)->content('
<script>
    $(function () {
        $('[data-toggle="tooltip"]').tooltip()
    })
</script>
')->appendTo('body')->load();
```

`unload` , `before`

icon 파일 로드하기

`icon` 메소드를 사용할 수 있습니다.

```
XeFrontend::icon($iconUrl)->load();
```

이미지처리(Image, Media)

XE에서는 사용자가 등록하는 이미지와 같은 미디어 파일들을 제어하기위한 `Media` 패키지가 있습니다. `Media` 는 `Intervention/image` 에서 제공하는 기능을 이용하여 설정에서 정의된 내용에 의해 섬네일을 어떤 형태로 생성할지 결정하고 사이즈별로 생성해줍니다. 또한 이미지외에도 오디오, 비디오 파일들을 간편하게 표현해주기 위한 인터페이스를 제공합니다.

설정

미디어의 설정은 `config/xe.php` 파일의 `media` 항목에서 지정합니다. 섬네일 설정은 다음과 같이 작성되어있습니다.

```
'thumbnail' => [
  'disk' => 'local',
  'path' => 'public/thumbnails',
  'type' => 'fit',
  'dimensions' => [
    'S' => ['width' => 200, 'height' => 200,],
    'M' => ['width' => 400, 'height' => 400,],
    'L' => ['width' => 800, 'height' => 800,],
  ],
],
```

- `disk` : 생성된 섬네일 이미지가 저장되어질 파일저장소를 지정합니다. 파일저장소는 `config/filesystems.php` 에서 정의합니다.
- `path` : 섬네일 이미지가 위치할 경로를 지정합니다.
- `type` : 어떤 형태로 섬네일 이미지를 생성할지 지정합니다. 지정할 수 있는 타입은 다음과 같습니다.
  - `fit` : 지정된 사이즈에 꽉 차고, 넘치는 영역은 삭제
  - `letter` : 지정된 사이즈안에 이미지가 모두 포함되면서 기존 비율과 같은 비율을 가지는 형태
  - `widen` : 가로 사이즈만을 기준으로 생성
  - `heighten` : 세로 사이즈만을 기준으로 생성
  - `stretch` : 비율을 무시하고 지정된 사이즈에 꽉 차게 생성
  - `spill` : 지정된 사이즈에 꽉 차고, 넘치는 영역은 삭제하지 않는 형태
- `dimensions` : 생성될 섬네일 이미지의 가로, 세로 사이즈를 지정합니다.

`Media` 에서는 비디오 파일에서 이미지를 추출하기 위한 확장기능을 제공합니다.

```
'videoExtensionDefault' => 'dummy',
'videoExtensions' => [
  'ffmpeg' => [
    'ffmpeg.binaries' => '/usr/local/bin/ffmpeg',
    'ffprobe.binaries' => '/usr/local/bin/ffprobe',
    'timeout' => 3600,
    'ffmpeg.threads' => 4,
  ]
],
'videoSnapshotFromSec' => 10
```

기본으로 지정된 `dummy` 는 이미지추출 작업을 하지 않는 가상의 확장기능입니다. 비디오에서 이미지를 추출하도록 하고 싶다면 `ffmpeg` 로 변경하고 필요한 항목을 설정해야 합니다. `ffmpeg` 를 사용하기 위해선 서버에 FFmpeg 가 설치되어있어야 합니다. 그리고 컴포저를 통해 다음 패키지가 설치되어야 합니다.

- php-ffmpeg/php-ffmpeg ~0.5

> 비디오 확장기능은 현재 `ffmpeg` 만 지원합니다.

섬네일 생성

섬네일을 생성하기 위해서는 우선 `File` 객체를 미디어 객체로 변환해야 합니다. `is` 메서드를 통해 특정파일이 미디어파일인지 확인하고, 미디어파일인 경우 섬네일을 생성하도록 합니다.

```
use XeStorage;
use XeMedia;
use Xpressengine\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
  public function uploadFile(Request $request)
  {
    $file = XeStorage::upload($request->file('attached'), 'path/to/dir');
    if (XeMedia::is($file)) {
      $media = XeMedia::make($file);
      $thumbnails = XeMedia::createThumbnails($media);
    }
  }
}
```

특정페이지는 섬네일 이미지 생성시 별도의 형태로 하고자 하는 경우 직접 타입을 지정할 수 있습니다.

```
$thumbnails = XeMedia::createThumbnails($media, 'widen');
```

편집기능 특별하게 사용하기

XE 에서 제공하는 이미지 편집 기능중에 `crop` 기능이 있습니다. `crop` 은 가로, 세로 사이즈뿐만 아니라 좌표값이 필요한 기능이므로 기본적인 섬네일 생성방식에서 제외되어 있습니다. 이 기능을 사용하기 위해서는 아래와 같이 조금 특별하게 코드가 작성되어야 합니다.

```
use XeStorage;
use Xpressengine\Media\Models\Image;
use Xpressengine\Media\Commands\CropCommand;
use Xpressengine\Media\Coordinators\Position;
use Xpressengine\Media\Coordinators\Dimension;
use Xpressengine\Media\Thumbnailer;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
  public function editImage($id)
  {
    $image = Image::find($id);

    $cropCmd = new CropCommand();
    $cropCmd->setPosition(new Position(50, 100));    // 시작 좌표 설정
    $cropCmd->setDimension(new Dimension(300, 200)); // 시작 좌표로 부터 가로, 세로 사이즈 지정

    $thumbnailer = new Thumbnailer();
    $content = $thumbnailer->setOrigin($image->getContent())->addCommand($cropCmd)->generate();

    XeStorage::create($content, 'path/to/dir', 'crop_file_name');
  }
```

`Thumbnailer` 는 처리할 명령을 순서대로 입력받을 수 있습니다. 만약 `crop` 한 이미지를 `letter` 타입으로 크기를 변환하려고 한다면 다음과 같이 할 수 있습니다.

```
$letterCmd = new LetterCommand();
$letterCmd->setDimension(new Dimension(100, 100));

$content = $thumbnailer->setOrigin($image->getContent())->addCommand($cropCmd)->addCommand($letterCmd)->generate();
```

미디어 표현

미디어의 객체는 간편하게 화면에 표현할 수 있도록 인터페이스를 제공합니다. 이것은 사전에 정의된 html 태그를 반환합니다.

```
<div>
  {{ $image->render() }}
</div>
<div>
  {{ $audio->render() }}
</div>
<div>
  {{ $video->render() }}
</div>
```

기본으로 제공하는 html 태그를 사용하지 않길 원할수도 있습니다. 이땐 단순히 파일의 url 을 반환받을 수 있으므로 별도의 태그를 직접 작성하여 해결할 수 있습니다.

```
<object data="{{ $video->url() }}" width="300" height="200">
  <param name="autoplay" value="true">
  <param name="showcontrols" value="true">
</object>
```

세션(Session)

설정

HTTP 기반의 어플리케이션은 상태를 저장할수 없기 때문에, HTTP 요청들에 관계없이 사용자의 정보를 저장하기위해서 세션이 사용됩니다. XE는 세션을 사용함에 있어 간결하고 통일된 API를 제공합니다. 세션 시스템으로는 Memcached, Redis 그리고 데이터베이스를 별다른 설정 없이, 동일한 API로 사용할 수 있습니다.

세션의 설정은 `config/session.php` 파일에 있습니다. 이 파일에는 각각의 옵션에 대한 정리된 문서가 포함되어 있으므로 잘 확인하시기 바랍니다. 기본으로는 `file` 세션 드라이버가 사용됩니다.

> **참고:** 세션을 암호화하여 저장하고자 한다면 `encrypt` 설정 옵션을 `true`로 지정하십시오.

예약어

XE는 내부적으로 `flash` 라는 세션키를 사용하고 있습니다. 이 이름으로 세션을 추가하지 마시기 바랍니다.

세션 사용법

여러 가지 방법으로 세션을 엑세스할 수 있습니다. HTTP 요청-request의 `session` 메소드를 사용하는 방법, `Session` 파사드를 사용하는 방법, 그리고 `session` 헬퍼함수를 사용할 수 있습니다. 아무런 전달 인자 없이 `session` 헬퍼함수를 호출할 때에는 전체 세션 객체가 반환됩니다

아이템 세션에 저장하기

```
Session::put('key', 'value');

session(['key' => 'value']);
```

배열 세션값으로 저장하기

```
Session::push('user.teams', 'developers');
```

세션에서 특정 아이템 가져오기

```
$value = Session::get('key');

$value = session('key');
```

특정 아이템을 찾거나 기본값 반환받기

```
$value = Session::get('key', 'default');

$value = Session::get('key', function() { return 'default'; });
```

아이템 값 가져온 후 삭제하기

```
$value = Session::pull('key', 'default');
```

세션의 모든 데이터 가져오기

```
$data = Session::all();
```

세션에 아이템이 존재하는지 확인하기

```
if (Session::has('users'))
{
    //
}
```

세션에서 특정 아이템 삭제하기

```
Session::forget('key');
```

세션의 모든 아이템 삭제하기

```
Session::flush();
```

세션 ID 재생성하기

```
Session::regenerate();
```

임시 데이터

때로는 바로 다음번의 요청에서만 사용하기 위해서 세션에 아이템을 저장할 수 있습니다. 바로 `Session::flash` 메소드를 사용하는 것입니다:

```
Session::flash('key', 'value');
```

현재의 임시 데이터를 다른 요청에서 사용하기 위해서 다시 임시 저장하기

```
Session::reflash();
```

임시 데이터의 일부 값만 다시 임시 저장하기

```
Session::keep(['username', 'email']);
```

데이터베이스 세션

`database` 세션 드라이버를 사용하는 경우 세션 아이템들이 저장될 테이블을 생성해야 합니다. 다음의 `Schema` 예제를 통해서 테이블을 생성할 수 있습니다.

```
Schema::create('sessions', function($table)
{
    $table->string('id')->unique();
    $table->text('payload');
    $table->integer('last_activity');
});
```

`session:table` 아티즌 명령어를 사용하면 이 마이그레이션을 생성할 수 있습니다!

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

세션 드라이버

세션 "드라이버"는 각각의 요청에 따라 세션이 어디에 저장될지를 결정합니다. XE는 별다른 설정 없이 다양한 드라이버를 사용할 수 있습니다:

- `file` - 세션이 `storage/framework/sessions` 폴더에 저장됩니다.
- `cookie` - 암호화된 쿠키를 사용하여 안전하게 세션을 저장할 것입니다.
- `database` - 세션은 어플리케이션에 의해서 데이터베이스에 저장됩니다.
- `memcached` / `redis` - 세션은 캐시 기반의 드라이버들에 의해 빠르게 저장됩니다.
- `array` - 세션은 간단한 PHP 배열에 저장되지만 현재 요청에 한해 일시적으로 저장됩니다.

Redis 세션을 사용하기 위해서는 컴포저를 통해서 `predis/predis (~1.0)` 패키지를 설치해야만 합니다.

> **주의:** 배열 드라이버의 경우에는 실제로 세션은 유지되지 않기 때문에 unit tests를 수행하는 데에만 사용하시기 바랍니다.

헬퍼(helpers)

쿠키(cookie)

XE에서 모든 쿠키는 서버에서 미들웨어를 거치면서 인증 코드와 함께 암호화 됩니다. 이 말은 클라이언트가 변경되었을 때 쿠키가 유효하지 않다는 것을 의미합니다. 쿠키 값은 Request 인스턴스와 Response 인스턴스를 사용하여 조회하거나 저장할 수 있습니다.

쿠키 값 가져오기

```
$value = $request->cookie('name');
```

Response에 새로운 쿠키 추가하기

`cookie` 헬퍼함수는 새로운 `Symfony\Component\HttpFoundation\Cookie` 인스턴스를 생성하기 위한 간단한 팩토리로 작동합니다. 쿠키를 `Response` 인스턴스에 추가하려면 `withCookie` 메소드를 사용하면 됩니다:

```
$response = new Illuminate\Http\Response('Hello World');

$response->withCookie(cookie('name', 'value', $minutes));
```

영원히 남게되는 쿠키 생성하기

영원히는 실제로는 *5*년을 의미합니다.

```
$response->withCookie(cookie()->forever('name', 'value'));
```

쿠키 큐처리 하기

플러그인 클래스의 `boot` 메소드는 XE의 라이프 사이클에서 매우 이른 시점에 실행됩니다. 이렇게 Response 인스턴스가 작성되기 전에 실행되는 코드에서는 반환되는 Response에 추가할 쿠키를 “큐 처리”할 수 있습니다:

```
<?php namespace App\Http\Controllers;

use Cookie;
use Illuminate\Routing\Controller;

class UserController extends Controller
{
    /**
     * Update a resource
     *
     * @return Response
     */
     public function update()
     {
        Cookie::queue('name', 'value');

        return response('Hello World');
     }
}
```

UI오브젝트 및 폼빌더

XE에서는 화면에 자주 출력되는 UI 요소들이 있습니다. 텍스트 입력 필드나 셀렉트 박스(select)와 같은 기본적인 폼 요소들이 있고, 테마 선택기나 메뉴 선택기, 또는 권한 설정 UI와 같이 XE에서만 사용되는 특별한 요소들도 있습니다. UI오브젝트는 이렇게 자주 사용되는 UI 요소를 개발자들이 쉽게 출력할 수 있는 방법을 제공합니다.

기본 사용법

`uio()` 함수를 사용하여 UI오브젝트를 출력할 수 있습니다.

만약 한줄 텍스트 폼 필드를 출력하려면 아래와 같이 작성할 수 있습니다.

```
// in blade template file
{{ uio('formText',
      ['label'=>'전화번호',
       'name'=>'phone',
       'id'=>'inputPhone',
       'description'=> '휴대폰 사용하여 입력하세요',
       'value'=>'010-000-0000'
]) }}
```

위 코드는 실제로 아래와 같이 변환되어 출력됩니다.

```
<div class="form-group">
  <label for="inputPhone">전화번호</label>
  <input type="text" class="form-control"
         id="inputPhone"
         name="phone" value="010-000-0000">
   <p class="help-block">휴대폰 사용하여 입력하세요</p>
</div>
```

uio 함수

`uio` 함수의 원형입니다.

```
uio($id, $args = [], $callback = null)
```

첫번째 파라미터 `$id` 는 UI오브젝트의 아이디(또는 별칭)입니다.

두번째 파라미터 `$args` 는 UI오브젝트를 출력하기 위해 전달할 정보를 담은 배열입니다. 각 UI오브젝트마다 전달하는 정보의 항목은 상이합니다.

세번째 파라미터 `$callback` 를 사용하면 UI오브젝트가 출력하는 html을 커스터마이징할 수 있습니다.

기본 제공되는 UI오브젝트

XE는 아래 목록과 같이 UI오브젝트를 기본으로 제공하고 있습니다.

별칭	설명
`formText`	`<input type="text">` 를 출력
`formPassword`	`<input type="password">` 를 출력
`formTextarea`	`<textarea>` 를 출력
`formSelect`	`<select>` 를 출력, 사용자로부터 단일 선택을 받을 때 사용하십시오.
`formCheckbox`	`<input type="checkbox">` 를 출력 사용자로부터 다중 선택을 받을 때 사용하십시오.
`formFile`	`<input type="file">` 출력, 파일을 업로드 받을 때 사용하십시오. 이 UI오브젝트는 jQueryFileUpload를 사용합니다.
`formImage`	`formFile` 과 같이 파일 업로드를 출력합니다. 하지만 이 UI오브젝트를 사용하면 사용자가 선택한 이미지 파일을 미리보기할 수 있습니다.
`formMenu`	XE에 등록된 메뉴 목록을 `<select>` 형식으로 출력합니다.
`formLangText` (=`langText`)	`formText` 와 같이 한줄 텍스트 박스를 출력합니다. 하지만 이 UI오브젝트는 다국어를 입력받을 수 있습니다.
`formLangTextarea` (=`langTextArea`)	여러줄의 다국어를 출력합니다.
`permission`	권한설정 UI를 출력합니다.
`themeSelect`	테마 선택 UI를 출력합니다.
`captcha`	캡챠 입력 UI를 출력합니다.

이 UI오브젝트들 이외에 여러 플러그인이 등록한 UI오브젝트를 사용할 수도 있습니다.

폼 빌더(폼 UI오브젝트)

폼은 웹페이지에서 사용자에게 입력을 받기 위해 매우 빈번히 사용됩니다. 특히 테마나 위젯, 스킨과 같은 컴포넌트는 사이트 관리자에게 폼으로 구성되어 있는 설정 페이지를 제공하므로 개발자가 설정 페이지를 작성해주어야 합니다. 이런 개발자의 수고를 줄이기 위해 UI오브젝트 중에는 폼 작성을 편하게 할 수 있도록 도와주는 UI오브젝트들이 있습니다.

폼 필드 UI오브젝트

폼은 `text` , `textarea` , `checkbox` 와 같은 다양한 '폼 필드'들로 구성됩니다. UI오브젝트 중 별칭(alias)이 `form*` 형식인 UI오브젝트들이 바로 '폼 필드 UI오브젝트'입니다. 폼 필드 UI오브젝트를 사용하면 손쉽게 폼 필드를 출력할 수 있습니다.

폼 UI오브젝트

또, UI오브젝트 중에 별칭(alias)이 `form` 인 UI오브젝트가 있는데, 이 UI오브젝트는 조금 특별합니다. 이 UI오브젝트는 하나의 완성된 폼을 출력합니다. 물론, 폼에 대한 기본정보(action, method 등)와 폼을 구성하는 필드의 목록은 파라미터로 입력받습니다.

폼 UI오브젝트도 다른 UI오브젝트와 같이 `uio` 함수로 사용할 수 있습니다.

```
{{ uio('form', $formData); }}
```

두번째 파라미터인 `$formData` 는 아래와 같은 형식을 가집니다.

```
$formData = [
  'fields' => [
    'title' => [
        '_type' => 'text',
        '_section' => '섹션1',
        'label' => '제목',
        'placeholder' => '제목을 입력하세요.',
    ],
    'content' => [
        '_type' => 'textarea',
        '_section' => '섹션1',
        'label' => '내용',
        'placeholder' => '내용을 입력하세요.',
    ],
  ],
  'value' => [
    'title' => 'foo..',
    'content' => 'bar..'
  ],
  'action' => '/',
  'method' => 'POST',
];
```

위 코드의 출력은 아래와 같습니다.

```
<form method="POST" enctype="multipart/form-data" action="/">
    <div class="panel form-section-섹션1">
        <div class="panel-heading">
            <div class="pull-left"><h3>섹션1</h3></div>
        </div>
        <div class="panel-body">
            <div class="row">
                <div class="col-md-6 form-col-title">
                    <div class="form-group">
                        <label for="">제목</label>
                        <input type="text" class="form-control" id="" placeholder="제목을 입력하세요." name="title" value
="foo..">
                        <p class="help-block"></p>
                    </div>
                </div>
                <div class="col-md-6 form-col-content">
                    <div class="form-group">
                        <label for="">내용</label>
                        <textarea class="form-control" rows="3" placeholder="내용을 입력하세요." name="content">bar..</t
extarea>
                    </div>
                </div>
            </div>
        </div>
    </div>
</form>
```

두번째 파라미터로 전달한 배열은 다음 항목을 제공해야 합니다.

`fields`

폼이 포함하고 있는 필드들의 목록입니다. 이 배열의 아이템 하나는 폼 필드 하나의 정보를 나타냅니다. 아이템의 키는 폼필트의 `name` 애트리뷰트로 사용됩니다.(위의 예에서는 `title`, `content`에 해당)

또 아이템 하나의 정보에서 `_type`, `_section`은 각각 폼필드의 타입과 폼필트가 출력될 그룹(섹션)명을 지정합니다. 그 이외의 정보들은 모두 지정된 폼필드를 설정할 때 전달됩니다.

위의 예제에서 첫번째 필드인 `title`의 경우, 타입이 `text`로 지정돼 있습니다. `text` 타입에 해당하는 `formText` UI오브젝트를 사용하여 이 폼 필드를 출력합니다. `label` 및 `placeholder` 정보는 `forxText` UI오브젝트에게 그대로 전달됩니다.

`value`

폼이 포함하고 있는 필드들이 가지는 값(value)의 목록입니다

카운터(counter)

수를 세는 일은 중요하지도 않고 복잡하지도 않으며 자주 사용되는 기능입니다. 우리는 글 조회수를 세거나 투표, 신고등 여러가지 기능을 구현할 때 로그를 남기고 그 수를 세려고 합니다.

수를 세는 기능들은 어떤 대상을 기준(이하 targetId)으로 인터넷 주소(IP)나 회원 아이디로 구분하여 처리합니다.

카운터 패키지는 IP와 회원 아이디를 기준으로하는 두가지 방법에 대해서 수를 셀 수 있는 기능을 제공합니다.

카운터 인스턴스

카운터를 사용하기 위해서는 카운터 인스턴스를 획득해야 합니다. 수를 셀 때 두가지 방식을 제공합니다. 그 첫번째는 게시물 조회수를 처리할 때 처럼 targetId에 대해서 IP, 회원 아이디를 기준으로 처리하면 되는 방식입니다. 이 때 조회수를 처리하기 위해 카운터 인스턴스를 획득하는 코드는 아래와 같습니다.

```
// 문서 조회에 사용할 카운터 반환
$readCounter = XeCounter::make($request, 'read');
```

두번째 방식은 투표수를 셀 때 사용하는 방식입니다. 투표는 targetId에 대해서 몇개의 옵션을 설정하고 각 옵션별 수를 셉니다. 그리고 IP, 회원 아이디는 targetId에 한번만 처리 되어야 합니다. 이러한 카운터의 인스턴스를 획득하는 코드는 아래와 같습니다.

```
// 투표(찬성, 반대)에 사용할 카운터 반환
$voteCounter = XeCounter::make($request, 'vote', ['assent', dissent']);
```

세번째 인자 `['assent', dissent']` 를 추가해서 사용가능합니다.

`XeCounter::make()` 로 반환된 카운터 인스턴스는 Interception Proxy 인스턴스로 intercept 할 수 있습니다.

조회수 증가

```
$doc = Document::find('id');

$readCounter = XeCounter::make($request, 'read');
// Counter는 $user 가 Guest라면 IP를 기준으로 수를 셉니다.
$readCounter->setGuest();

$user = Auth::user();
if ($readCounter->has($doc->id, $user) === true) {
  $readCounter->add($doc->id, $user);
}

$doc->readCount = $readCounter->getPoint($doc->id);
$doc->save();
```

> 조회수를 적용하기 위해서는 이와 같이 조회수를 카운터가 전달한 조회수를 문서에 직접 입력해야 합니다.

투표 취소

```
$doc = Document::find('id');

// 투표는 회원에 대해서만 동작합니다.
$voteCounter = XeCounter::make($request, 'vote', ['assent', 'dissent']);

// 인스턴스에 설정된 두개의 옵션 중에서 'assent' 에 투표
$option = 'assent';

$user = Auth::user();

// 회원에 대해서 동작하기 때문에 로그인하지 않은 회원인 경우 GuestNotSupportException 발생
try {
    $voteCounter->remove($doc, $user, $option);
} catch (GuestNotSupportException $e) {
    throw new AccessDeniedHttpException;
}

$doc->assentCount = $readCounter->getPoint($doc->id, 'assent');
$doc->save();
```

투표 로그

투표에 참여한 사용자 로그를 확인합니다.

```
$doc = Document::find('id');

// 투표는 회원에 대해서만 동작합니다.
$voteCounter = XeCounter::make($request, 'vote', ['assent', 'dissent']);

// 찬선에 투표한 회원 목록을 가져옵니다.
$users = $voteCounter->getUsers($doc->id, 'assent');
```

휴지통(trash)

Trash는 서비스 또는 서드파티에서 휴지통(bin)을 등록받아 처리합니다. BinInterface 를 따르는 구현체를 구현하고 TrashManager를 통해 등록하면 휴지통 비우는 요청이 있을 때 관련된 휴지통을 찾아 처리합니다.

Trash 서비스는 휴지통을 관리하며 휴지통에 포함된 아이템을 직접 컨트롤하지 않습니다. 커맨드를 통해 사용가능한 휴지통을 확인하고 휴지통 비우기 기능을 제공할 뿐입니다.

기본 사용법

XeTrash 파사드로 TrashManager를 사용합니다.

> 휴지통의 기능은 휴지통에 포함된 요약 정보와 그것을 비우는 역할이므로 코드 보다는 명령어 기준으로 설명합니다. 아래 나열된 사용법은 터미널에서 입력하는 명령어 사용 방법입니다.

휴지통 등록

휴지통 구현체를 TrashManager를 통해 등록합니다. artisan 명령어로 휴지통을 제어하기 위해서 휴지통을 등록해야하 합니다.

```
XeTrash::register(NAME_SPACE\CLASS_NAME::class);
```

요약 정보 확인

```
$bins = XeTrash::gets();
foreach ($bins as $bin) {
    echo $bin::summary();
}
```

전체 휴지통 비우기

```
XeTrash::clean();
```

명령어 사용

터미널에서 아래와 같이 명령어를 실행합니다.

전체 휴지통 요약 정보

```
$ php artisan trash
```

게시판 휴지통 정보 보기

```
$ php artisan trash board
```

댓글, 게시판의 요약 정보

한개 이상의 휴지통 요약 정보를 확인하고자 할 경우에는 콤마(,)로 구분해서 입력합니다.

```
$ php artisan trash board,comment
```

전체 휴지통 비우기

```
$ php artisan trash:clean
```

게시판 휴지통 비우기

```
$ php artisan trash:clean board
```

댓글, 게시판 비우기

```
$ php artisan trash:clean board,comment
```

유효성검사(validataion)

소개

XE는 유입되는 데이터의 유효성을 검사하기 위한 다양한 방법을 제공합니다.

유효성검사 살펴보기

유효성 검사 기능에 대해 알아보기 위해서, form을 확인한 뒤 사용자에게 에러 메세지를 보여주는 예제를 살펴보도록 하겠습니다.

라우트 정의하기

우선 다음의 라우트들이 정의되어 있다고 가정해 보겠습니다:

```
// Display a form to create a blog post...
Route::get('post/create', 'PostController@create');

// Store a new blog post...
Route::post('post', 'PostController@store');
```

`GET` 라우트는 사용자가 새로운 블로그 포스트를 생성하기 위한 form을 나타낼 것이고, `POST` 라우트는 데이터베이스에 새로운 블로그 포스트를 저장할 것입니다.

컨트롤러 생성하기

다음으로, 이 라우트들을 다루는 간단한 컨트롤러를 살펴보겠습니다. 지금은 `store` 메소드를 비어있는 채로 둘 것입니다:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Show the form the create a new blog post.
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```

유효성검사 로직 작성하기

이제 새로운 블로그 포스트에 대해 유효성을 검사하는 로직을 `store` 메소드에 채워넣을 준비가 되었습니다. 베이스 컨트롤러 (`App\Http\Controllers\Controller`) 클래스를 살펴보면 클래스가 `ValidatesRequests` 트레이트-trait을 사용한다는 것을 알 수 있습니다. 이 트레이트(trait)는 모든 컨트롤러에서 편리하게 사용할 수 있는 `validate` 메소드를 제공합니다.

`validate` 메소드는 HTTP 요청의 유입과 유효성 검사 룰의 집합을 전달 받습니다. 유효성 검사 룰들을 통과하게되면 코드는 계속해서 정상적으로 실행될 것입니다. 하지만 유효성 검사를 통과하지 못할 경우, 예외(exception)가 던져지고 적절한 오류 응답이 사용자에게 자동으로 보내질 것입니다. 전통적인 HTTP 요청의 경우, 리다이렉트 응답이 생성될 것이며 AJAX 요청에는 JSON 응답이 보내질 것입니다.

`validate` 메소드에 대해 더 잘 이해하기 위해, 다시 `store` 메소드로 돌아가 보겠습니다:

```
/**
 * Store a new blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid, store in database...
}
```

위에서 볼 수 있듯이, 간단하게 유입되는 HTTP 요청과 유효성 검사 룰들을 `validate` 메소드로 전달하면 됩니다. 이 때에도 유효확인이 실패하면 적절한 응답이 생성될 것입니다. 유효성 검사를 통과하면 컨트롤러는 계속해서 정상적으로 수행합니다.

중첩된 속성에 대한 유의사항

HTTP 요청이 "중첩된" 파라미터를 가지고 있다면 ".(점)" 문법을 사용하여 유효성 검사 규칙을 지정할 수 있습니다:

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

유효성 검사 에러 표시하기

그럼 입력되는 요청(request)의 파라미터들이 유효성 검사를 통과하지 못하는 경우에는 어떻게 될까요? 이 경우 앞서 언급한대로, 자동으로 사용자를 이전의 위치로 리다이렉트합니다. 또한 모든 유효성 확인 에러는 자동으로 세션에 임시 저장될 것입니다.

`GET` 라우트에서도 에러 메세지를 뷰와 명시적으로 출력하지 않아도 됩니다. 왜냐하면, 기본적으로 세션에 저장된 유효성 확인 에러를 출력합니다.

이 예제에서, 유효성 검사를 통과하지 못할 경우 사용자는 컨트롤러의 `create` 메소드로 리다이렉트 될것이고, XE는 세션에 저장된 유효성 확인 에러를 확인한 후, 토스트 팝업(팝업레이어 형식)으로 오류 메시지를 출력합니다.

AJAX 요청과 유효성 검사

이 예제에서는, 어플리케이션에 전통적인 form을 이용하여 데이터를 보냈습니다. 하지만 많은 어플리케이션이 AJAX 요청을 사용합니다. AJAX 요청 중에서 `validate` 메소드를 사용한다면 라라벨은 리다이렉트 응답을 생성하지 않습니다. 대신 라라벨은 유효성 검사의 모든 실패 에러들을 포함하는 JSON 응답을 생성할 것입니다. 이 JSON 응답은 422 HTTP 상태 코드와 함께 보내질 것입니다.

다른 유효성 검사 방법

수동으로 Validator 생성하기

컨트롤러에서 제공하는 `validate` 메소드를 사용하고 싶지 않다면 `Validator` 파사드를 사용하여 validator 인스턴스를 수동으로 생성할 수 있습니다. 파사드에 `make` 메소드를 사용하면 새로운 validator 인스턴스가 생성됩니다:

```
<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Store the blog post...
    }
}
```

`make` 메소드로 전달되는 첫번째 인자는 유효성 검사를 받을 데이터입니다. 두번째 인자는 데이터에 적용되어야 하는 유효성 검사 규칙들입니다.

요청(request) 정보가 유효성 검사를 통과하지 못한 것을 확인였다면 `withErrors` 메소드로 세션에 에러 메세지를 임시저장(flash)할 수 있습니다. 이 메소드를 사용하면 리다이렉트 후에 `$errors` 변수가 자동으로 뷰에서 공유되어 손쉽게 사용자에게 보여질 수 있습니다. `withErrors` 메소드는 validator, `MessageBag`, 혹은 PHP `array` 를 전달 받습니다.

이름이 지정된 Error Bags

한 페이지 안에서 여러개의 form을 가지고 있다면 에러들의 `MessageBag` 에 이름을 붙여 지정한 form에 맞는 에러 메세지를 조회할 수 있도록 할 수 있습니다. 단순히 `withErrors` 에 이름을 두번째 인자로 전달하면 됩니다:

```
return redirect('register')
            ->withErrors($validator, 'login');
```

그러면 `$errors` 변수에서 지정된 `MessageBag` 인스턴스에 접근할 수 있습니다.

```
{{ $errors->login->first('email') }}
```

에러 메시지 사용하기

`Validator` 인스턴스의 `messages` 메소드를 호출하면, 에러 메시지를 편하게 사용할 수 있는 다양한 메소드를 가진 `MessageBag` 인스턴스를 받을 수 있습니다.

하나의 필드에 대한 첫번째 에러 메시지 조회하기

특정 필드에 대한 첫번째 에러 메세지를 조회하려면 `first` 메소드를 사용하면 됩니다:

```
$messages = $validator->errors();

echo $messages->first('email');
```

하나의 필드에 대한 모든 에러 메세지 조회하기

간단하게 하나의 필드에 대한 모든 에러 메세지를 조회하고 싶다면 `get` 메소드를 사용하면 됩니다:

```
foreach ($messages->get('email') as $message) {
    //
}
```

모든 필드에 대한 모든 에러 메세지 조회하기

모든 필드에 대한 모든 에러 메세지를 조회하기 위해서는 `all` 메소들 사용하면 됩니다:

```
foreach ($messages->all() as $message) {
    //
}
```

하나의 필드에 대하여 에러 메시지가 존재하는지 검사하기

```
if ($messages->has('email')) {
    //
}
```

특정 형식으로 에러 메시지 획득하기

```
echo $messages->first('email', '<p>:message</p>');
```

특정 형식으로 모든 에러 메시지 획득하기

```
foreach ($messages->all('<li>:message</li>') as $message) {
    //
}
```

사용자 지정(커스텀) 에러 메세지

필요하다면 기본적인 에러 메세지 대신에 커스텀 에러 메세지를 유효성 검사에 사용할 수 있습니다. 커스텀 메세지를 지정하는 데에는 여러가지 방법이 있습니다. 먼저 `Validator::make` 메소드에 커스텀 메세지를 세번째 인자로 전달할 수 있습니다:

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

다음의 예에서 `:attribute` 플레이스 홀더는 유효성 검사를 받는 필드의 실제 이름으로 대체됩니다. 유효성 검사 메세지에서 다른 플레이스 홀더들 또한 활용할 수 있습니다. 예를 들어:

```
$messages = [
    'same'    => 'The :attribute and :other must match.',
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in'      => 'The :attribute must be one of the following types: :values',
];
```

주어진 속성에 대해 커스텀 메세지 지정하기

종종 하나의 특정 필드에 대해서만 커스텀 오류 메세지를 지정해야 하는 경우가 있습니다. 이것은 ".(점)" 표기법을 통해서 할 수 있습니다. 속성의 이름을 먼저 지정하고, 규칙을 명시하면됩니다:

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

언어 파일에 커스텀 메세지 지정하기

많은 경우에서, `Validator`에 직접 메세지를 전달하는 대신, 언어 파일에 속성에 따른 커스텀 메세지를 지정하기 원할 수 있습니다. 이렇게 하기 위해서는 `resources/lang/xx/validation.php` 언어 파일의 `custom` 배열에 메제지를 추가하면 됩니다.

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

사용가능한 유효성 검사 규칙들

다음은 모든 유효성 검사 규칙과 그들의 함수 목록입니다.

- Accepted
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Array
- Before (Date)
- Between
- Boolean
- Confirmed
- Date
- Date Format
- Different
- Digits
- Digits Between
- E-Mail
- Exists (Database)
- Image (File)
- In
- Integer

- IP Address
- JSON
- Max
- MIME Types (File)
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required Unless
- Required With
- Required With All
- Required Without
- Required Without All
- Same
- Size
- String
- Timezone
- Unique (Database)
- URL

accepted

필드의 값이 *yes*, *on*, *1*, 또는 *true*이어야 합니다. 이 것은 "이용약관" 동의와 같은 필드의 검사에 유용합니다.

active_url

필드의 값이 PHP 함수 `checkdnsrr`에 기반하여 올바른 URL이어야 합니다.

after:*date*

필드의 값이 주어진 날짜 이후여야 합니다. 이때 날짜는 `strtotime` PHP 함수를 통해 생성된 값입니다.

```
'start_date' => 'required|date|after:tomorrow'
```

`strtotime`에 의해 계산될 날짜 문자열을 전달하는 대신 날짜와 비교할 다른 필드를 명시할 수 있습니다:

```
'finish_date' => 'required|date|after:start_date'
```

alpha

필드의 값이 완벽하게 (숫자나 기호가 아닌) 알파벳[자음과 모음] 문자로 이루어져야 합니다.

(역자주: 영문 알파벳만을 의미하지 않고, 숫자나 기호가 아닌경우에 해당하여, 한글도 허용합니다.)

alpha_dash

필드의 값이 (숫자나 기호가 아닌) 알파벳[자음과 모음] 문자 및 숫자와 dash(-), underscore(_)로 이루어져야 합니다.

alpha_num

필드의 값이 완벽하게 (숫자나 기호가 아닌) 알파벳[자음과 모음] 문자 및 숫자로 이루어져야 합니다.

array

필드의 값이 반드시 PHP 배열 형태이어야 합니다.

before:*date*

필드의 값이 반드시 주어진 날짜보다 앞서야 합니다. 날짜는 `strtotime` PHP 함수를 통해 비교됩니다.

between:*min*,*max*

필드의 값이, 주어진 *min* 과 *max*의 사이의 값이어야 합니다. 문자열, 숫자, 그리고 파일이 `size` 룰에 의해 같은 방식으로 평가될 수 있습니다.

boolean

필드의 값이 반드시 boolean으로 캐스팅될 수 있어야 합니다. 허용되는 값은 `true` , `false` , `1` , `0` , `"1"` , `"0"` 입니다.

confirmed

필드의 값이 `foo_confirmation` 의 매칭되는 필드를 가져야 합니다. 예를 들어 만약 필드가 `password` 라면, `password_confirmation` 라는 필드가 입력값 중에 있어야 합니다.

date

필드의 값이 `strtotime` PHP 함수에서 인식할 수 있는 올바른 날짜여야 합니다.

date*format:_format*

필드의 값이 반드시 주어진 *format*과 일치해야 합니다. 주어진 포맷은 `date_parse_from_format` PHP 함수에 의해서 연산될 것입니다. 필드의 유효성을 검사할 때에는 `date` 와 `date_format` 중 **하나만** 사용해야 합니다.

different:*field*

필드의 값이 주어진 *field*의 값과 달라야 합니다.

digits:*value*

필드의 값이 반드시 숫자여야 하고, 길이가 *value*이어야 합니다.

digits*between:_min*,*max*

필드의 값이 주어진 *min*과 *max* 사이의 길이를 가져야 합니다.

email

필드의 값이 이메일 주소 형식이어야 합니다.

exists:*table*,*column*

필드의 값이 주어진 데이터베이스 테이블에 존재하는 값이어야 합니다.

exists 룰의 기본 사용법

```
'state' => 'exists:states'
```

특정 컬럼명 지정하기

```
'state' => 'exists:states,abbreviation'
```

쿼리문의 "where" 구문에 추가될 더 많은 조건을 지정할 수도 있습니다:

```
'email' => 'exists:staff,email,account_id,1'
```

"where" 구분에 `NULL` 혹은 `NOT_NULL` 을 전달할 수도 있습니다:

```
'email' => 'exists:staff,email,deleted_at,NULL'

'email' => 'exists:staff,email,deleted_at,NOT_NULL'
```

image

이미지 파일(jpeg, png, bmp, gif, svg)이어야 합니다.

in:*foo*,*bar*,...

필드의 값이 주어진 목록에 포함돼 있어야 합니다.

integer

필드의 값이 정수여야 합니다.

ip

필드의 값이 IP 주소여야 합니다.

json

필드의 값이 유효한 JSON 문자열이어야 합니다.

max:*value*

필드의 값이 반드시 *value*보다 작거나 같아야 합니다. 문자열, 숫자, 그리고 파일이 `size` 룰에 의해 같은 방식으로 평가될 수 있습니다.

mimes:*foo*,*bar*,...

파일의 MIME 타입이 주어진 확장자 리스트 중에 하나와 일치해야 합니다.

MIME 룰의 기본 사용법

```
'photo' => 'mimes:jpeg,bmp,png'
```

여러분은 확장자를 지정하기만 하면 되지만, 이 경우 파일의 컨텐트를 읽고 MIME 타입을 추정함으로써 이 파일의 MIME의 유효성을 검사합니다.

MIME 타입과 그에 상응하는 확장의 전체 목록은 다음의 위치에서 확인하실 수 있습니다:
http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:*value*

필드의 값이 반드시 *value* 보다 크거나 같아야 합니다. 문자열, 숫자, 그리고 파일이 `size` 룰에 의해 같은 방식으로 평가될 수 있습니다.

not*in:_foo,bar,...*

필드의 값이 주어진 목록에 존재하지 않아야 합니다.

numeric

필드의 값이 숫자여야 합니다.

regex:*pattern*

필드의 값이 주어진 정규식 표현과 일치해야 합니다.

참고: `regex` 패턴을 사용할 때, 특히 정규 표현식에 파이프 문자열이 있다면, 파이프 구분자를 사용하는 대신 배열 형식을 사용하여 룰을 지정할 필요가 있습니다.

required

입력 값 중에 해당 필드가 존재해야 하며 비어 있어서는 안됩니다. 필드는 다음의 조건 중 하나를 충족하면 "빈(empty)" 것으로 간주됩니다:

- 값이 `null` 인 경우.
- 값이 비어있는 문자열인 경우.
- 값이 비어있는 배열이거나, 비어있는 `Countable` 객체인경우
- 값이 경로없이 업로드된 파일인 경우

required*if:_anotherfield,value,...*

만약 *filed*의 값이 *value*중의 하나와 일치한다면, 해당 필드가 반드시 존재해야 합니다.

required*unless:_anotherfield,value,...*

*anotherfield*가 어떤 *value*와 동일하지 않은 이상 필드가 존재해야 합니다.

required*with:_foo,bar,...*

다른 지정된 필드중 하나라도 존재한다면, 해당 필드가 반드시 존재해야 합니다.

required*with_all:_foo,bar,...*

다른 지정된 필드가 모두 존재한다면, 해당 필드가 반드시 존재해야 합니다.

required*without:_foo,bar,...*

다른 지정된 필드중 하나라도 존재하지 않으면, 해당 필드가 반드시 존재해야 합니다.

required*without_all:_foo,bar,...*

다른 지정된 필드가 모두 존재하지 않으면, 해당 필드가 존재해야 합니다.

same:*field*

필드의 값이 주어진 *field*의 값과 일치해야 합니다.

size:*value*

필드의 값이 주어진 *value*와 일치하는 크기를 가져야 합니다. 문자열 데이터에서는 문자의 개수가 *value*와 일치해야 합니다. 숫자형식의 데이터에서는 주어진 정수값이 *value*와 일치해야 합니다. 파일에서는 킬로바이트 형식의 파일 사이즈가 *size*와 일치해야 합니다.

string

필드의 값이 반드시 문자열이어야 합니다.

timezone

필드의 값이 `timezone_identifiers_list` PHP 함수에서 인식 가능한 유효한 timezone 식별자여야 합니다.

unique:*table*,*column*,*except*,*idColumn*

필드의 값이 주어진 데이터베이스 테이블에서 고유한 값이어야 합니다. 만약 `column` 이 지정돼 있지 않다면 필드의 이름이 사용됩니다.

특정 컬럼명 지정하기:

```
'email' => 'unique:users,email_address'
```

특정 데이터베이스 커넥션

때때로, 여러분은 Validator에 의해서 생성되는 데이터베이스 쿼리에 사용자가 지정한 커넥션을 필요로 할지도 모릅니다. 위에서의 검증 규칙 `unique:users` 에서는 데이터베이스를 쿼리하기 위해 기본 데이터 베이스 커넥션이 사용됩니다. 이를 재지정하려면 테이블 이름 후에 "." 표기법으로 커넥션을 지정하십시오:

```
'email' => 'unique:connection.users,email_address'
```

주어진 ID에 대해서 유니크 규칙을 무시하도록 강제하기:

때때로 유니크 검사를 할 때 특정 ID를 무시하고자 할 수 있습니다. 예를 들어 사용자 이름, 이메일 주소 그리고 위치를 포함하는 "프로필 업데이트" 화면이 있습니다. 물론 이메일 주소가 고유하다는 것을 확인하고 싶을 것입니다. 하지만 사용자가 이름 필드만 바꾸고 이베일 필드를 바꾸지 않는다면 사용자가 이미 이메일 주소의 주인이기 때문에 유효 검사 오류가 던져지지 않아야 합니다. 유효 검사 오류는 사용자가 다른 사용자가 이미 사용하고 있는 이메일 주소를 제공할 때에만 던져져야 합니다. 유니크 규칙에 사용자 ID를 무시하라고 강제하기 위해서는 세번째 파라미터로 ID를 전달해야 합니다:

```
'email' => 'unique:users,email_address,'.$user->id
```

테이블이 `id` 가 아닌 primary 키 컬럼 이름을 사용한다면 그 이름을 네번째 파라미터로 지정하면 됩니다:

```
'email' => 'unique:users,email_address,'.$user->id.',user_id'
```

추가적인 Where 구문 추가하기:

더 추가되는 조건은 쿼리에 추가될 "where" 구문에 대해서 지정될 것입니다.

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

위의 룰에서는, `account_id` 가 `1` 인 데이터 만이 유니크 검사에 포함됩니다.

url

필드는 반드시 PHP `filter_var` 함수에 따라 유효한 URL이어야 합니다.

위젯박스

위젯박스는 말 그대로 위젯을 담는 박스라고 생각할 수 있습니다. 위젯박스는 사이트의 화면에 출력됩니다. 사이트 관리자는 위젯박스의 편집 페이지에서 다수의 위젯을 생성한 다음, 생성된 위젯들을 위젯박스에 배치할 수 있습니다. 사이트 관리자나 개발자는 위젯박스 삽입 코드를 사용하여 화면 어디에나 위젯박스를 출력할 수 있습니다.

예를들어, 위젯박스는 사이트의 사이드바 영역에 사용될 수 있습니다. 일반적인 테마의 레이아웃을 고려해보면, 헤더와 푸터, 그리고 메인 컨텐츠 영역으로 구성되며, 사이드바가 추가되기도 합니다. 사이드바 영역은 보통 사이트 관리자가 원하는 위젯들을 자유롭게 배치할 수 있기를 원합니다. 테마 개발자가 사이드바 영역에 위젯박스의 삽입 코드를 추가해놓으면, 사이트 관리자는 사이드바 영역을 자신이 원하는대로 구성할 수 있습니다. 비단 사이드바 영역이 아니더라도 개발자가 원하는 어느 영역이든 위젯박스를 출력할 수 있습니다.

위젯박스 출력하기

위젯박스는 아래와 같은 코드로 간단하게 출력할 수 있습니다.

```
{{ uio('widgetbox', ['id'=>'sidebar']) }}
```

위젯박스는 고유의 아이디를 가지며, 위 코드는 아이디가 `sidebar` 인 위젯박스를 화면에 출력합니다. 만약 아이디가 `sidebar` 인 위젯박스가 아직 존재하지 않는다면, 사이트 관리자가 클릭하면 즉시 위젯박스를 생성할 수 있는 링크를 아래와 같이 출력합니다.

```
위젯박스[sidebar]를 찾을 수 없습니다. [바로 생성하기]
```

이미 `sidebar` 위젯박스가 존재한다면, 바로 위젯박스가 출력되며, 위젯박스 하단에 위젯박스의 편집페이지 링크가 출력됩니다.

위젯박스 생성하기

위젯박스를 출력하기 전에 미리 위젯박스를 생성할 수 있습니다. `XeWidgetBox` 파사드를 사용하십시오.

```
XeWidgetbox::create(['id'=>'sidebar', 'title'=>'사이드바', 'content'=>'');
```

위젯박스 삭제하기

생성된 위젯박스는 `XeWidgetBox` 파사드의 `delete` 메소드를 사용하여 삭제할 수 있습니다.

```
XeWidgetbox::delete('sidebar');
```

테마

테마 컴포넌트는 테마 생성 커맨드를 제공합니다. 테마 생성 커맨드를 사용하면, 전문적인 PHP 지식이 없어도 테마를 제작할 수 있도록 테마를 구성하는 여러가지 파일을 자동으로 생성해 주고, 테마를 XE에 등록해 해줍니다.

테마 생성 커맨드를 사용하지 않더라도 테마를 제작할 수 있으며, 좀 더 자유롭운 형식으로 테마를 제작할 수 있습니다. 하지만 이 문서는 테마 생성 커맨드를 사용한 보편적인 방법의 테마 제작 방법을 설명합니다.

빈 테마 생성

테마 생성 커맨드를 사용하려면 우선 테마가 소속될 플러그인이 마련되어 있어야 합니다. 아직 플러그인이 준비되지 않았다면, 플러그인 생성하기 문서를 참고하여 플러그인을 생성하길 바랍니다.

플러그인이 준비되었다면 아래 커맨드를 사용하여 빈 테마를 생성합니다.

```
$ php artisan make:theme <path> <title>
```

`path` 는 테마가 위치할 디렉토리의 경로입니다. 플러그인의 디렉토리 이름을 포함한 경로를 입력해줍니다.

`title` 에는 테마의 제목을 지정해 주십시오. 지정한 테마 제목은 사이트 관리자에서 테마의 이름으로 표시됩니다.

만약 `my_plugin` 플러그인에 테마를 넣고, 테마 이름을 `First Theme` 로 지정하고 싶다면, 아래와 같이 커맨드를 실행하시면 됩니다. 커맨드를 실행하면 생성되는 테마의 개략적인 정보를 터미널에서 볼 수 있습니다.

```
$ php artisan make:theme my_plugin/theme 'First Theme'

[New theme info]
  plugin:        my_plugin
  path:          my_plugin/theme
  class file:    my_plugin/theme/Theme.php
  class name:    SungbumHong\MyPlugin\Theme
  id:            theme/my_plugin@theme
  title:         First Theme
  description:   The Theme supported by My_plugin plugin.

 Do you want to add theme? [yes|no]:
 > yes

Generating autoload files
Theme is created successfully.
```

테마 등록

테마 생성 커맨드를 사용할 경우, 자동으로 테마를 등록해 줍니다. 플러그인의 `composer.json` 파일에 아래와 같이 컴포넌트 정보가 등록되어 있습니다.

```
// plugins/my_plugin/composer.json
...
"extra": {
    "xpressengine": {
        "title": "my plugin",
        "component": {
            "theme/my_plugin@theme": {
                "class": "SungbumHong\\MyPlugin\\Theme\\Theme",
                "name": "First Theme",
                "description": "The Theme supported by My_plugin plugin."
            }
        }
    }
},
...
```

테마 아이디도 자동으로 생성됩니다. 위의 예시에서는 테마 아이디가 `theme/my_plugin@theme` 로 생성되었습니다.

테마 디렉토리 구조

생성된 테마는 아래의 디렉토리 구조를 가집니다. `plugins/my_plugin/theme/` 디렉토리는 테마의 모든 파일이 담겨 있는 '테마 디렉토리'입니다.

```
plugins/my_plugin/theme/
├── Theme.php
├── assets/
│   └── css/
│       └── theme.css
├── info.php
└── views/
    ├── gnb.blade.php
    └── theme.blade.php
```

assets 디렉토리

테마에 필요한 `.js` 나 `.css` 파일 그리고 이미지 파일과 같은 asset 파일을 담는 디렉토리입니다.

views 디렉토리

테마를 출력할 때 사용하는 템플릿 파일을 저장하는 디렉토리입니다. 템플릿 파일은 blade 템플릿 언어로 작성되어야 합니다. blade 템플릿 언어의 사용법은 템플릿 문서에 자세히 기술되어 있습니다.

Theme.php 파일

`Theme.php` 는 테마클래스 파일입니다. 테마 생성 커맨드로 생성된 테마의 클래스는 `\Xpressengine\Theme\GenericTheme` 클래스를 상속받고 있습니다. 또, `GenericTheme` 클래스는 테마 컴포넌트의 추상클래스인 `\Xpressengine\Theme\AbstractTheme` 를 상속받고 있습니다.

```
\Xpressengine\Theme\AbstractTheme
└── \Xpressengine\Theme\GenericTheme
    └── Theme(Theme.php)
```

PHP 개발자가 아닌 테마 제작자가 테마 클래스를 직접 작성하는 것은 쉬운 일이 아닙니다. `GenericTheme` 클래스는 테마 제작자들에게 규격화 된 테마 구조를 제공함으로써 테마 제작을 손쉽게 할 수 있도록 도와줍니다. `Theme.php` 파일을 처음 생성한 다음부터는 거의 직접 수정할 필요가 없습니다. `Theme.php` 를 직접 수정하는 대신, `info.php` 를 수정하십시오.

일반적으로 `Theme.php` 파일을 직접 수정하지 않아도 원하는 테마를 제작하는 데에는 문제가 없습니다. 다만, 좀더 고수준의 테마를 제작하고 싶거나, 특별한 기능을 테마에 추가하고 싶다면 `Theme.php` 파일을 직접 수정하셔도 됩니다. `Theme.php` 파일을 수정하면 훨씬 자유롭고 편하게 테마를 제작할 수 있습니다. 물론 그 전에 테마 클래스의 각 메소드가 갖는 역할과 원리를 잘 이해하고

있어야 합니다.

info.php 파일

`info.php` 파일은 테마의 구동에 필요한 여러가지 정보를 입력하는 파일입니다. 간단한 php 문법으로 쉽게 작성할 수 있습니다.

```
<?php
return [
    'view' => 'theme',
    'setting' => [
        ...
    ],
    'support' => [
        'mobile' => true,
        'desktop' => true
    ],
    'editable' => [
        'views' => [
            'theme.blade.php',
            'gnb.blade.php',
        ],
        'assets' => [
            'css/theme.css'
        ]
    ]
];
```

`view` 필드는 테마를 출력할 때 사용할 템플릿 파일을 지정하는 필드입니다. 위 설정의 경우 `theme` 로 지정되어 있습니다. 이는 `views` 디렉토리에 들어있는 `theme.blade.php` 을 지정한 것입니다.

`setting` 필드에는 테마 설정 페이지에서 사용할 설정 항목에 대한 정보를 지정합니다.

`support` 필드에는 이 테마가 데스크탑 버전과 모바일 버전을 지원하는지를 지정합니다.

`editable` 필드에는 테마 편집 페이지에서 편집할 수 있는 파일의 목록을 지정합니다.

테마 출력

본격적으로 테마를 제작해보겠습니다.

테마가 웹페이지에 출력될 때 사용될 템플릿은 `info.php` 파일의 `view` 필드에 지정되어 있습니다. 위 설정의 경우 실제로는 `views/theme.blade.php` 파일에 해당됩니다.

메인컨텐츠 출력하기

웹페이지가 출력될 때, 테마는 그 웹페이지의 메인컨텐츠를 반드시 출력해줘야 합니다. XE는 웹페이지를 출력할 때, 메인컨텐츠의 내용을 $content 변수에 담아 테마에 전달합니다.

메인컨텐츠가 출력될 영역에 아래와 같이 변수를 출력하십시오.

```
<div class="content" id="content">
{!! $content !!}
</div>
```

레이아웃 구성하기

테마에서 출력할 내용이 많아지면 `theme.blade.php` 이 너무 복잡해집니다. 이럴 때 블레이드 문법의 `@extends` , `@section` , `@yield` `@include` 키워드를 사용하면 템플릿 파일을 분리할 수 있습니다.

`theme.blade.php` 에서 `@include` 를 사용하여 header와 footer 영역을 분리해보았습니다.

```
<!-- theme.blade.php -->

@include($theme::view('header'))

<div class="content" id="content">
{!! $content !!}
</div>

@include($theme::view('footer'))
```

header와 footer에 해당하는 파일은 `views` 디렉토리에 미리 추가되어 있어야 합니다.

```
..
└── views/
    ├── header.blade.php
    ├── footer.blade.php
    └── theme.blade.php
```

테마의 템플릿 파일에서 뷰이름을 지정할 때에는 `$theme::view()` 메소드를 사용하십시오. 이 메소드는 `views` 디렉토리를 기준으로 하는 상대경로로 템플릿 파일의 경로를 지정할 수 있도록 도와줍니다.

asset 파일 로드하기

이미지 파일 로드하기

이미지 파일을 로드하려면 `$theme::asset()` 메소드를 사용하십시오. `assets` 디렉토리를 기준으로 파일의 상대경로를 입력해주시면 됩니다.

```
<!-- theme.blade.php -->

<img src="{{ $theme::asset('img/logo.png') }}" alt="로고 이미지">
```

css, js 파일 로드하기

css, js 파일은 `XeFrontend` 파사드(Frontend 서비스)를 사용하여 로드하십시오.

```
<!-- theme.blade.php -->

{{ XeFrontend::css(
    $theme::asset('css/theme.css')
)->load() }}
```

역시 `$theme::asset()` 메소드를 사용하면 상대경로로 파일경로를 입력할 수 있습니다.

`XeFrontend` 파사드의 자세한 사용법은 Frontend 서비스 문서를 참고하십시오.

테마 설정 변수 사용하기

사이트 관리자가 테마 설정 페이지를 통해 지정한 설정들의 값은 테마를 출력할 때 `$config` 변수를 사용하여 참조할 수 있습니다.

```
<!-- theme.blade.php -->

@if($config->get('use_sidebar', false))
<div class="sidebar">
...
</div>
@endif
```

`$config` 변수는 `ConfigEntity` 의 인스턴스입니다. `get` 메소드를 사용하면 손쉽게 설정값을 가져올 수 있습니다. `get` 메소드의 두번째 파라메터는 기본값(default value)입니다.

```
// use_sidebar 설정 조회, use_sidebar 설정이 존재하지 않을 경우 기본값으로 false를 반환
$config->get('use_sidebar', false);
```

메뉴 출력하기

XE에 등록된 메뉴도 테마 설정 변수로 저장할 수 있습니다.

만약 설정변수에 'mainMenu'라는 이름으로 메뉴가 저장되어 있다면, `menu_list` 함수를 사용하여 메뉴 정보를 가져올 수 있습니다.

```
@foreach(menu_list($config->get('mainMenu')) as $menu)
  ...
@endforeach
```

`menu_list` 는 지정된 메뉴에 속해 있는 메뉴 아이템의 목록을 배열로 반환합니다. 위의 예제에서는 `$menu` 변수가 하나의 메뉴 아이템입니다.

메뉴 아이템은 아래와 같은 프로퍼티를 가지고 있습니다. 이 프로퍼티를 사용하여 메뉴를 원하는대로 출력하십시오.

- `$menu['url']` - 메뉴의 링크주소
- `$menu['link']` - 메뉴의 링크텍스트, 만약 메뉴 이미지가 사용됐을 경우 이미지를 출력하는 태그를 출력합니다.
- `$menu['selected']` - 현재 페이지에 해당하는 메뉴일 경우 `true` 를 가집니다.
- `$menu['children']` - 자식 메뉴아이템 리스트

업로드한 이미지 출력하기

테마 설정페이지에서 사이트 관리자는 이미지를 업로드할 수 있습니다.

사이트 관리자가 업로드한 이미지를 출력할 때에는 아래와 같이 코드를 작성하십시오.

```
<img src="{{$config->get('logoImage.path')}}">
```

이미지를 저장한 변수명(`logoImage`)에 `.path` 를 붙여서 출력하면 됩니다.

테마 설정

테마는 사이트 관리자는 테마를 커스터마이징할 수 있도록 테마 설정 페이지를 제공할 수 있습니다. 테마 설정 페이지를 작성하는 방법은 크게 두가지가 있습니다.

폼빌더로 작성하기

폼빌더로 작성하는 방법을 사용하면 손쉽게 테마 설정 페이지를 작성할 수 있습니다.

`info.php` 파일의 `setting` 항목에 폼에서 사용할 필드의 정보를 담은 배열을 추가합니다. `setting` 항목을를 작성하는 규칙은 폼빌더 문서의 `fields` 항목을 참고하시기 바랍니다.

템플릿 파일로 작성하기

폼빌더를 사용할 경우, 간편하게 테마 설정 페이지를 작성할 수는 있지만, 자유롭지 못한 단점이 있습니다. 좀 더 자유롭게 설정 페이지를 작성하고 싶다면 설정 페이지를 출력하는 블레이드 템플릿 파일을 작성하십시오. 그리고 `info.php` 파일의 `setting` 항목에 블레이드 템플릿 파일의 경로를 지정하면 됩니다.

이때 템플릿 파일은 반드시 `views` 디렉토리에 존재해야 합니다. 만약 `views/config.blade.php` 를 설정 페이지의 템플릿으 로 사용하고 싶다면, 아래와 같이 `setting` 항목을 지정합니다.

```
<?php
return [
  ...

  'setting' => 'config',

  ...
];
```

테마 편집

사이트 관리자는 테마의 템플릿을 직접 편집하여 테마를 커스터마이징 할 수 있습니다. 테마 제작자는 사이트관리자가 편집할 수 있는 파일 목록을 지정해주어야 합니다.

`info.php` 파일의 `editable` 필드에 편집할 수 있는 파일 목록을 지정해주십시오.

스킨

스킨 컴포넌트는 스킨 생성 커맨드를 제공합니다. 스킨 생성 커맨드를 사용하면, 전문적인 PHP 지식이 없어도 스킨을 제작할 수 있도록 스킨을 구성하는 여러가지 파일을 자동으로 생성해 주고, 스킨을 XE에 등록해 해줍니다.

스킨 생성 커맨드를 사용하지 않더라도 스킨을 제작할 수 있으며, 좀 더 자유롭운 형식으로 스킨을 제작할 수 있습니다. 하지만 이 문서는 스킨 생성 커맨드를 사용한 보편적인 방법의 스킨 제작 방법을 설명합니다.

스킨 타겟

스킨은 항상 그 스킨이 사용되는 대상(타겟)이 있습니다. 예를 들어 게시판 스킨은 게시판이 대상이 되며, 회원 프로필 스킨은 회원 프로필 컨트롤러가 대상이 됩니다. 또 특정 위젯이 대상이 될 수도 있습니다. 게시판, 회원 프로필 컨트롤러, 위젯과 같이 스킨이 적용되는 대상을 '스킨 타겟'이라고 합니다. 모든 스킨은 스킨 타겟을 반드시 지정해야 합니다.

XE의 구성요소중 HTML을 출력하는 모든 구성요소는 스킨 타겟이 될 수 있습니다. 스킨 타겟은 HTML을 출력할 때, 스킨 시스템이 적용될 수 있도록 구현되어야 하며, 스킨 타겟 아이디를 지정해 주어야 합니다.

빈 스킨 생성

스킨 생성 커맨드를 사용하려면 우선 스킨이 소속될 플러그인이 마련되어 있어야 합니다. 아직 플러그인이 준비되지 않았다면, 플러그인 생성하기 문서를 참고하여 플러그인을 생성하길 바랍니다. 또, 스킨 타겟의 아이디를 미리 알고 있어야 합니다.

소속될 플러그인과 스킨 타겟 아이디가 준비되었다면 아래 커맨드를 사용하여 빈 스킨을 생성합니다.

```
$ php artisan make:skin <path> <skin_target_id> <title>
```

`path` 는 스킨이 위치할 디렉토리의 경로입니다. 플러그인의 디렉토리 이름을 포함한 경로를 입력해줍니다.

`skin_target_id` 는 스킨 타겟 아이디를 지정해주십시오.

`title` 에는 스킨의 제목을 지정해 주십시오. 지정한 스킨 제목은 스킨 선택시 스킨의 이름으로 표시됩니다.

만약 `my_plugin` 플러그인에 스킨을 넣고, 스킨 이름을 `나만의 프로필 스킨` 으로 지정하고 싶다면, 아래와 같이 커맨드를 실행하시면 됩니다. 커맨드를 실행하면 생성되는 스킨의 개략적인 정보를 터미널에서 볼 수 있습니다. 회원 프로필 컨트롤러의 스킨 타겟 아이디는 `member/profile` 입니다.

```
$ php artisan make:skin my_plugin/skins/profile member/profile "나만의 프로필 스킨"

[New skin info]
  plugin:        my_plugin
  path:          my_plugin/skins/profile
  class file:    my_plugin/skins/profile/ProfileSkin.php
  class name:    SungbumHong\MyPlugin\ProfileSkin
  id:            member/profile/skin/my_plugin@profileskin
  title:         나만의 프로필 스킨
  description:   The Skin supported by My_plugin plugin.

 Do you want to add skin? [yes|no]:
 > yes

Generating autoload files
Skin is created successfully.
```

스킨 컴포넌트 아이디

스킨도 컴포넌트이므로 컴포넌트 아이디를 가지고 있어야 합니다. 스킨의 컴포넌트 아이디는 아래 규칙을 따라야 합니다.

```
<skin_target_id>/skin/<plugin_id>@<pure_id>
```

위의 예시에서 생성한 스킨의 아이디는 `member/profile/skin/my_plugin@profileskin` 입니다.

스킨 등록

스킨 생성 커맨드를 사용할 경우, 자동으로 스킨을 등록해 줍니다. 플러그인의 `composer.json` 파일에 아래와 같이 컴포넌트 정보가 등록되어 있습니다.

```
// plugins/my_plugin/composer.json
...
  "extra": {
        "xpressengine": {
            "title": "my plugin",
            "component": {
                "member/profile/skin/my_plugin@profileskin": {
                    "class": "SungbumHong\\MyPlugin\\Skin\\ProfileSkin",
                    "name": "나만의 프로필 스킨",
                    "description": "The Skin supported by My_plugin plugin."
                }
            }
        }
    },

    ...

    "autoload": {
        "classmap": [
            "skins/profile/ProfileSkin.php"
        ]
    }
```

`autoload` 항목에 스킨 클래스가 등록돼 있는 것도 볼 수 있습니다.

스킨 디렉토리 구조

생성된 스킨은 아래의 디렉토리 구조를 가집니다. `plugins/my_plugin/skins/profile` 디렉토리는 스킨의 모든 파일이 담겨 있는 '스킨 디렉토리'입니다.

```
plugins/my_plugin/skins/
└── profile/
    ├── assets/
    │   └── css/
    │       └── skin.css
    ├── views/
    │   └── index.blade.php
    ├── ProfileSkin.php
    └── info.php
```

assets 파일

스킨에 필요한 `.js` 나 `.css` 파일 그리고 이미지 파일과 같은 asset 파일을 담는 디렉토리입니다.

views 디렉토리

스킨을 출력할 때 사용하는 템플릿 파일들을 저장하는 디렉토리입니다. 템플릿 파일은 blade 템플릿 언어로 작성되어야 합니다. blade 템플릿 언어의 사용법은 템플릿 문서에 자세히 기술되어 있습니다.

Skin.php 파일

`Skin.php` 파일(위의 예에서는 `ProfileSkin.php` 파일)은 스킨클래스 파일입니다. 스킨 생성 커맨드로 생성된 스킨의 클래스는 `\Xpressengine\Skin\GenericSkin` 클래스를 상속받고 있습니다. 또, `GenericSkin` 클래스는 스킨 컴포넌트의 추상클래스인 `\Xpressengine\Skin\AbstractSkin` 를 상속받고 있습니다.

```
\Xpressengine\Skin\AbstractSkin
└── \Xpressengine\Skin\GenericSkin
    └── Skin(Skin.php)
```

PHP 개발자가 아닌 스킨 제작자가 스킨 클래스를 직접 작성하는 것은 쉬운 일이 아닙니다. `GenericSkin` 클래스는 스킨 제작자들에게 규격화 된 스킨 구조를 제공함으로써 스킨 제작을 손쉽게 할 수 있도록 도와줍니다. `Skin.php` 파일은 처음 생성한 다음부터는 거의 직접 수정할 필요가 없습니다. `Skin.php` 를 직접 수정하는 대신, `info.php` 를 수정하십시오.

일반적으로 `Skin.php` 파일을 직접 수정하지 않아도 원하는 스킨을 제작하는 데에는 문제가 없습니다. 다만, 좀더 고수준의 스킨을 제작하고 싶거나, 특별한 기능을 스킨에 추가하고 싶다면 `Skin.php` 파일을 직접 수정하셔도 됩니다. `Skin.php` 파일을 수정하면 훨씬 자유롭고 편하게 스킨을 제작할 수 있습니다. 물론 그 전에 스킨 클래스의 각 메소드가 갖는 역할과 원리를 잘 이해하고 있어야 합니다.

info.php 파일

`info.php` 파일은 스킨의 구동에 필요한 여러가지 정보를 입력하는 파일입니다. 간단한 php 문법으로 쉽게 작성할 수 있습니다.

```php
<?php
return [
    'setting' => [
        'sample_text' => [
            '_type' => 'text',
            '_section' => '기본설정',
            'label' => '샘플 문구',
            'placeholder' => '샘플용 설정 필드입니다.',
            'description' => '샘플용 설정 필드입니다.',
        ],
    ],
    'support' => [
        'mobile' => true,
        'desktop' => true
    ]
];
```

`setting` 필드에는 스킨 설정 페이지에서 사용할 설정 항목에 대한 정보를 지정합니다.

`support` 필드에는 이 스킨이 데스크탑 버전과 모바일 버전을 지원하는지를 지정합니다.

스킨의 출력

하나의 스킨은 하나 이상의 페이지의 출력을 담당할 수 있습니다. 출력하는 페이지의 갯수는 스킨 타겟에 따라 다릅니다. 게시판의 경우, 글보기, 글쓰기, 글목록 등의 페이지를 가지고 있는데, 스킨은 게시판이 필요로 하는 모든 페이지의 출력을 담당합니다.

스킨 설정

위젯

가장 먼저 해야할 작업은 다른 컴포넌트와 마찬가지로 추상클래스인 `\Xpressengine\Widget\AbstractWidget` 을 상속받는 클래스를 구현해야 합니다. 그 다음 구현한 클래스를 XE에 등록해 주면 됩니다.

클래스 생성하기

이 문서에서는 위젯 작성법을 쉽게 이해할 수 있도록 '로그인 정보 위젯'을 예로 들어 설명하겠습니다. 로그인 정보 위젯은 현재 로그인되어 있는 회원의 정보를 출력하는 간단한 위젯입니다.

```
<?php
// plugins/myplugin/src/Widgets/UserInfoWidget.php
namespace MyPlugin\Widgets;

class UserInfoWidget extends \Xpressengine\Widget\AbstractWidget
{
    public function render()
    {
    }

    public function renderSetting(array $args = [])
    {
    }

    public function resolveSetting(array $inputs = [])
    {
    }
}
```

위와 같이 작성한 클래스 파일을 컴포넌트를 담을 플러그인에 생성합니다. 파일의 위치는 플러그인 디렉토리 내의 어느 곳이든 상관없습니다. 다만 플러그인 디렉토리의 `src/Widgets/UserInfoWidget.php` 에 생성하는 것을 권장합니다.

위젯 클래스는 기본적으로 작성해야 할 메소드가 있습니다.

`render` - 위젯을 출력할 때 호출되는 메소드입니다.

`renderSetting` - 위젯변수를 입력받기 위한 폼을 출력합니다. 위젯변수는 위젯을 출력할 때 필요한 컨텐츠에 대하여 사이트 관리자로부터 입력받는 값을 말합니다. 사이트 관리자가 위젯 생성기를 통해 위젯(위젯코드)을 생성할 때, 위젯 생성기는 이 메소드를 실행하여 위젯 변수 입력폼을 출력합니다.

`resolveSetting` - 사이트 관리자가 위젯 생성기에서 입력한 위젯 변수를 조합하여 위젯코드를 생성합니다. 이 메소드는 입력받은 위젯 변수를 파라미터로 전달 받은 다음, 위젯 변수를 한번 재가공하여 반환합니다.

위젯 등록하기

작성한 위젯 클래스는 다른 컴포넌트와 마찬가지로 XE에 등록해야 합니다. 위젯은 `widget/<plugin_name>@<pure_id>` 형식의 컴포넌트 아이디를 지정해야 합니다. 여기에서는 `widget/myplugin@userinfo` 를 아이디로 사용하겠습니다.

등록 방법은 컴포넌트 등록 문서를 참고하시기 바랍니다.

성공적으로 등록되었는지 아래 코드로 테스트해 볼 수 있습니다.

```
<xewidget id="widget/myplugin@userinfo"></xewidget>
```

위젯 출력하기

기본적인 방법으로 출력하기

앞서 말한 바와 같이, 위젯이 출력될 때에는 `render` 메소드가 사용됩니다.

```
<?php
// plugins/myplugin/src/Widgets/UserInfoWidget.php
namespace MyPlugin\Widgets;

class LoginInfoWidget extends \Xpressengine\Widget\AbstractWidget
{
    public function render()
    {
      // 로그인 상태일 경우, 로그인 회원의 이름이 출력
      if(auth()->check()) {
        return auth()->user()->getDisplayName();
      } else {
        return '로그인 상태가 아닙니다'
      }
    }
}
```

로그인 회원의 이름을 출력하거나 '로그인 상태가 아닙니다' 메시지를 출력하도록 작성된 코드입니다. 이 예제에서는 간단히 문자열을 반환했습니다. 좀 더 복잡한 위젯일 경우 블레이드 템플릿을 사용할 수도 있습니다.

위젯 스킨을 사용하여 출력하기

위젯은 스킨시스템을 기본적으로 지원합니다. 동일한 컨텐츠를 출력하는 위젯이라고 해도, 사이트의 디자인이나 테마에 따라 다른 스킨을 선택하여 출력할 수 있습니다.

```
<?php
// plugins/myplugin/src/Widgets/UserInfoWidget.php

    ...

    public function render()
    {
      $data = [
        'isLogged' => auth()->check(),
        'user' => auth()->user()
      ];
      return $this->renderSkin($data);
    }
```

위 코드에서와 같이 `renderSkin`을 사용하면 됩니다. `renderSkin` 메소드는 위젯코드에 지정된 스킨정보를 자동으로 가져온 다음, 스킨을 적용하여 출력합니다.

위젯 변수 사용하기

사이트 관리자로부터 위젯 변수를 입력받기 위해 위젯 생성기는 위젯 변수 입력폼을 출력합니다. 위젯 변수를 사용하려면 위젯 변수 입력폼을 먼저 작성해야 합니다.

위젯 설정폼 구현

위젯 클래스의 `renderSetting` 메소드가 위젯 변수 입력폼을 출력하도록 하십시오.

```
<?php
// plugins/myplugin/src/Widgets/UserInfoWidget.php

    ...

    public function renderSetting(array $args = [])
    {
      // 회원이름 뒤에 나오는 호칭(예: xxx님)
      return uio('formText', ['name'=>'postfix', 'value'=>array_get($args, 'postfix'), 'label'=>'호칭', 'description'
=>'회원이름 뒤에 출력될 호칭을 적어주세요']);
    }
```

위 코드의 경우, 위젯 생성기에 하나의 텍스트박스가 출력됩니다. 텍스트박스를 출력하기 위해 UI오브젝트를 사용하고 있습니다.

입력된 설정 정보 처리

사이트 관리자가 위젯 변수 입력폼에 입력한 내용을 위젯코드로 변환하기 전에 한번 더 재가공할 수 있습니다. `resolveSetting` 메소드를 사용하십시오. 위젯시스템은 위젯코드를 생성하기 전에 이 메소드를 실행합니다. 만약 사용자가 입력한 위젯 변수의 유효성 검사나 재처리가 필요하다면 이 메소드에 구현하면 됩니다.

```
<?php
// plugins/myplugin/src/Widgets/UserInfoWidget.php

    ...

    public function resolveSetting(array $inputs = [])
    {
      if(!array_get($inputs, 'postfix'))
      {
        throw new ValidationException();
      }

      return $input;
    }
```

위젯을 출력할 때 위젯 변수 사용하기

사이트 관리자가 입력한 위젯 변수는 `render` 메소드에서 바로 사용할 수 있습니다. 위젯변수는 `$this->config` 배열에 저장되어 있습니다.

```
<?php
// plugins/myplugin/src/Widgets/UserInfoWidget.php

    ...

    public function render()
    {
        // 로그인 상태일 경우, 로그인 회원의 이름이 출력
        if(auth()->check()) {
          return auth()->user()->getDisplayName().array_get($this->config, 'postfix');
        } else {
          return '로그인 상태가 아닙니다'
        }
    }
}
```

모듈

XE는 사이트 관리 페이지를 통해 각각의 메뉴를 정의할 수 있는 기능을 제공합니다. 메뉴의 아이템으로 등록되는 컴포넌트는 모듈로 생성하여 등록할 수 있습니다.

모듈 생성

새로운 모듈을 생성하기 위해선 `Xpressengine\Menu\AbstractModule` 을 상속받아 클래스를 구현해야 합니다. 이 클래스는 XE에서 사용되어지기 위해 다음과 같이 모듈을 지칭하는 아이디를 가져야 합니다.

```
module/<plugin_id>@<module_id>
```

모듈 구조

모듈은 메뉴에 등록, 수정 또는 삭제처리를 위한 메서드를 구현해야 합니다.

```
use Xpressengine\Menu\AbstractModule;

class MyModule extends AbstractModule
{
  public function createMenuForm()
  {
    ...
  }

  public function storeMenu($instanceId, $menuTypeParams, $itemParams)
  {
    ...
  }

  public function editMenuForm($instanceId)
  {
    ...
  }

  public function updateMenu($instanceId, $menuTypeParams, $itemParams)
  {
    ...
  }

  public function summary($instanceId)
  {
    ...
  }

  public function deleteMenu($instanceId)
  {
    ...
  }

  public function getTypeItem($id)
  {
    ...
  }
}
```

`createMenuForm` 와 `editMenuForm` 메서드는 메뉴를 생성 또는 수정할 때, 모듈이 필요로 하는 값을 사이트 관리자로부터 입력받기 위한 폼을 반환하는 메서드입니다. 이 폼에 의해 입력된 값은 메뉴를 등록하거나 수정할 때 호출되는 `storeMenu` , `updateMenu` 메서드의 `$menuTypeParams` 파라미터로 전달됩니다.

메뉴가 삭제될 때에는 `deleteMenu` 메서드를 호출하게 되는데, 만약 삭제를 처리하기 전 사용자에게 알려야 할 메시지가 있다면 `summary` 메서드에 작성하십시오. 해당 메시지가 사용자에게 노출됩니다.

때때로 XE의 기능요소들이 메뉴를 통해 특정 모듈에 속한 객체 아이템에 접근하고자 하는 경우가 있습니다. `getTypeItem` 는 이럴 때 모듈의 객체 아이템을 반환하기 위해 정의된 메서드입니다.

라우트

메뉴는 대부분 각각의 메뉴아이템에 연결된 주소로의 이동을 위해 라우트를 가집니다. 그러나 외부링크와 같은 특별한 경우에는 라우트를 가지지 않기도 합니다. 이럴 때는 `isRouteAble` 메서드를 통해 `false` 값을 반환해야 합니다.

```
class MyModule extends AbstractModule
{
  ...
  public static function isRouteAble()
  {
    return false;
  }
  ...
}
```

설정페이지

모듈은 다른 컴포넌트와 마찬가지로 `getSettingsURI` 메서드를 통해 설정페이지 주소를 반환합니다. 하지만 모듈은 각각의 메뉴의 아이템으로서 생성되어지면서 인스턴스를 가지게 됩니다. 이때 각 인스턴스별로 설정페이지가 존재할 수 있습니다. 이럴땐 `getInstanceSettingURI` 메서드를 통해 각 인스턴스별 설정페이지 주소를 반환해야 합니다.

```
class MyModule extends AbstractModule
{
  ...
  public static function getInstanceSettingURI($instanceId)
  {
    return route('mypluing.mymodule.setting', $instanceId);
  }
  ...
}
```

UI오브젝트

UI오브젝트는 아래와 같이 매우 간단한 절차로 작동합니다.

1. PHP코드나 블레이드 파일에서 `uio($id, $arguments, $callback)` 함수 호출
2. `uio` 함수는 `$id` 에 해당하는 UI오브젝트의 인스턴스를 생성(생성시 `$arguments` 와 `$callback` 을 파라미터로 전달)
3. 생성된 인스턴스의 `render` 메소드를 호출
4. `render` 메소드가 반환하는 결과값을 출력

가장 먼저 해야할 작업은 다른 컴포넌트와 마찬가지로 추상클래스인 `\Xpressengine\UIObject\AbstractUIObject` 를 상속받는 클래스를 구현해야 합니다. 그 다음 구현한 클래스를 XE에 등록해 주면 됩니다.

클래스 생성하기

이 문서에서는 UI오브젝트의 작성법을 쉽게 이해할 수 있도록 '이미지 UI오브젝트'를 예로 들어 설명하겠습니다. 이미지 UI오브젝트는 이미지 파일 주소를 입력받아 `<img src="이미지 파일 주소">` 형태로 출력하는 간단한 역할을 합니다.

```
<?php
// plugins/myplugin/src/UIObjects/ImageUIObject.php
namespace MyPlugin\UIObjects;

class ImageUIObject extends AbstractUIObject
{
    public function render()
    {
      // implement code

      return parent::render();
    }
}
```

위와 같이 작성한 클래스 파일을 컴포넌트를 담을 플러그인에 생성합니다. 파일의 위치는 플러그인 디렉토리 내의 어느 곳이든 상관없습니다. 다만 플러그인 디렉토리의 `src/UIObjects/ImageUIObject.php` 에 생성하는 것을 권장합니다.

출력코드 작성하기

클래스를 생성하였다면, `render` 메소드를 구현합니다.

`render` 메소드에서는 입력된 파라미터를 사용하여 출력할 html 스트링을 생성한 다음, `$this->template` 변수에 저장합니다.

```
    public function render()
    {
      // 입력값 가져오기
      $src = $this->arguments['src'];
      $alt = $this->arguments['alt'];

      // 이미지 태그 생성
      $this->template = '<img src="'.$src.'" alt="'.$alt.'">';

      // 출력
      return parent::render();
    }
```

마지막으로는 반드시 `return parent::render()` 를 해주어야 합니다. `parent::render()` 메소드는 `$callback` 입력이 있는지 검사하고, `$callback` 를 자동으로 실행해주는 역할을 합니다.

UI오브젝트 등록하기

작성한 UI오브젝트 클래스는 다른 컴포넌트와 마찬가지로 XE에 등록해야 합니다. UI오브젝트는 `uiobject/<plugin_name>@<pure_id>` 형식의 컴포넌트 아이디를 지정해야 합니다. 여기에서는 `uiobject/myplugin@image` 를 아이디로 사용하겠습니다.

등록 방법은 컴포넌트 등록 문서를 참고하시기 바랍니다.

성공적으로 등록되었는지 아래 코드로 테스트해 볼 수 있습니다.

```
uio('uiobject/myplugin@image', ['src'=>'path/to/image.jpg', 'alt'=>'test image']);
```

alias 등록하기

위 예제에서 생성한 UI오브젝트의 아이디는 `uiobject/myplugin@image` 로 꽤 복잡합니다. XE에서는 UI오브젝트에 별칭(alias)를 지정할 수 있습니다. 별칭이 지정된 UI오브젝트는 실제 아이디 대신 별칭을 사용할 수 있습니다. UI오브젝트의 별칭은 사이트 관리자가 `config/xe.php` 의 `uiobject > aliases` 항목에 지정할 수 있으며, 코드상에서는 아래의 방법으로 지정할 수 있습니다.

```
XeUIObject::setAlias($alias, $id);
```

위 예제에서 제작한 UI오브젝트에 `image` 라는 별칭을 아래와 같이 지정할 수 있습니다.

```
XeUIObject::setAlias('image', 'uiobject/myplugin@image');
```

위와 같이 지정한 이후부터는 별칭을 사용할 수 있습니다.

```
uio('image', ['src'=>'path/to/image.jpg', 'alt'=>'test image']);
```

폼 관련 UI오브젝트

`config/xe.php` 에 지정된 UI오브젝트의 별칭 목록입니다.

```
'uiobject' => [
    'aliases' => [
        'form' => 'uiobject/xpressengine@form',
        'formText' => 'uiobject/xpressengine@formText',
        'formPassword' => 'uiobject/xpressengine@formPassword',
        'formTextarea' => 'uiobject/xpressengine@formTextArea',
        'formSelect' => 'uiobject/xpressengine@formSelect',
        'formCheckbox' => 'uiobject/xpressengine@formCheckbox',
        'formFile' => 'uiobject/xpressengine@formFile',
        'formImage' => 'uiobject/xpressengine@formImage',
        'formMenu' => 'uiobject/xpressengine@menuSelect',
        'formLangText' => 'uiobject/xpressengine@formLangText',
        'formLangTextarea' => 'uiobject/xpressengine@formLangTextArea',
        'langText' => 'uiobject/xpressengine@langText',
        'langTextArea' => 'uiobject/xpressengine@langTextArea',
        'menuType' => 'uiobject/xpressengine@menuType',
        'permission' => 'uiobject/xpressengine@permission',
        'themeSelect' => 'uiobject/xpressengine@themeSelect',
        'captcha' => 'uiobject/xpressengine@captcha',
    ]
],
```

위 별칭 목록 중에 `form*` 의 형식으로 등록된 UI오브젝트가 있습니다. 이 UI오브젝트들은 의미 그대로 폼을 구성할 때 사용할 수 있는 UI오브젝트들입니다. 이 UI오브젝트들은 테마나 스킨, 위젯와 같은 컴포넌트가 사이트 관리자에게 설정 폼을 출력할 때 사용됩니다. 폼 관련 UI오브젝트의 사용법은 폼 출력 문서에서 자세히 설명합니다.

폼 관련 UI오브젝트로 등록되는 UI오브젝트는 몇가지 규칙이 준수해야 합니다.

- 아래의 마크업 형식으로 출력해야 합니다. (이 마크업 형식은 bootstrap 형식을 따르고 있습니다.)

```
<div class="form-group">
    <label for=""></label>
    <!-- 실제 폼 요소를 여기에 작성 -->
    <p class="help-block"></p>
</div>
```

- `label` , `description` 파라미터를 받아서 처리할 수 있어야 합니다. `label` 은 폼요소의 라벨 문자열입니다. `<label for=""></label>` 에 출력되어야 합니다. `description` 은 `<p class="help-block"></p>` 에 출력될 폼 요소에 대한 설명입니다.
- `id` , `name` 필드를 파라미터로 받아서 처리할 수 있어야 합니다. 이 두 필드는 실제 폼 요소의 애트리뷰트로 지정됩니다. `id` 필드는 `label` 의 `for` 애트리뷰트의 값으로도 지정해주어야 합니다.
- `values` 파라미터를 받아서 처리할 수 있어야 합니다. `values` 는 폼 요소에 지정할 값(value)입니다. `values` 의 형식은 각 폼 요소마다 다릅니다.

토글메뉴

토글메뉴는 형태에 따라 3가지 타입을 지원합니다.

- MENUTYPE_EXEC
- MENUTYPE_LINK
- MENUTYPE_RAW

토글메뉴 생성

추가하고자 하는 새로운 토글메뉴 클래스에서 `AbstractToggleMenu` 를 상속 받습니다. 그리고 명시된 추상메서드를 구현합니다.

- `getText` : 메뉴가 펼쳐졌을때 보여지게될 문자열입니다.
- `getType` : `MENUTYPE_EXEC` , `MENUTYPE_LINK` , `MENUTYPE_RAW` 네가지 중 한가지를 반환해야 합니다.
- `getAction` : 해당 메뉴를 클릭했을때 실행 되어질 js 문자열입니다. 만약 타입이 `raw` 인 경우 html 이 반환되어야 합니다.
- `getScript` : 메뉴의 동작을 위해 특정 js 파일이 필요한 경우 해당 파일의 경로를 반환해 줍니다.

exec

`exec` 타입은 `getAction` 에 의해 반환된 문자열이 그 자체로 js 로 실행 가능한 형태를 가집니다. 특정 함수를 실행하는 경우 함수에서 필요로하는 인자는 해당 토글메뉴내에서 제공되어야 합니다.

```
public function getType()
{
  return static::MENUTYPE_EXEC;
}

public function getAction()
{
  return 'alert("hello")';
}
```

link

`link` 타입은 클릭시 다른페이지로 이동하는 메뉴입니다.

```
public function getType()
{
  return static::MENUTYPE_LINK;
}

public function getAction()
{
  return '/';
}
```

raw

`raw` 타입은 메뉴에 표현될 형태부터 실행될 방식까지 직접 지정하여 사용하는 방식입니다.

```
public function getType()
{
  return static::MENUTYPE_RAW;
}

public function getAction()
{
  return '<a href="#" onclick="method('argument')">Raw메뉴</a>';
}
```

다이나믹 필드 (draft)

사이트 관리자가 사용자 입력 필드를 추가할 수 있는 기능을 제공합니다. 회원, 게시판 관리 페이지에서 기능을 제공하고 있습니다. 입력 필드 추가 기능은 다른 서드파티 플러그인에서도 제공할 수 있습니다.

다이나믹 필드는 다양한 형태를 제공할 수 있도록 설계되었습니다. AbstractType 클래스로 데이터를 처리하기 위한 형태만 제한하고 다이나믹 필드 제작자가 유연하게 구현할 수 있도록 했습니다. 다이나믹 필드는 데이터를 처리하는 필드 타입과 출력에 필요한 처리를 담당하는 필드 스킨으로 구성 됩니다. 여기는 필드 타입에 대한 설명입니다.

AbstractType

다이나믹 필드를 만들때 사용되는 추상클래스 입니다. 모든 다이나믹 필드는 반드시 이 추상클래스를 사용해합니다. 이것은 각 구현체가 제공하기 위한 필요한 데이터베이스 테이블 컬럼의 정의와 데이터를 처리하는데 집중할 수 있도록 해줍니다.

```
class Address extends AbstractType {}

class Category extends AbstractType {}
```

데이터베이스 테이블 생성,삭제 그리고 데이터 등록,수정,삭제 및 검색에 필요한 요소를 구현했습니다. 제작자는 다이나믹 필드의 이름, 설명, 데이터베이스 테이블 컬럼 구성등의 정보만 처리하여 새로운 필드를 만들 수 있습니다.

빈 다이나믹 필드 생성 (draft)

다이나믹 필드 생성 커맨드를 사용하려면 우선 플러그인이 마련되어 있어야 합니다. 플러그인 생성은 플러그인 개발 시작하기를 참고 바랍니다.

아래 커맨드로 빈 다이나믹 필드를 만들 수 있습니다.

```
$ php artisan make:dyanmicField <path> <component_id> <title>
```

`path` 는 다이나믹 필드가 위치할 디렉토리 경로입니다. 플러그인 디렉토리 이름을 포함한 경로를 입력합니다. `component_id` 에는 다이나믹 필드의 아이디를 입력합니다. `title` 에는 관리자 사이트에서 표시 할 이름입니다.

컴포넌트 아이디

컴포넌트 아이디는 아래와 같은 규칙으로 작성합니다.

```
fieldType/<plugin_id>@<pure_id>
```

`plugin_id` 는 플러그인 디렉토리 이름이고 `pure_id` 는 다이나믹 필드의 아이디 입니다.

다이나믹 필드 등록

커맨드를 사용할 경우, 자동으로 등록됩니다. 플러그인의 `composer.json` 파일에 아래와 같이 컴포넌트 정보가 등록되어 있습니다.

```
// plugins/my_plugin/composer.json
...
  "extra": {
        "xpressengine": {
            "title": "my plugin",
            "component": {
                "fieldType/my_plugin@my_field": {
                    "class": "Sample\\MyPlugin\\MyField",
                    "name": "나의 다이나믹 필드",
                    "description": "나의 다이나믹 필드"
                }
            }
        }
    },

    ...
```

관리자 설정 페이지

다이나믹 필드를 생성할 때 사용자로 부터 입력값이 필요하다면 `getSettingsView()` 메소드를 구현 합니다. 제작자는 `$config` 를 이용해 설정 입력 폼을 추가할 수 있습니다. 카테고리 다이나믹 필드(`/app/FieldTypes/Category.php`) 클래스를 참고하세요.

다이나믹 필드 스킨 (draft)

사이트 관리자는 다이나믹 필드를 이용해서 사용자 입력 필드를 추가할 수 있습니다. 추가된 다이나믹 필드는 데이터베이스 테이블 구조 및 데이터 입출력을 제어하는 필드 타입과 디자인 요소를 처리하는 필드 스킨으로 구분됩니다.

필드 스킨은 등록, 수정, 검색 폼, 상세 보기 등의 디자인을 구성할 수 있도록 구조를 제공합니다.

AbstractSkin

스킨을 만들때 사용하는 추상클래스 입니다.

```
class DefaultSkin extends AbstractSkin {}
```

빈 스킨 생성 (draft)

스킨 생성 커맨드를 사용하려면 플러그인과 대상 다이나믹 필드가 마련되어 있어야 합니다. 플러그인 생성과 다이나믹 필드 생성은 다이나믹 필드 매뉴얼을 참고 바랍니다.

아래 커맨드로 스킨을 만들 수 있습니다.

```
$ php artisan make:dyanmicFieldSkin <path> <component_id> <title>
```

`path` 는 스킨이 위치할 디렉토리 경로입니다. 플러그인 디렉토리 이름을 포함한 경로를 입력합니다.

`component_id` 에는 스킨의 아이디를 입력합니다.

`title` 에는 관리자 사이트에서 표시 할 이름입니다.

컴포넌트 아이디

컴포넌트 아이디는 아래와 같은 규칙으로 작성합니다.

```
<field_type>/fieldSkin/<plugin_id>@<pure_id>
```

`field_type` 스킨이 사용될 대상 다이나믹 필드 아이디 입니다. 아이디는 대상 클래스에 `dd(SampleType:getId())` 로 확인할 수 있습니다.

`plugin_id` 는 플러그인 디렉토리 이름입니다.

`pure_id는` 스킨의 아이디 입니다.

스킨 등록

커맨드를 사용할 경우, 자동으로 등록됩니다. 플러그인의 composer.json 파일에 아래와 같이 컴포넌트 정보가 등록되어 있습니다.

```
// plugins/my_plugin/composer.json
...
  "extra": {
        "xpressengine": {
            "title": "my plugin",
            "component": {
                "fieldType/my_plugin@my_field/fieldSkin/my_plugin@my_skin": {
                    "class": "Sample\\MyPlugin\\MySkin",
                    "name": "나의 다이나믹 필드 스킨",
                    "description": "나의 다이나믹 필드 스킨"
                }
            }
        }
    },

    ...
```

디자인 파일 처리

AbstractSkin 추상 클래스는 제작자가 스킨구현에 집중할 수 있도록 blade 템플릿 파일에서 사용할 데이터를 처리하여 제공합니다. 제작자가 구현체 클래스에 `getPath()` 를 구현하여 블레이드 템플릿 파일이 있는 디렉토리 경로를 설정하면 이 추상 클래스는 스킨에서 사용해야할 설정과 데이터베이스 데이터를 전달합니다.

전달 데이터

AbstractSkin 이 View 로 전달하는 데이터는 아래와 같습니다.

- `$config` 다이나믹 필드를 만들때 관리자에서 입력한 설정값 입니다.
- `$data` 데이터베이스 테이블에 등록된 정보 입니다.
- `$key` 다이나믹 필드에서 정의한 데이터베이스 컬럼 이름과 실제 사용하는 입력 필드 이름의 정보입니다.

스킨에서 만들어야 하는 템플릿 파일은 아래와 같습니다.

- `create.blade.php` 회원 등록, 새글 작성에 사용합니다.
- `edit.blade.php` 회원 정보 수정, 글 수정에 사용합니다.
- `show.blade.php` 회원 정보 보기, 작성된 글 보기 페이지에 사용 합니다.
- `search.blade.php` 게시판 리스트의 상세 검색 폼에 사용합니다.
- `settings.blade.php` 관리자에서 다이나믹 필드를 생성할 때 설정 값을 입력 받기위해 사용합니다.

에디터

XE에서는 에디터를 추가하여 사용할 수 있고, 또한 에디터에서 사용할 도구를 추가할 수 있습니다.

에디터 생성

컴포넌트 추가

에디터가 사용되기 위해선 `AbstractEditor` 를 상속받은 컴포넌트가 생성되어야 합니다. 그리고 정의된 추상 메서드를 구현하세요.

- `getName` : js 스크립트에서 정의된 에디터의 이름입니다.
- `htmlable` : 해당 에디터가 wisywig 에디터 인지를 반환합니다.
- `compileBody` : 작성된 내용이 에디터를 통해 출력되기전 출력을 위한 처리동작을 수행합니다.

스크립트 load

에디터는 frontend 에서 동작하므로 js 파일을 필요로 합니다. 이때 해당 파일을 불러오기 위한 방법으로 다음과 같은 방식을 권장합니다.

이벤트

에디터는 표현되기 전에 처리하기 위한 이벤트 시점이 등록되어 있습니다. 이 이벤트 시점에 에디터에서 사용하고자 하는 스크립트 파일을 불러올 수 있습니다.

```
app('event')->listen('xe.editor.render', function ($editor) {
  app('xe.frontend')->js('path/to/editor.js')->load();
  app('xe.frontend')->js('path/to/editor.define.js')->before(['path/to/editor.js', 'assets/core/common/js/xe.editor.c
ore.js'])->load();
});
```

override

에디터는 `AbstractEditor` 의 `render` 메서드를 통해 표현됩니다. 에디터는 이 메서드를 구현하여 원하는 동작을 추가할 수 있습니다.

```
public function render()
{
  app('xe.frontend')->js('path/to/editor.js')->load();
  app('xe.frontend')->js('path/to/editor.define.js')->before(['path/to/editor.js', 'assets/core/common/js/xe.editor.c
ore.js'])->load();

  return parent::render();
}
```

> 에디터를 정의한 스크립트 파일은 반드시 XE의 `xe.editor.core.js` 를 필요로 합니다. 그러므로 XE core 의 스크립트가 해당 에디터 스크립트보다 먼저 불러질 수 있도록 하여야 합니다.

에디터 도구 생성

컴포넌트 추가

에디터 도구는 `AbstractTool` 을 상속받아 구현되어야 합니다. 그리고 다음 추상 메서드를 구현해야 합니다.

- `initAssets` : 도구가 동작하는데 필요로하는 파일들을 불러옵니다. (js, css)
- `getIcon` : 에디터 도구영역에 추가되기 위해 필요한 아이콘 파일입니다. 해상도에 따라 다른게 사용되기 위해 배열로 `normal` , `large` 두개의 파일 경로가 제공되어야 합니다.

- `compile` : 등록된 내용중 해당 도구로 작성된 영역을 표현하기 위한 동작을 수행합니다.

> frontend 관련 내용 추가요망 @blueng