edirectory

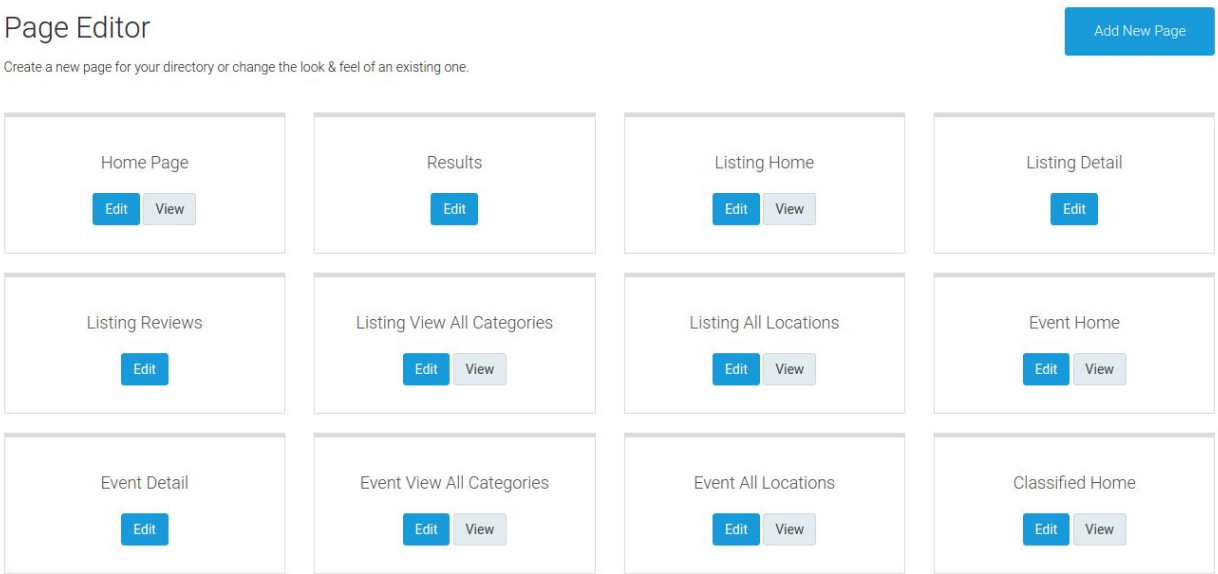
Developer Documentation

eDirectory 11.2
Page Editor
DevDocs

Introduction

The all-new Widget-based, Front-End Page Editor is the new eDirectory functionality available within the Site Manager to give greater creative control of the display layout and design of your eDirectory website, without any coding knowledge required. Create custom pages and add custom content for any pre-defined or one you've added yourself. You can also edit the page-specific SEO settings and create vanity URLs within the page editor.

Since the platform is widget-based, with the new Page Editor you'll be able to reorder the widgets of any page as you wish. You can add a custom widget to any page, remove it, and/or edit its content.



(Page Editor Main Menu)

What on Earth is a **WIDGET**?

A widget is a piece of code that allows users to define specific functionalities for a specific area of a webpage, displayed as a graphical user interface that allows sections of a site to be controlled independently. Still confused? More simply, and for eDirectory.com, a widget is nothing more than a horizontal block of content for a frontend page. Let's take a look:

Leaderboard Banner (728 x 90) as represented within the site manager.

It consists of only one block with a banner of the "Leaderboard" type, it's the simplest example of a widget we have, there is no customizable content whatsoever.

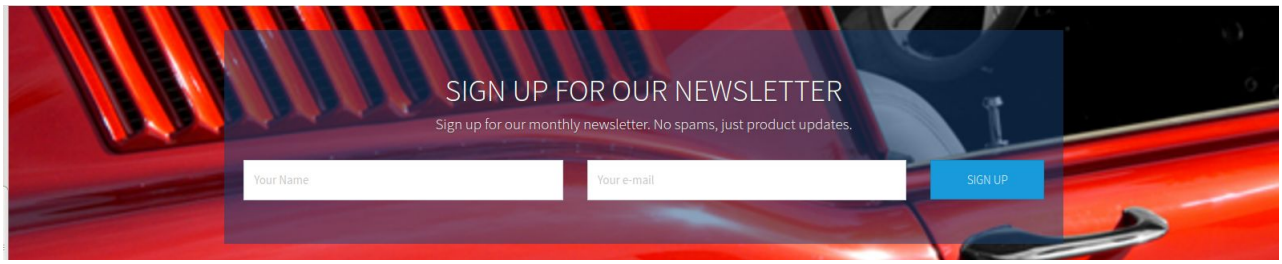
(view of Leaderboard Banner on the Front-End)

(Newsletter SignUp Widget as represented within the Site Manager)

The Newsletter widget already has content of its own, which can be easily

edited within site manager, like its title, description and background image.

Notice that there's an icon for edition on the right section, unlike the last example:



Look on the frontend.

There are other more complex widgets, like the "Header" widget, for example. Inside it, the site manager can edit all of the labels of the login navbar, as well as the items of the site navigation, its labels and links. The logo can also be edited.

* Important: When the site manager edits the content of a widget like the **"Header"** or **"Footer"**, he is editing that widget for ALL of the other pages as well. Unlike the other widgets that have different content for each different page. These are known as "Master Widgets".

**Important: The widgets that have only text fields editable display a checkbox in case the site manager wants to replicate all of the changes to the other pages as well. The widget *"Download our apps bar"* also follows the same rule. So you have the option to make this change extend to all other areas displaying the same widget.

Each widget has its own *twig* file, which contains its own *html code,* and those files are located on the **widgets** folder of each theme, for example, on the Restaurant theme: '/Resources/themes/restaurant/widgets'.

Every widget's position can be changed on the site manager, you just need to drag it to the desired position and save the changes. For example, we can move the *"Header"* widget below the *"Footer"* widget if desired.

The site manager can add or remove any widget from the pages. There are some exceptions when it comes to adding, because some widgets are only available for specific pages. For example: the "*Result content with left filters*" widget is only available on the **Results** page. The complete list of exceptions can be found on 'LoadWidgetPageTypeData.php'.

Some widget groups are limited to only one widget per group for that page. This limitation is due to the fact that some conflicts of javascript features could happen, or simply for not making sense adding more than one specific widget on the same page. For example the "*Listing Detail*".

The complete list of widget groups that are unable to be duplicated are found on the private var $widgetNonDuplicate on the *Wysiwyg Service* (Wysiwyg.php)

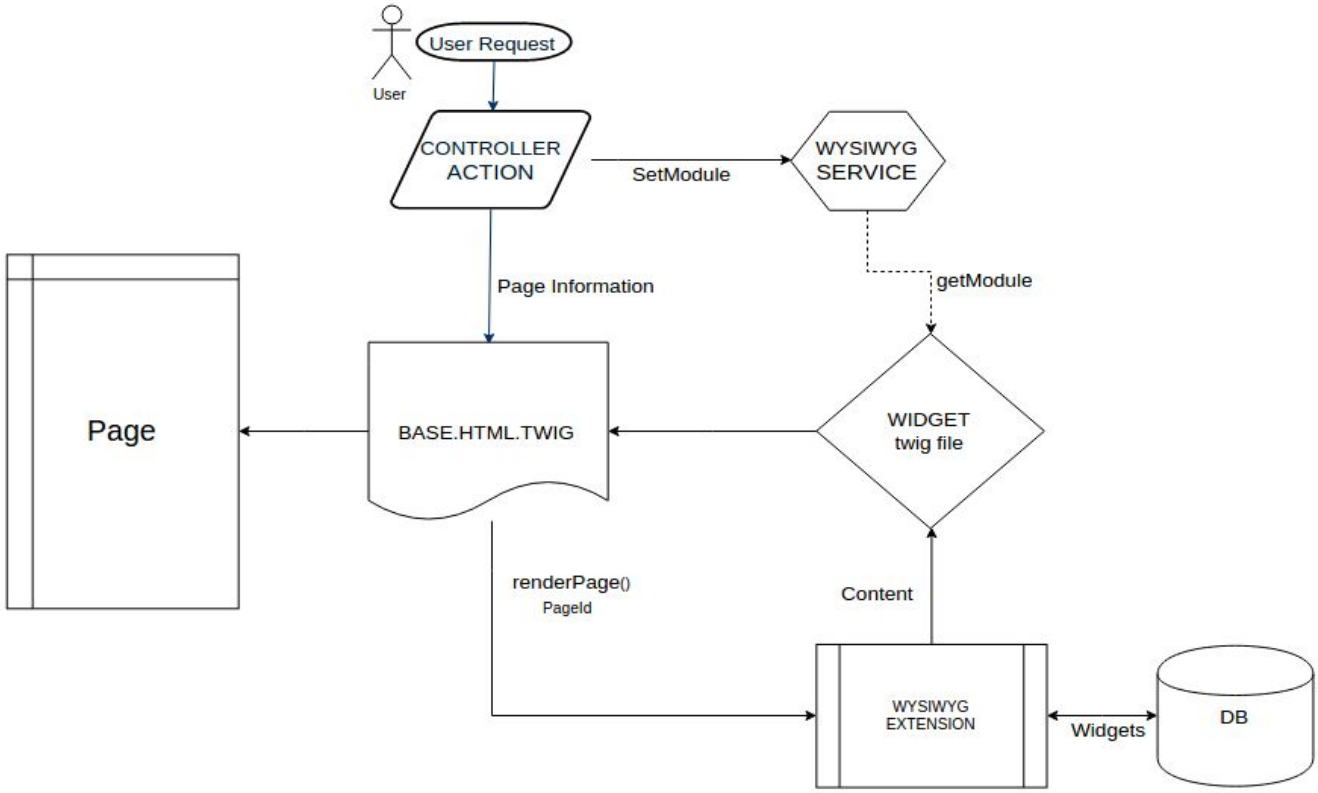
```php
/**
 * This array contains the groups of the widgets that can only be one of them per page,
 * if widget's name changing at the load data is necessary change here as well.
 * You can add the title of the widget to block the whole group, or create another group.
 *
 * @var array
 */
private $widgetNonDuplicate = [
    'header'            => [
        'Header',
        'Header with Contact Phone',
        'Navigation with Centered Logo',
        'Navigation with left Logo plus Social Media',
    ],
```

We're very excited to see what our community builds with this new Page Editor

The magic behind the Page Editor

Now let's talk about good stuff, the Page Editor Workflow:

- Frontend Workflow



With the new Page Editor, the front-end workflow has undergone some changes.

Now when the user makes a request to the system, it falls into the "Action of the Controller" responsible for the requested route and in it we inform the *Wysiwyg Service* which module that action belongs to (since some widgets need to know which module they belong to in order to correctly display their content). In addition to all banner-type widgets or those that contain a banner, there are more widgets that depend on the module, here are **some examples**:

- **Browse by category block with images**
- **Reviews block**
- **All Locations**

Basically widgets that have content that can vary depending on the module,

such as categories, locations, reviews.
When it comes to a page that does not belong to any module, *Wysiwyg Service* defaults to the listing module.
At the end of all Action we returned the Twig from that page and passed it to the page information (id, title, SEO). With the Page Editor changes, the Twig returned will be the **base.html.twig** in most cases

* Important: Only the Results Actions and Module Detail Actions remain using their respective **twig** and do not use **base.html.twig** like the others.

On **base.html.twig** the *renderPage* method of the *WysiwygExtension* class is called, passing the ID of the page. The *renderPage* method in turn retrieves from the database all the widgets that belong to that sorted page, as well as the content that the site manager has configured for that widget.

The *renderPage* follows the **order field** of each widget configured by the site manager and assembles the final page by adding the **twig files** of the path contained in the twig_file field of each widget and its **content** (content field).

* Important: The **content** that is passed to the **twig** file is that of the *PageWidget* entity. The **content** of the *Widget* entity will always be the **default** for that widget, and never the widget configured by the site manager.

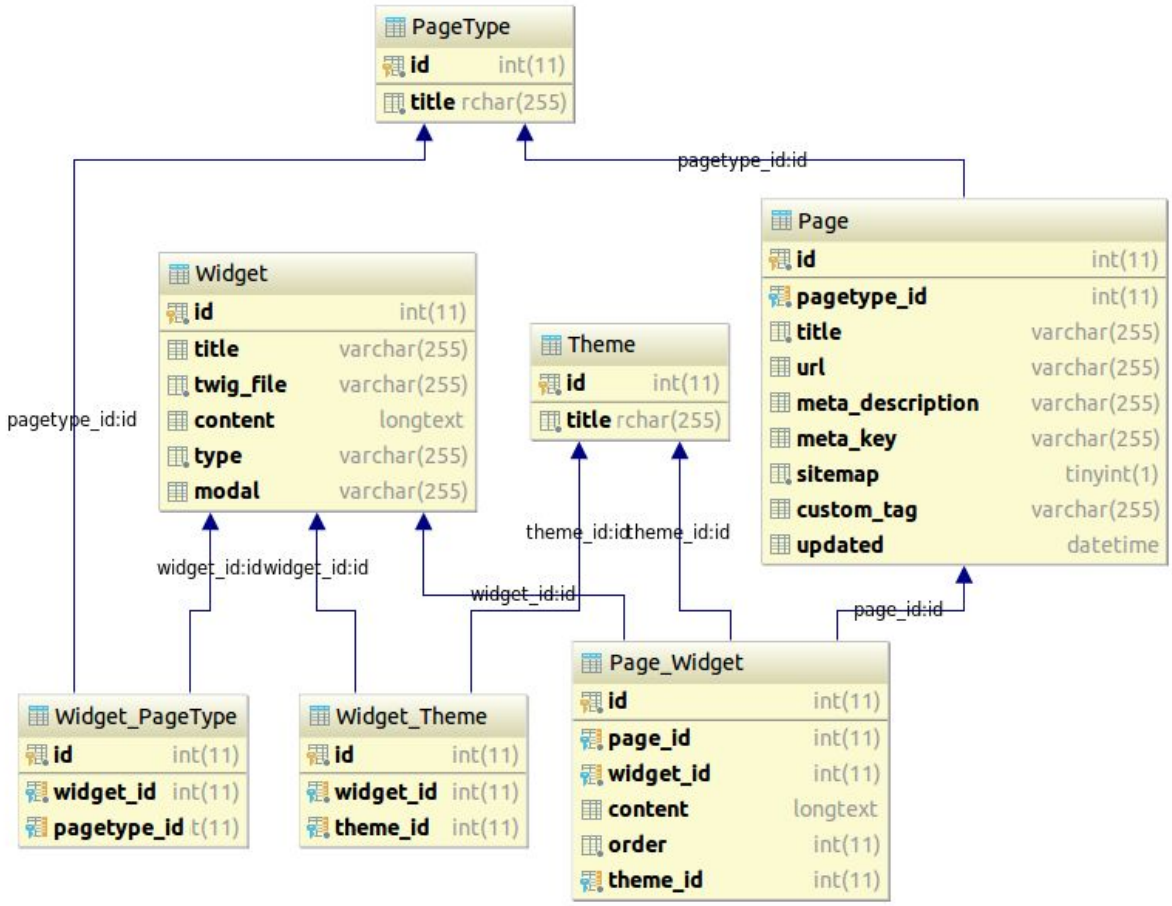
** Important: The old variables that were sent by the *Controller* to the **twig** of each *Action* are now global variables of the twigs, because as now to render we go through several twigs we need them as global references. Be careful not to use variables with the same name. Example of how to add a global variable:

```
$twig = $this->container->get("twig");

$twig->addGlobal('item', $item);
```

Only when *renderPage* has finished assembling all the widgets does it return the page to the user, marking the end of the cycle.

- *The Database*



An important part of all functionality is understanding your database, and with the new **Page Editor** this is just as important.

The first important point regarding the Page Editor database is to understand that there are two main entities, the *Widget* entity and *Page*. The whole idea of functionality revolves around these two entities.

The *Page* table replaced the old *Content* table, and had its fields related to

title and SEO incorporated by Page. The **type** column has become a new table, the *PageType*, which records all page types. These types are listed at the beginning of the Wysiwyg Service as constants.

For the **content** field of the *Content* table a 'Custom Content' widget has been created where the site manager can add the *html* content you want. So now the site manager can add *html* code anywhere on the page through the 'Custom Content' widget.

The *Widget* table concentrates the main information related to the widgets, among them: the **twig** file (twig_file field) that will be rendered when the widget is included in a page, what **content** is editable by the site manager, what is the modal id (**Modal**) that allows the edition of the widget for the site manager and which type (**type**) of the widget that defines in which group it will be embedded in the modal of adding widgets.

In the Widget_Theme table, which widget is available for which theme, because not all widgets are available for all themes.

The *Widget_PageType* table records the widgets that are unique to certain pages. That is, if a widget is unique, there will be a record in this table between the widget and the page type. Otherwise there will be only one record of the widget without page (null).

The *Theme* table contains the existing eDirectory themes.

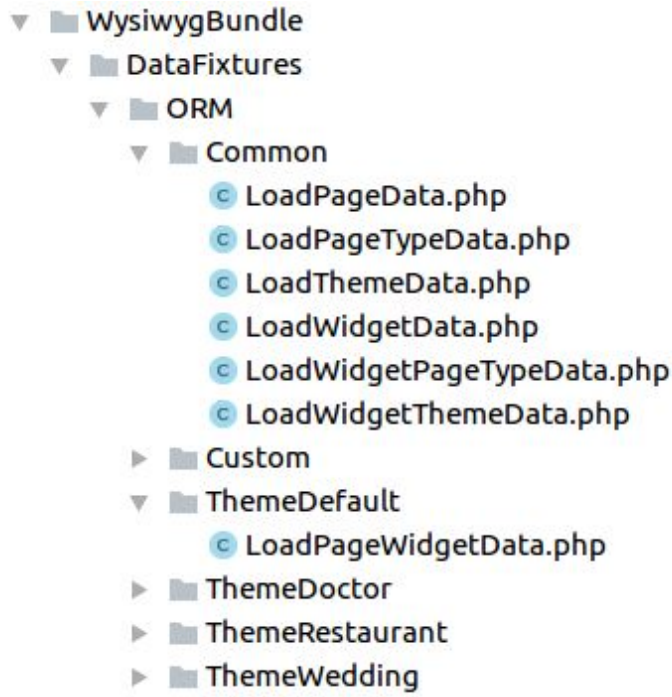
Finally, the Page_*Widget* table makes the relationship between the pages and the widgets, that is, each row in this table contains a record saying that a page (**page_id**) has a widget (**widget_id**) in a given order, and also the content (**Content**) configured by the site manager, for the theme (**theme_id**) indicated.

It is in this table that you can find out which widgets make up the *Home Page* in the *Default theme*, what is the order of them, and what is the content of each widget.

* Important: Widget images are not saved in the content **column**. They follow the structure of the previous version. As for example the "slider" that had a specific table for this.

- *Load Data*

In version 11.2 of eDirectory a new important bundle for the Page Editor named **DataFixtures** has been added, which is responsible for loading all database records related to the Page Editor.



Since almost all information about the Page Editor would be stored in the database, we added this bundle to leave the default content of the eDirectory pages registered.

The bundle transforms the objects that you create into it and inserts in the database. You define an order of insertion of classes to obey the dependencies of the bank for relationships..

To create relationships through LoadData you need to create a reference to the object you want to relate to another, so that the next class following the defined order knows how to find the object to create a relationship.

The complete bundle documentation can be found here on GITHUB.

- *Site manager*

The Page Editor is available on the site manager in the Design & Customization session. In this session the site manager can manage the eDirectory pages and their widgets.

Some important points about the Page Editor area:
- The javascript functionality of the Page Editor interface is concentrated in the files 'web/scripts/widgets.js' and 'sitemgr/assets/custom-js/widget.php'.

- Every editable widget calls a modal whose ID corresponds to the value of the **modal** field. Widgets that only have text fields to be edited use the same modal **'edit-generic-modal'** ID. And those that do not have editable content have the empty **modal** field.

- Every widget has a representation image of it for each theme. Pictures can be found at: 'web/sitemgr/assets/img/widget-placeholder'. It is important to note that the name of each image refers to the title of the widget passing through the function 'system_generateFriendlyURL', If there is no image for that widget the system uses the image of the "Custom Content" widget.

Another interesting point is the functionality of redefining a page, which allows the site manager to restore the widgets of a page to its default format. For this, it was added to the *Wysiwyg Service (*Wysiwyg.php) the default configuration of eDirectory pages. For each page there is a function that returns the default widgets of that page for each theme.

```php
/**
 * Returns the widgets that compose the Listing Home
 * Used for load data and reset feature at sitemgr
 *
 * @return mixed
 */
public function getListingHomeDefaultWidgets()
{
    $pageWidgetsTheme = [
        Theme::DEFAULT_THEME    => [
            'Header',
            'Search Bar',
            'Leaderboard ad bar (728x90)',
            '3 Featured Listings',
            'Browse by category block with images',
            'Best Of Listings',
            '3 rectangle ad bar',
            'Browse by Location with Right Banner (160x600)',
            'Banner Large Mobile, one banner Sponsored Links and one Google Ads',
            'Download our apps bar',
            'Footer',
        ],
```

This is an excerpt from the function that returns the default widgets on the "Listing Home" page.
The full function of each page can be found in the *Wysiwyg Service (*Wysiwyg.php).

```php
/**
 * Returns all the pages that have some widget that has any content different from its default
 * USED IN LOAD DATA
 *
 * @return array
 */
public function getDefaultSpecificWidgetContents()
{
    $translator = $this->container->get('translator');
    $language = substr($this->container->get("multi_domain.information")->getLocale(), 0, 2);

    // Set specific contents
    $contents = [];
    $contents[Wysiwyg::EVENT_HOME_PAGE]['Search Bar'] = json_encode(['labelExploreAndFind' => 'Explore and find Events']);
    $contents[Wysiwyg::CLASSIFIED_HOME_PAGE]['Search Bar'] = json_encode(['labelExploreAndFind' => 'Explore and find Classifieds']);
```

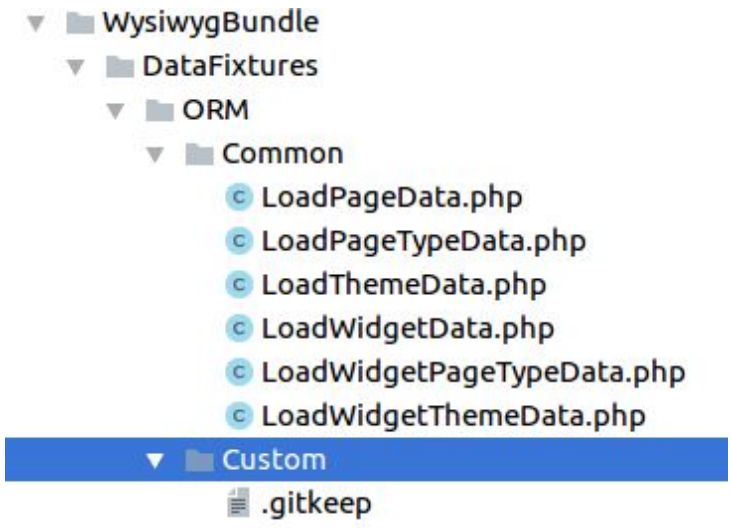
* Important: Some pages have widgets with content that is different from that widget's default. The complete list can be found in the 'getDefaultSpecificWidgetContents' at *Wysiwyg Service (*Wysiwyg.php).

Customizing the Page Editor

In order to customize the Page Editor, we must take into account the fact that most of the information is stored in the **database**, so it is possible to create widgets only by inserting your information in the database and adding your twig, for example. But this is not the best way or the focus of this guide.

The main customizations involving Page Editor would be the creation of new widgets, pages or themes. The best way to do this is through *LoadData*, so everything gets registered in the code and not just in the database.

So for any of the following customizations, add a widget, page or theme, you need to copy the files from the *LoadData* **Common** folder into the **Custom** folder.



* Important: Do not forget to fix the namespace of the copied files

- *Adding a Widget*

For this guide we will use for example the creation of the. **'Popular Listings'** widget.

To get started you need to create your *twig* file **'popular-listings.html.twig'** in the "listing" subfolder contained in the "widgets" folder of each theme, for example for the default theme would be in '/Resources/themes/**default**/widgets/listing', unless the new widget has theme exclusivity, then only needed for the themes in which it appears.

Next we need to put the widget information in the LoadData bundle.

In the newly copied file 'LoadWidgetData.php', at the end of the *array* $standardWidgets there is a sample comment on how to add a widget.
If in doubt, you should create a new node in the array with the information of the new widget and its default content. Example:

```
[
    'title'     => 'Popular Listings',
    'twigFile' => '/listing/popular-listings.html.twig',
    'type'      => Widget::LISTING_TYPE,
    'content'   => [
        'labelPopularListings' => 'Popular Listings',
    ],
    'modal'     => 'edit-generic-modal',
],
```

With this, we define the title, the *twig* file, the type to know in which tab it will be made available, the content with a single *label*, and how it's editable element will be *label* (s), the generic **modal** of the new widget **'Popular Listings'**.

* Important note: If the widget is more complex and the site manager can edit more things besides just *label* (s) you need to create a specific modal for this new widget in the widget modals folder in the site manager area: '/web/includes/modals/widget'.

At the new file, 'LoadWidgetPageTypeData.php' we'll add if necessary, a node at the end of the *array* $exceptionsWidgets with the pages that this new widget can be added. If the new widget is available for all pages skip this

step.

```
'Popular Listings'                    => [
    $this->getReference("TYPE_".Wysiwyg::HOME_PAGE),
    $this->getReference("TYPE_".Wysiwyg::LISTING_HOME_PAGE),
],
```

In our example, we defined that the new widget, **'Popular Listings'**. It will be exclusive of **"Home Page"** and **"Listing Home"**.

Now to set in which themes the new widget will be available, you must add a new node with the title of the new widget in the return array in the functions that define which widgets of each theme in the widget *Wysiwyg Service (*Wysiwyg.php*).*
Each theme has its function listing your widgets, and there is also one for the widgets common to all themes.

```
/**
 * Returns the commons and the Default Theme widgets
 *
 * @return array
 */
public function getDefaultThemeWidgets()
{
    $trans = $this->container->get('translator');

    return array_merge($this->getCommonThemeWidgets(), [
        $trans->trans('Header', [], 'widgets', 'en'),
        $trans->trans('Footer', [], 'widgets', 'en'),
        $trans->trans('Popular Listings', [], 'widgets', 'en'),
        /*
         * CUSTOM ADDWIDGET
         * here are an example of how you add the widget 'Widget test' for Default theme
         * if you need that 'Widget test' to be available for all themes you have
         * to remove it from here and add at the right function above
         *
         * $trans->trans('Widget test', [], 'widgets', 'en'),*/
    ]);
}
```

In this case we leave our new widget **'Popular Listings'** available only at the **default theme**. If it needs to be common to all the themes it just needs to be added on the function 'getCommonThemeWidgets' and return.

Okay, now you only have to execute the command in the terminal for *LoadData* to insert the new data, **do not forget to put the correct domain**.

```
php app/console doctrine:fixtures:load
--fixtures=src/ArcaSolutions/WysiwygBundle/DataFixtures/ORM/Custom
--append --domain=your.domain.com
```

The following messages should be returned if all steps have been done correctly:

```
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetData
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadThemeData
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadPageTypeData
> loading [2] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadPageData
> loading [2] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetThemeData
> loading [4] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetPageTypeData
```

DONE! The widget **'Popular Listings'** has been added.

- *Adding a new Page*

As an example, we'll use the creation of the page **"Categories Home"**, Considering that a route has already been added, a *Controller* and an *Action* has been created, and everything else it took to add a new page to the eDirectory front.

Adding a new page is very similar to adding a widget. Just like adding a new widget, you need to copy the files from the **Common** folder of *LoadData* to the **Custom** folder.

* Important: Do not forget to fix the *namespace* of the copied files.

Before adding something to LoadData we need to create a constant for the type of this new page. Every page has its type listed as constant in the *Wysiwyg Service (*Wysiwyg.php). Add the new type at the end of the list:

```
const BLOG_HOME_PAGE = "Blog Home";
const BLOG_DETAIL_PAGE = "Blog Detail";
const BLOG_CATEGORIES_PAGE = "Blog View All Categories";
/**
 * CUSTOM ADDPAGETYPE
 * here are an example of how you add a PageType constant to be used in the load data
 */
/*const TEST_PAGE = "Test Page";*/
const CATEGORIES_HOME = "Categories Home";
```

At the newly copied file 'LoadPageTypeData.php' add a new page type to a new node at the end of the *array* '$standardPageTypes'.

```
[
    'title' => Wysiwyg::ITEM_UNAVAILABLE_PAGE,
],
/*
 * CUSTOM ADDPAGETYPE
 * here are an example of how you add a PageType to be used in LoadPageData
 */
/* [
    'title' => Wysiwyg::TEST_PAGE,
],*/
[
    'tittle' => Wysiwyg::CATEGORIES_HOME,
]
```

At the newly copied file 'LoadPageData.php' add the new page information to

a new node at the end of the array '$standardPages'.

```
/**
 * CUSTOM ADDPAGE
 * Here are an example of how you add a page,
 * and you will need to create a PageType for this page
 *
 * Don't forget to create the alias for the url, if needed
 */
[
    'title'     => $trans->trans('Categories Home', [], 'widgets', 'en'),
    'url'       => $this->container->getParameter('alias_categories_home_url_divisor'),
    'metaDesc'  => '',
    'metaKey'   => '',
    'sitemap'   => false,
    'customTag' => '',
    'pageType'  => $this->getReference("TYPE_".Wysiwyg::CATEGORIES_HOME),
],
```

The next step is to create the function that defines the default page widgets for each theme. At the *Wysiwyg Service (*Wysiwyg.php) create a function by using the constant with the new page type that was added as part of the function name, because the page reset functionality picks up the function according to the page type.

```
/**                                                                                                                    *
 * Returns the widgets that compose the Categories Home
 * Used for load data and reset feature at sitemgr
 *
 * @return mixed
 */
public function getCategoriesHomeDefaultWidgets()
{
    $pageWidgetsTheme = [
        Theme::DEFAULT_THEME    => [
            'Header',
            'Search Bar',
            'Leaderboard ad bar (728x90)',
            'Browse by category block with images',
            '3 rectangle ad bar',
            'Banner Large Mobile, one banner Sponsored Links and one Google Ads',
            'Download our apps bar',
            'Footer',
        ],
        Theme::DOCTOR_THEME     => [...],
        Theme::RESTAURANT_THEME => [...],
        Theme::WEDDING_THEME    => [...],
    ];

    return $pageWidgetsTheme[$this->theme];
}
```

Important: The name of this function should always obey as follows:
'get' + '**value of the page type constant with no spaces**' + 'DefaultWidgets'.

```
/* Get Default Widgets Method */
$method = 'get'.str_replace(' ', '', $pageType).'DefaultWidgets';
$pageWidgetsArr = $this->$method();
```

** Important: Do not forget to create the array with the page widgets for each theme, and each theme has its *Header* and *Footer* widget.

Once you have created the function that defines the default of the widgets of the page you need to list it there for the function 'getAllPageDefaultWidgets' used on *LoadData.*

```php
/**
 * Returns an array with all the standard pages and its own array of default widgets
 * USED IN LOAD DATA
 *
 * @return array
 */
public function getAllPageDefaultWidgets()
{
    $pagesDefault = [];
    $pagesDefault[Wysiwyg::CATEGORIES_HOME] = $this->getHomePageDefaultWidgets();
    $pagesDefault[Wysiwyg::HOME_PAGE] = $this->getHomePageDefaultWidgets();
```

If one of the widgets in this new page has different content only for this page, you must define its different content in the function 'getDefaultSpecificWidgetContents'.

```php
/**
 * Returns all the pages that have some widget that has any content different from its default
 * USED IN LOAD DATA
 *
 * @return array
 */
public function getDefaultSpecificWidgetContents()
{
    $translator = $this->container->get('translator');
    $language = substr($this->container->get("multi_domain.information")->getLocale(), 0, 2);

    // Set specific contents
    $contents = [];
    $contents[Wysiwyg::CATEGORIES_HOME]['Search Bar'] = json_encode(['labelExploreAndFind' => 'Explore and find Categories']);
    $contents[Wysiwyg::EVENT_HOME_PAGE]['Search Bar'] = json_encode(['labelExploreAndFind' => 'Explore and find Events']);
```

Okay, now you only have to execute the command in the terminal for *LoadData* to insert the new data, **do not forget to put the correct domain**.

php app/console doctrine:fixtures:load --fixtures=src/ArcaSolutions/WysiwygBundle/DataFixtures/ORM/Custom --append --domain=your.domain.com

The following messages should be returned if all steps have been done correctly:

```
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetData
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadThemeData
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadPageTypeData
> loading [2] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadPageData
> loading [2] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetThemeData
> loading [4] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetPageTypeData
```

Done, the new page, **'Categories Home'**, was added. Finally, when editing the new page inside the Page Editor, click reset the page for the default widgets to be added.

- *Add a new Theme*

For this guide we will use for example the creation of the new theme, **School**, considering that all the previous process of creation of a new theme was followed in accordance with this tutorial.

The first step is to create a constant with the title of the new theme in the entity *Theme*(Theme.php).

```php
/**
 * Existing themes titles of eDirectory
 */
const DEFAULT_THEME = 'Default';
const DOCTOR_THEME = 'Doctor';
const RESTAURANT_THEME = 'Restaurant';
const WEDDING_THEME = 'Wedding';
const SCHOOL_THEME = 'School';
```

At the newly copied 'LoadThemeData.php' add a new node at the array '*$standardThemes*' with the title constant of the new theme.

```php
/* Theme title is used as reference in LoadPageWidgetData,
 *  so if you change here don't forget to change there
 **/
$standardThemes = [
    Theme::DEFAULT_THEME,
    Theme::DOCTOR_THEME,
    Theme::RESTAURANT_THEME,
    Theme::WEDDING_THEME,
    Theme::SCHOOL_THEME,
    /**
     * CUSTOM ADDTHEME
     * here are an example of how you add the theme 'Test'
     *
     * Theme::TEST_THEME,   */
];
```

Unlike the other tabs, to add a new Theme you also need to create a new folder in *LoadData* for the new theme. The folder must be created inside the '**ORM**' with the name of '**Theme**' plus the title of the new theme, in our example it will be '**ThemeSchool**'.

Inside the new folder you need to create a copy of the file 'LoadPageWidgetData.php' of the folder **ThemeDefault**. And in it change all the flames of the tem constant **Default**, to the new theme. **Example**:

```
$repository = $manager->getRepository('PageWidget');
$wysiwyg = $this->container->get('wysiwyg.service');
$wysiwyg->setTheme(Theme::SCHOOL_THEME);
$pagesDefault = $wysiwyg->getAllPageDefaultWidgets();
```

Now there is a hard part to do in *Wysiwyg Service (*Wysiwyg.php).

The first part is simple, you need to create a function that returns an array with all the widgets that will be available in this new theme. For example let's say that our new theme **'School'** is a copy of **Default**, therefore, it will have all the widgets of this theme in common. The function would look like this.:

```
/**
 * Returns the commons and the School Theme widgets
 *
 * @return array
 */
public function getSchoolThemeWidgets()
{
    $trans = $this->container->get('translator');

    return array_merge($this->getCommonThemeWidgets(), [
        $trans->trans('Header', [], 'widgets', 'en'),
        $trans->trans('Footer', [], 'widgets', 'en'),
    ]);
}
```

The important thing is to return all widgets common to all themes plus the unique ones of this new theme. The function name consists of 'get' + '**value of the constant of the new theme**' + 'ThemeWidgets'. Like in our example: 'getSchoolThemeWidgets'.

The second part is more laborious, you need to create a new node in the array of each function that lists the default widgets of each page of the system. This node should list the default page widgets for that new theme. In the example below we make this addition in the "*Custom Page*" function.

Okay, now you only have to execute the command in the terminal for LoadData to insert the new data, **do not forget to put the correct domain**.

Note that there is **an additional parameter where the folder path created for this theme is located**, in our example the theme **'School'** :

```
php app/console doctrine:fixtures:load
--fixtures=src/ArcaSolutions/WysiwygBundle/DataFixtures/ORM/Custom
--fixtures=src/ArcaSolutions/WysiwygBundle/DataFixtures/ORM/ThemeSchoo
l --append --domain=your.domain.com
```

The following messages should be returned if all steps have been done correctly:

```
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadThemeData
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetData
> loading [1] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadPageTypeData
> loading [2] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetThemeData
> loading [2] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadPageData
> loading [3] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\ThemeSchool\LoadPageWidgetData
> loading [4] ArcaSolutions\WysiwygBundle\DataFixtures\ORM\Custom\LoadWidgetPageTypeData
```

And...done! **'School'** theme was added!