Formation Drupal 8 Développement Avancé

Animée par Romain DORTIER

Fichiers de base d'un module

- Les fichiers suivants sont à placer à la racine du module dans le répertoire */modules/toto/* :
  - ***toto.info.yml*** : métadonnées (nom, description, version…) du module. Ce fichier suffit pour figurer dans la liste des modules dans le back-office de Drupal. **OBLIGATOIRE**
  - ***toto.module*** : code PHP du module (principalement les fonctions de hook).
  - ***toto.install*** : code d'installation, de désinstallation et de mise à jour du module.
- Le nom machine d'un module ne doit contenir que des **lettres minuscules** et des **underscores**. Pas de chiffres !

Fichier *.info.yml*

```yaml
# Nom du module.
name: Hello
# Description du module.
description: 'Hello module for D8 developer training.'
# Type de code (module, theme).
type: module
# Version de Drupal pour laquelle le module est développée.
core: 8.x
# Version du module.
version: 8.x-1.0
# Groupe du module.
package: Training
# Dépendances.
dependencies:
  - views
  - field
```

Fichier *.info.yml*

- Les propriétés **obligatoires** sont :
  - name
  - type
  - core
- En indiquant « VERSION » comme valeur de « version », Drupal affiche la version courante du système (par exemple 8.x-0.1).
- Les propriétés « description » et « package » sont rendues automatiquement **traduisibles**.

Entités

Entité de configuration
Entité de contenu

Configurations et données

- Chaque formulaire de paramétrage du site fait l'objet d'une **configuration simple**. Exemple :
  - formulaire comme sur *Admin > Configuration > Performance*.
- Certaines configurations créent de multiples objets - On parle alors de **Configuration complexe (Config Entity)**. Exemples :
  - vues
  - rôles utilisateur
  - styles d'image
  - blocs
  - …
- Le contenu principalement généré par les utilisateurs - **Content Entity**. Exemples :
  - noeuds
  - utilisateurs
  - termes de taxonomie
- Etat de l'environnement - **State API**. Exemple :
  - la date de la dernière exécution des tâches planifiées.

Entity API

- Avec **Drupal 8** tout est **entité** : les noeuds, les utilisateurs, les blocs, les styles d'image…
- **Entité créée par un utilisateur** : c'est du contenu, comme par exemple les :
  - Noeuds
  - Termes de taxonomy
  - Utilisateurs
- **Entité créée par un administrateur/webmaster** (fait partie de la construction/structure du site) : il s'agit de configuration, comme par exemple les :
  - Vues
  - Styles d'image
  - Rôles utilisateur
- On dispose d'APIs dédiées :

  Drupal\Core\Entity\ConfigEntityBase

  Drupal\Core\Entity\ContentEntityBase

Entity API

- Chaque type d'entité à son propre système de stockage. Il faut l'utiliser, et ne pas faire de chargement/modification à la main.
- Pour récupérer l'objet stockage correspondant par exemple aux noeuds :

  $storage = \Drupal::entityTypeManager()->getStorage('node').
- De façon générale on a pour tous les types d'entité :

$storage = \Drupal::entityTypeManager()->getStorage('type_entité').

- Pour récupérer tous les identifiants d'un type d'entité :

  $ids = \Drupal::entityQuery('entity_type')->execute();
- Il est possible d'ajouter des conditions (« ->condition() ») sur les champs à la requête précédente.
- Pour charger tous les objets correspondants :

  $entities = $storage->loadMultiple($ids);
- On obtient ainsi un tableau d'objets.
- ***Remarque*** : l'objet entityQuery est disponible dans ***$entityQuery = \Drupal::entityTypeManager()->getStorage('entity_type')->getQuery()***.

Annuaire d’annonces

Pourquoi ne pas utiliser les noeuds ?

- On désire mettre en ligne des annonces.
- Les noeuds ont des propriétés par défaut comme « *Promu en page d'accueil* », « *Epingler en tête de liste* » qui ne sont pas nécessaires pour les annonces.
- Afin d'optimiser la base de donnée et de pouvoir personnaliser le comportement de nos données, il est donc préférable de définir sa propre structure. On peut ainsi reprendre la main en terme d'accès, d'affichage, de création, de cache…

Qu’est-ce qui est nécessaire ?

- Il nous faut **définir** les champs de base dont on a besoin, par exemple un identifiant unique, un titre…
- On a besoin de **créer les formulaires** d’ajout et mise à jour de ces contenus.
- Chaque contenu doit avoir **sa propre page d’affichage**, comme c’est le cas avec les noeuds. Il faut définir également le chemin d’accès correspondant.
- Afin de retrouver les contenus, on va **ajouter une liste** dans le back-office de ***Drupal***. Il faut également définir son chemin d’accès.
- Pour administrer ce nouveau type d’entité, on va **créer une page de réglage** avec comme pour tout type d’entité les onglets *Gérer les champs*, *Gérer l’affichage du formulaire* et *Gérer l’affichage*.

Entity API

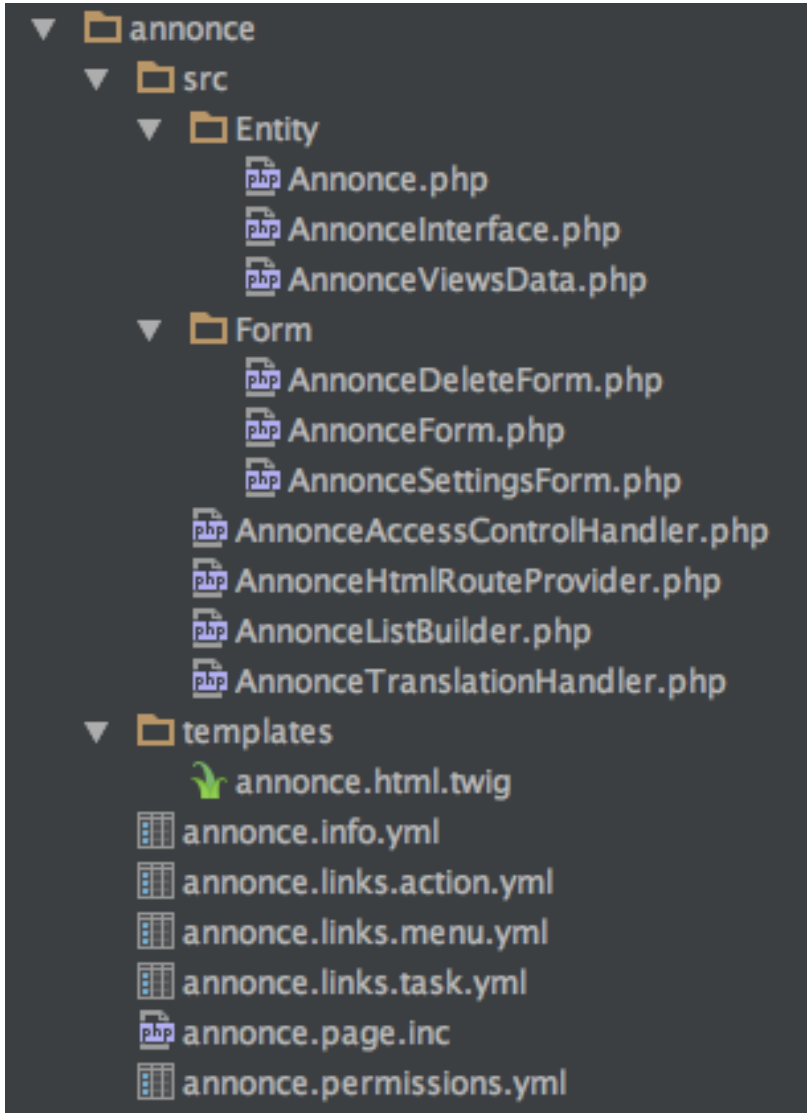
- En se basant sur la *Content Entity API*, on bénéficie de nombreuses fonctionnalités automatiquement :
  - Définition des **Champs de base** : ID, UUID, UID, name, langcode, created et changed. Ces champs ne sont pas obligatoires.
  - Possibilité d'**ajouter des champs** via Drupal UI, comme pour les types de contenu.
  - **Intégration avec Views**, fournissant un outil puissant de listing.
  - **Hooks d'altération** disponibles pour tous les autres modules désirant modifier le comportement de base : hook_ENTITE_insert(), hook_ENTITE_load(), hook_ENTITE_view(), hook_ENTITE_access(), hook_ENTITE_create().
  - Gestion du **contrôle d'accès**.
  - Gestion du **cache**.
  - Possibilité de définir des **templates** par défaut.

Création du module

Génération du code

- Le code est généré avec **Console** (*www.drupalconsole.com*), qui créer automatiquement tous les fichiers nécessaires (ou presque). ***Console*** est un outil en ligne de commande permettant d'administrer le site et de générer du code.
- Le **code** auto-généré est **fonctionnel**, mais peut être également modifié/amélioré afin de coller davantage aux besoins spécifiques.
- Le module généré se trouve dans l'archive ***advanced-d8/annonce.zip***.

Fichiers générés



Définition de notre type d'entité *Annonce*

- Tout type d'entité doit déclarer une interface et l'utiliser pour créer la classe de base. Cette dernière définissant un type d'entité étend la classe *ContentEntityBase* et implémente l'interface *AnnonceInterface* (fichier ***/src/Entity/AnnonceInterface.php***).
- Le fichier de définition de l'entité se trouve dans ***/src/Entity/Annonce.php***. La classe correspondante est donc **Annonce**.
- On déclare principalement :
  - sous forme d'**annotation** : le nom machine, la table de base, les classes handler (contrôle d'accès, affichage…), les formulaires de création/modification, les routes (affichage, suppression, modification, liste).
  - sous forme de **méthode** : les getters/setters et les champs de base.

Routing

- On définit un certain nombre de routes pour afficher et administrer ce type d'entité.
  - **Affichage** d'une annonce
  - **Liste** des annonces
  - **Formulaires** d'ajout, de modification et de suppression

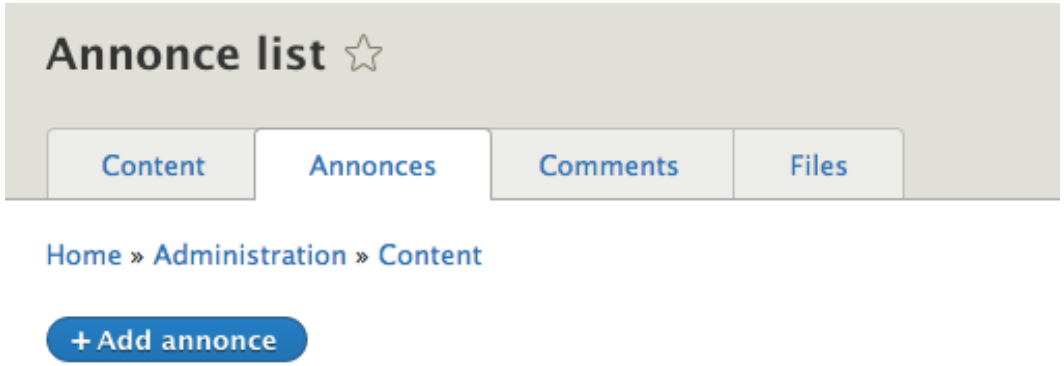
```
/**
 * Defines the Annonce entity.
 *
 * @ingroup annonce
 *
 * @ContentEntityType(
 *   id = "annonce",
 *   label = @Translation("Annonce"),
 *   handlers = {
 *     "view_builder" = "Drupal\Core\Entity\EntityViewBuilder",
 *     "list_builder" = "Drupal\annonce\AnnonceListBuilder",
 *     "views_data" = "Drupal\annonce\Entity\AnnonceViewsData",
 *     "translation" = "Drupal\annonce\AnnonceTranslationHandler",
 *
 *     "form" = {
 *       "default" = "Drupal\annonce\Form\AnnonceForm",
 *       "add" = "Drupal\annonce\Form\AnnonceForm",
 *       "edit" = "Drupal\annonce\Form\AnnonceForm",
 *       "delete" = "Drupal\annonce\Form\AnnonceDeleteForm",
 *     },
 *     "access" = "Drupal\annonce\AnnonceAccessControlHandler",
 *     "route_provider" = {
 *       "html" = "Drupal\annonce\AnnonceHtmlRouteProvider",
 *     },
 *   },
 *   base_table = "annonce",
 *   data_table = "annonce_field_data",
 *   translatable = TRUE,
```

Contrôle d'accès

- Contrôle d'accès pour **les entités** *annonce* (_entity_access) : view, edit, delete.
- Contrôle d'accès à la **création d'une entité** *annonce* (_entity_create_access) : annonce.
- Permissions :
  - **administer annonce entities** -> Destinée à administrer le paramétrage de l'entité.
  - **add annonce entities** -> Création d'annonces.
  - …
- Field UI :
  - Administrer les champs de l'entité.
  - Administrer l'affichage des champs de l'entité.
  - Administrer l'affichage des champs du formulaire de création/édition de l'entité.

Adapter le code généré

- **Modifier** les chemins des entités et le lien vers la page de listing dans le fichier de routing (*/src/AnnonceHtmlRouteProvider.php*) et de définition de l'entité ***Annonce***.
- **Cacher** le label du champ *Name* dans le fichier ***/src/Entity/Annonce.php***.
- **Modifier le lien** vers les annonces dans le listing (*Admin > Content > Annonces*) de sorte que l'on soit diriger vers la page de l'annonce et non le formulaire d'édition.
- Faites-en sorte que **seul l'auteur** d'une annonce puisse la modifier et la supprimer. Vous ajouterez pour ce faire les permissions adéquates et la logique de contrôler d'accès.
- **Créer un lien** sous forme d'onglet (*/annonce.links.task.yml*) vers la page de listing. On souhaite obtenir ceci :



Gestion de la configuration

- Toute la **configuration** du site est stockée **en base**.
- Toute configuration (type de contenu, champ, vue, style d'image…) peut être extraite sous forme de fichier YAML. Cela permet de déployer son site sur différents serveurs (test, recette, pre-production, production par exemple).
- Un module peut ainsi embarquer un certain nombre de configurations, qui sont importées dans le site lors de l'installation du module.
- Tout changement via le back-office de ces configurations est effectué en base et jamais dans les fichiers du module original.

Ajouter un champ à un entité

- Ajouter le champ à l'entité de contenu via l'interface de *Drupal*. Le chemin dans le back-office dépend du type d'entité. Par exemple pour les noeuds cela se fait via *Admin > Structure > Types de contenu > TYPE > Gérer les champs.*
- Exporter les fichiers YAML correspondants en allant sur *Admin > Configuration > Développement >* ***Synchronisation de configuration***.
- Intégrer les fichiers à son propre module. Les placer dans le répertoire */config/optional* (à créer s'il n'existe pas). Ces fichiers seront parsés à l'installation du module, et la configuration correspondante sera enregistrée en base.

Ajouter un champ images

- Via le back-office de **Drupal** **ajouter un champ** ***Images*** au type d'entité ***Annonce***, permettant d'avoir un nombre indéterminé d'images sur une annonce.
- Sur *Admin > Configuration > Développement > Configuration management*, **exporter la configuration** correspondante au champ précédent :
  - **Field** : définition du champ.
  - **Field storage** : instance du champ pour l'entité Annonce.
  - **Entity view display** : paramètrage de l'affichage des champs.
  - **Entity form display** : paramètrage de l'affichage des champs dans le formulaire de création.
- **Copier le code** dans les fichiers correspondants :
  - */config/optional/field.field.annonce.annonce.field_images.yml*.
  - */config/optional/field.storage.annonce.field_images.yml*.
  - /*config/optional/core.entity_view_display.annonce.annonce.default.yml*.
  - /*config/optional/core.entity_form_display.annonce.annonce.default.yml*.
- **Désinstaller/installer** votre module et vérifier que le champ ***Images*** est effectivement bien attaché aux annonces.

Event Dispatcher

Event Dispatcher

- Le système d'événement repose sur le composant **Symfony** **EventDispatcher**.
- Le pattern est similaire aux hooks de ***Drupal***. On désire pouvoir intervenir lors d'événements bien particuliers : création d'un noeud, envoie d'un email…
- Pour des raisons de performance, la liste des listeners est en cache dans le service *event_dispatcher* du **Dependency Injection Container** (ou container).

Event Dispatcher

- Afin de s'abonner à un événement, il faut déclarer un nouveau **tagged service** « event_subscriber ».
- Pour récupérer la liste des listeners :

$eventDispatcher = \Drupal::service('event_dispatcher')->getListeners();.

- Ce service correspond à une classe implémentant l'interface **EventSubscriberInterface**. Cette interface ne comporte qu'une seule méthode ***getSubscribedEvents()***.

```php
/**
 * {@inheritdoc}
 */
static function getSubscribedEvents() {
  $events[KernelEvents::REQUEST][] = array('methodeAppelee');
  return $events;
}
```

Event Dispatcher

- On souhaite afficher un message à tous les utilisateurs sur toutes les pages :
  - **Déclarer** un nouveau service.
  - **Créer la classe** *AnnonceEventDispatcher* et le fichier correspondant, implémentant l'interface *EventSubscriberInterface* afin de vous abonner à l'événement « KernelEvents::REQUEST ».
  - **Injecter le service** « current_user » dans votre classe *AnnonceEventDispatcher*.
  - Faites en sorte d'**intercepter l'événement** KernelEvents::REQUEST, puis d'afficher le message « *Event* for USERNAME» où USERNAME est le nom de l'utilisateur courant.

Event Dispatcher

- **Injecter le service** « current_route_match » dans votre service a*nnonce.event_subscriber*.
- **Adapter** les arguments de la méthode *__construct()* de votre classe *AnnonceEventDispatcher.*
- **Afficher le message** « Entité annonce » si l'on est bien sur une page d'annonce
- **Vérifier** bien que le message n'apparait pas pour les autres types d'entité ou sur d'autres pages.

Ajouter une table en base

Historique de vues

- On souhaite enregistrer pour chaque utilisateur les annonces qu'il a consultées.
- Il faut donc stocker ces informations en base dans **notre propre table**.
- Pour créer notre table *annonce_history*, on la déclare dans le fichier *annonce.install* (à créer à la racine du module) en utilisant la **Schema API**.
- De quels champs avons-nous besoin ?

Exemple de schema

```php
<?php

/**
 * @file
 * Install, update and uninstall functions for the Ban module.
 */

/**
 * Implements hook_schema().
 */
function ban_schema() {
  $schema['ban_ip'] = array(
    'description' => 'Stores banned IP addresses.',
    'fields' => array(
      'iid' => array(
        'description' => 'Primary Key: unique ID for IP addresses.',
        'type' => 'serial',
        'unsigned' => TRUE,
        'not null' => TRUE,
      ),
      'ip' => array(
        'description' => 'IP address',
        'type' => 'varchar_ascii',
        'length' => 40,
        'not null' => TRUE,
        'default' => '',
      ),
    ),
    'indexes' => array(
      'ip' => array('ip'),
    ),
    'primary key' => array('iid'),
  );
  return $schema;
}
```

Créer une table en base

- **Désinstaller** votre module ***Annonce***.
- **Créer le fichier** */annonce.install*.
- **Implémenter** le *hook_schema()* pour déclarer la structure de la table *annonce_history*.
- **Installer** votre module ***Annonce***.
- **Vérifier** avec phpMyAdmin (ou équivalent) que la table est bien créée.

Enregistrement en base

Modifier la méthode appelée lorsqu'une requête est interceptée, afin d'enregistrer en base nos données :

- **Modifier le service** *annonce.event_subscriber* afin d'y injecter le service « database ».
- **Adapter** les arguments de la méthode *__construct()*.
- En fonction du chemin courant, **enregistrer** ou non les données **en base** (utiliser la méthode *insert()* de l'objet $this->database).
- **Vérifier** avec **phpmyadmin** (ou équivalent) que votre table *annonce_history* est bien alimentée.

Listing avec Views

Views

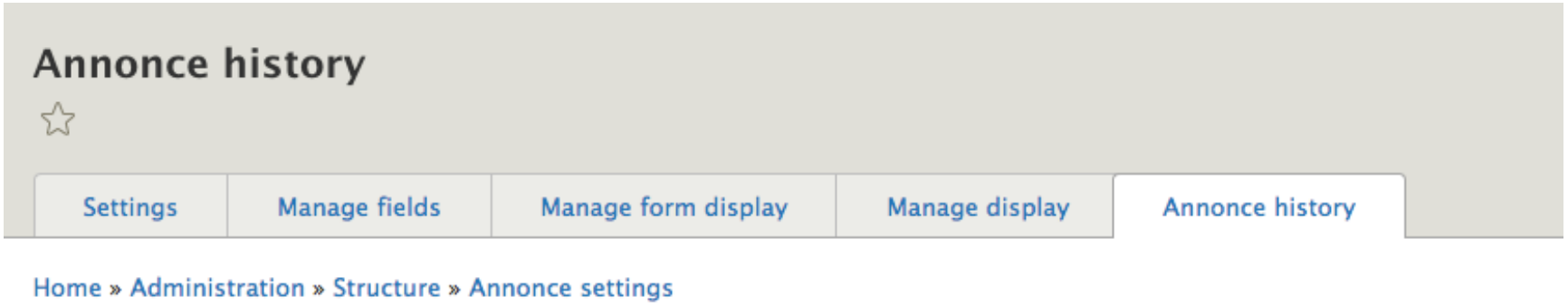
- Le module ***Views*** permet de lister tout (ou presque) de ce qui se trouve en base de données.
- A partir du moment où l'on déclare au système une nouvelle entité, ***Views*** sait lister les contenus correspondants.
- **La table de base est celle déclarée par le type d'entité**. Par exemple, la table de base des noeuds est *node*.
- Ainsi ***Views*** est capable nativement de lister nos annonces.
- Par contre ***Views*** ne connait pas la table *annonce_history*. Il est donc nécessaire de la déclarer, soit en utilisant le *hook_views_data()* (depuis le fichier ***/annonce.module***), soit avec le fichier dédié ***/src/Entity/AnnonceViewsData.php***.

Déclarer sa propre table au module Views

- **Déclarer** au module ***Views*** votre table *annonce_history* en base en modifiant le fichier */src/Entity/AnnonceViewsData.php*.
- ***Remarque** : la syntaxe est la même que celle utilisée pour le hook_views_data(). Vous trouverez un exemple sur api.drupal.org (pour Drupal 8!).*

Créer une vue

- Ajouter la vue ***Annonce history***, permettant de lister l'historique d'affichage des annonces. Vous afficherez le nom de l'annonce, ainsi que son auteur et la date d'affichage.
- Cette liste doit apparaitre dans le back-office sous forme d'**onglet** dans la section ***Admin > Structure > Annonce settings*** (voir la capture ci-dessous).
- Vous ajouterez des filtres sur l'annonce et son auteur.



Exporter une vue

- **Exporter la vue** *Annonce history* sous forme de fichier YAML via *Admin > Configuration > Développement > Synchronisation de configuration*, puis sur l'onglet *Single Import/Export*.
- **Placer ce dernier** dans le répertoire */config/optional*.
- Supprimer cette même vue via le back-office de Drupal.
- Dés-installer votre module, puis ré-installer le. La vue est-elle bien disponible ?

Template

Template

- Chaque type d'entité peut avoir un ou plusieurs fichier de template.
- Il suffit de **déclarer** un **hook de thème** correspondant simplement au nom de l'entité. Par exemple le module **Block** déclare le hook de thème *block* associé au fichier de template *block.html.twig* utilisé pour afficher n'importe quel bloc du système.
- De plus, on dispose d'une fonction de preprocess pour chaque template permettant de manipuler l'objet correspondant avant de la passer au fichier de template. Par exemple si l'on déclare le hook de thème *test* et son template *test.html.twig*, alors on dispose de la fonction *template_test_preprocess()*. Cette fonction peut être étendue par d'autres modules ou bien le thème.

TWIG Debug

- Pour activer les outils de debug de ***TWIG***, ouvrir le fichier ***/sites/default/services.yml***.
- Dans ce fichier trouver les variables ***debug***, ***auto_reload*** et ***cache*** et adapter leurs valeurs.
- **Vider les caches** sur *Admin > Configuration > Développement > Performance*.

```
  # @default false
  debug: true
  # Twig auto-reload:
  #
  # Automatically recompile Twig templates whenever the source code changes.
  # If you don't provide a value for auto_reload, it will be determined
  # based on the value of debug.
  #
  # Not recommended in production environments
  # @default null
  auto_reload: null
  # Twig cache:
  #
  # By default, Twig templates will be compiled and stored in the filesystem
  # to increase performance. Disabling the Twig cache will recompile the
  # templates from source each time they are used. In most cases the
  # auto_reload setting above should be enabled rather than disabling the
  # Twig cache.
  #
  # Not recommended in production environments
  # @default true
  cache: true
factory.keyvalue:
```

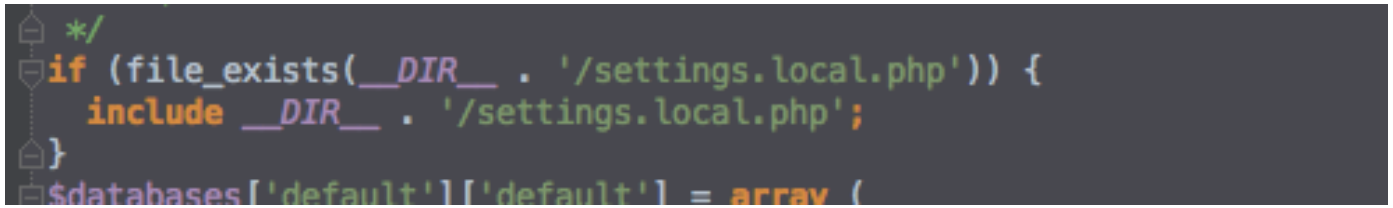
TWIG Debug

- L'option de debug ajoute dans les fichiers de template des **commentaires** (emplacement du fichier de template, suggestion de nom de fichier…).

```
▼<div class="views-row">
    <!-- THEME DEBUG -->
    <!-- THEME HOOK: 'node' -->
    <!-- FILE NAME SUGGESTIONS:
       * node--view--frontpage--page-1.html.twig
       * node--view--frontpage.html.twig
       * node--1--teaser.html.twig
       * node--1.html.twig
       * node--article--teaser.html.twig
       * node--article.html.twig
       * node--teaser.html.twig
       x node.html.twig
    -->
    <!-- BEGIN OUTPUT from 'themes/ive/templates/node.html.twig' -->
  ▼<article data-history-node-id="1" data-quickedit-entity-id="node/1" role="article
```

Cache

- Certains templates (par exemple *node.html.twig*) sont mis en cache même si l'option est désactivée au niveau de ***TWIG***.
- Pour désactiver le cache correspondant copier le fichier ***/sites/example.settings.local.php*** dans */sites/default/* et renommer le ***settings.local.php***.
- Dans le fichier ***/sites/default/settings.php*** décommenter les lignes correspondantes à la capture ci-dessous.
- **Vider les caches** sur *Admin > Configuration > Performance*.



Fichier de template

- Dans le fichier /***annonce.module***, ajouter le *hook_theme()* et **déclarer le hook de thème** *annonce* avec le fichier de template ***/templates/annonce.html.twig***.
- Faites-en sorte que le fichier ***/annonce.page.inc*** soit associé au template ***/templates/annonce.html.twig*** afin que la fonction de preprocess qu'il contient soit bien prise en compte.
- **Ajouter une suggestion de thème** permettant d'avoir un template dédiée par mode d'affichage (utiliser le hook adéquat dans le fichier ***/annonce.module***).

Plugin Condition API

Condition Plugin

- Pour limiter la visibilité d'un bloc, ***Drupal*** propose un certain nombre de critères utilisant le système de condition : type de contenu, rôle utilisateur, chemin, langue.
- Chaque type de critère implémente la **Condition Plugin API**.
- Par exemple le critère relatif au type de noeud est déclaré par le module ***node*** (fichier */core/module/node/src/Plugin/Condition/NodeType.php*).
- On souhaite créer notre propre critère de visibilité basé sur la date.

Condition Plugin API

- Le manager de *Condition Plugin* utilise les **annotation** comme mécanisme de découverte.
- Chaque plugin est un unique fichier décrivant une classe qui étend la classe ***ConditionPluginBase()***.
- Que faut-il déclarer ?
  - Le formulaire correspondant : méthodes *buildConfigurationForm()* et *submitConfigurationForm()*.
  - La logique d'évaluation des conditions : méthode *evaluate()*.
  - La condition par défaut : méthode *defaultConfiguration()*.

```php
<?php
/** @file ...*/
namespace Drupal\annonce\Plugin\Condition;
use ...
/**
 * Provides a 'Test' condition.
 *
 * @Condition(
 *   id = "annonce_test",
 *   label = @Translation("Annonce test"),
 * )
 */
class AnnonceTest extends ConditionPluginBase implements ContainerFactoryPluginInterface {
  public function __construct(array $configuration, $plugin_id, $plugin_definition) {
    parent::__construct($configuration, $plugin_id, $plugin_definition);
  }
  public static function create(ContainerInterface $container, array $configuration, $plugin_id, $plugin_definition) {
    return new static(
      $configuration,
      $plugin_id,
      $plugin_definition
    );
  }
  public function buildConfigurationForm(array $form, FormStateInterface $form_state) {
    $form['test'] = array(
      '#title' => $this->t('Test'),
      '#type' => 'textfield',
      '#default_value' => $this->configuration['test'],
    );
    return parent::buildConfigurationForm($form, $form_state);
  }
  public function submitConfigurationForm(array &$form, FormStateInterface $form_state) {
    $this->configuration['test'] = $form_state->getValue('test');
    parent::submitConfigurationForm($form, $form_state);
  }
  public function summary() {
    return $this->t('Test is...');
  }
  public function evaluate() {
    return TRUE;
  }
  public function defaultConfiguration() {
    return array('test' => array()) + parent::defaultConfiguration();
  }
}
```

Visibilité des blocs en fonction de la date

- On souhaite ajouter un **plugin de visibilité** des blocs en fonction de la date courante.
- Pour chaque bloc on souhaite pouvoir indiquer une date de début et une date de fin correspondant à la **période d'affichage du bloc** sur le site.
- Vous traiterez les cas suivants :
  - aucune date renseignée
  - date de début uniquement.
  - date de fin uniquement.
  - date de début et date de fin.
- Par défaut, le formulaire comprend une case à cocher permettant d'inverser la condition. Ceci n'est pas nécessaire ici; vous ferez donc en sorte de ne pas l'afficher.

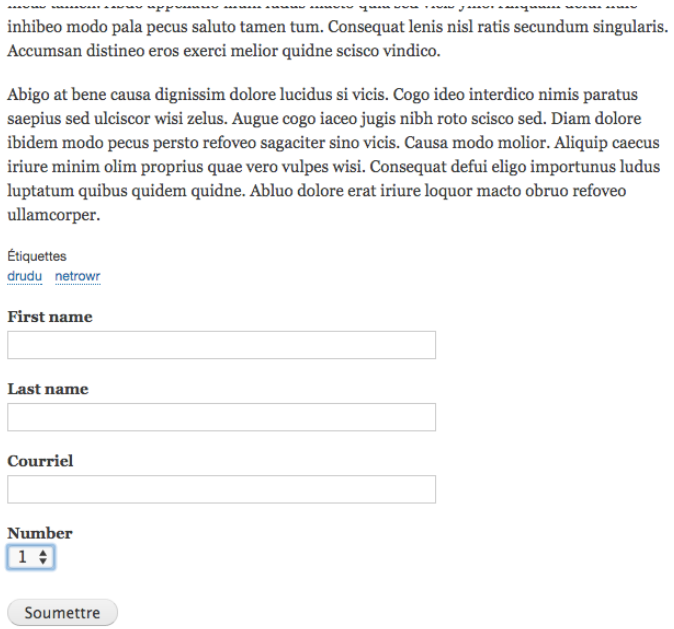
Type de plugin

Type de plugin

- De nombreux modules utilisent le système de plugin soit en implémentant un type de plugin existant soit en déclarant un nouveau type de plugin.
- C'est le cas du module ***Block*** par exemple, qui permet à n'importe quel module de déclarer son propre bloc qui est alors disponible via le back-office de ***Drupal***.
- Nous désirons ici déclarer un tel type de plugin.

Plugin Reusable Form

- On souhaite créer un **nouveau type de plugin** permettant d'ajouter sur chaque noeud d'un certain type un formulaire.
- Chaque type de contenu pourra alors avoir son propre formulaire automatiquement ajouté. Ce dernier peut par exemple permettre à l'utilisateur d'être prévenu par email lorsqu'un noeud est modifié.



Type de plugin

- **Manager de plugin** : permet de découvrir les différentes implémentations du type de plugin. Ce gestionnaire est défini comme service au Container (***/$module.services.yml*** et ***/src/$ModuleManager.yml***).
- **Interface** du type de plugin : chaque implémentation de ce plugin doit respecter cette interface (***/src/$ModulePluginInterface.yml***).
- **Classe de base** : fournit un exemple d'implémentation de l'interface (***/src/$ModulePluginBase.yml***).
- **Annotation** : définit les paramètres nécessaires à déclarer afin de créer un nouveau plugin (***/src/Annotation/$Module.php***).

Plugin Reusable Form

- Notre type de plugin doit permettre d'ajouter des formulaires sur mesure sur n'importe quel type de noeud.
- Il est donc préférable de fournir également un ou plusieurs exemples d'implémentation du plugin.
- C'est ce que l'on fait en ajoutant l'implémentations */src/Plugin/ReusableForm/BasicForm.php*, ainsi que le formulaire correspondant */src/Form/BasicForm.php*.

Plugin Reusable Forms

Afin d'ajouter un formulaire sur les noeuds il est nécessaire de :

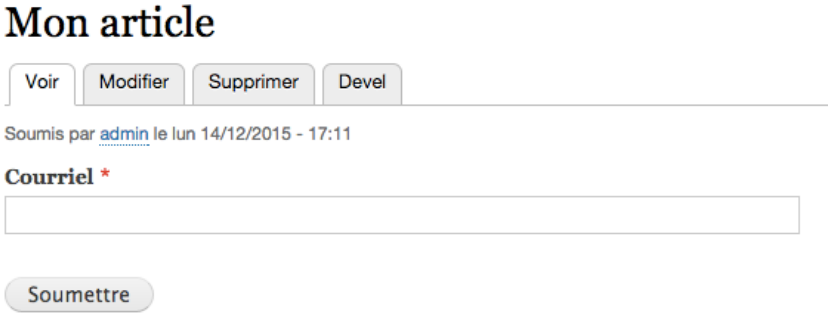
- **Modifier le formulaire de paramètrage** des types de contenu (*hook_form_alter()*).
- **Modifier l'apparence de chaque noeud** en y ajoutant le type de formulaire sélectionné pour le type de contenu en question.
- Permettre d'**ordonner le formulaire** comme s'il s'agissait d'un champ pour chaque type de contenu (*Admin > Structure > Types de contenu > TYPE > Gérer l'affichage*).

Plugin Reusable Forms

```php
/**
 * Implements hook_form_BASE_FORM_ID_alter().
 */
function reusable_forms_form_node_type_form_alter(&$form, FormStateInterface $form_state) {...}

/**
 * Entity form builder for the node type form to map some values to third party
 * settings.
 */
function reusable_forms_form_node_type_form_builder($entity_type, NodeTypeInterface $type, &$fo

/**
 * Implements hook_entity_extra_field_info().
 */
function reusable_forms_entity_extra_field_info() {...}

/**
 * Implements hook_ENTITY_TYPE_view().
 */
function reusable_forms_node_view(array &$build, EntityInterface $entity, EntityViewDisplayInte
```

Ajouter un formulaire sur les noeuds de type article

- Créer le répertoire *email_form* ainsi que le fichier *email_form.info.yml* afin de créer le nouveau module ***Email Form***.
- Activer ce dernier.
- Implémenter le plugin ***Reusable Forms*** sur les noeuds de type article. Vous devez ajouter le fichier ***/src/Plugin/ReusableForm/EmailForm.php*** faisant référence dans ses annotations au formulaire ***/src/Form/EmailForm.php***.
- Ce formulaire sur les articles doit permettre d'enregistrer un email d'utilisateur souhaitant « s'abonner » à un des articles. Créer ainsi une table en base permettant de stocker ces emails. Vous ferez attention à ce qu'un utilisateur ne puissent pas enregistrer plus d'une fois son email.



Envoi d'email - Mail API

- Chaque email correspond à un « template » défini avec la fonction ***HOOK_mail()***.
- Pour envoyer effectivement un email on utilise le service ***plugin.manager.mail***.

```php
/**
 * Implements hook_mail().
 */
function example_mail($key, &$message, $params) {
  switch ($key) {
    case 'exampla_mail':
      $message['from'] = 'example@example.fr';
      $message['subject'] = t('Example subject.');
      $message['body'][] = check_markup(t('Example message body.'));
      break;
  }
}
```

```php
$mailManager = \Drupal::service('plugin.manager.mail');
$mailManager->mail('example', 'example_mail', 'to@example.fr', $langcode, $params, NULL, TRUE);
```

Envoi d'email lorsqu'un article est édité

- Créer le fichier */email_form.module*.
- Implémenter le *hook_mail()* afin de déclarer l'email qui servira à prévenir de l'édition d'un article.
- Utiliser le *HOOK_ENTITY_XXX()* adéquat afin de détecter l'édition d'un noeud. Dans cette fonction faites en sorte que tous les utilisateurs abonnés reçoivent un mail les prévenant de la modification du noeud en question.

Plugin Derivatives

Plugin derivatives

- Classiquement lorsque l'on implémente un plugin on définit une unique instance.
- Parfois on désire créer de multiple instances sans avoir à implémenter autant de fois le plugin.
- C'est le but des **derivatives** (ou *Derivative Discovery Decorator*).
- Par exemple chaque fois que l'on ajoute un menu dans le back-office de Drupal, on crée un bloc automatiquement qui permet d'afficher le menu en question.

Plugin derivatives

```php
<?php

namespace Drupal\annonce\Plugin\Block;

use Drupal\Core\Block\BlockBase;

/**
 * Provides a Custom block.
 *
 * @Block(
 *   id = "multiple_block",
 *   admin_label = @Translation("Multiple Block"),
 *   deriver = "Drupal\annonce\Plugin\Derivative\MyDeriver"
 * )
 */
class MultipleBlock extends BlockBase {

  public function build() {
    return array(
      '#markup' => $this->pluginDefinition['content'],
    );
  }

}
```

```php
<?php

namespace Drupal\annonce\Plugin\Derivative;

use Drupal\Component\Plugin\Derivative\DeriverBase;

class MyDeriver extends DeriverBase {

  /**
   * {@inheritdoc}
   */
  public function getDerivativeDefinitions($base_plugin_definition) {
    $this->derivatives['0'] = $base_plugin_definition;
    $this->derivatives['0']['admin_label'] = t('Block title');
    $this->derivatives['0']['content'] = t('Block content');

    return $this->derivatives;
  }

}
```

Injection de dépendances

- Afin d'injecter des services dans une classe étendant la classe de base *DeriverBase*, on dispose de l'interface ***ContainerDeriverInterface***.
- Cette dernière permet d'utiliser la méthode create() recevant un objet implémentant l'interface ***ContainerInterface*** ainsi que l'identifiant du plugin de base.

```php
<?php

namespace Drupal\my_module\Plugin\Derivative;

use Drupal\Component\Plugin\Derivative\DeriverBase;
use Drupal\Core\Entity\EntityStorageInterface;
use Drupal\Core\Plugin\Discovery\ContainerDeriverInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class MyDeriver extends DeriverBase implements ContainerDeriverInterface {

  protected $service;

  public function __construct(EntityStorageInterface $service) {
    $this->service = $service;
  }

  public static function create(ContainerInterface $container, $base_plugin_id) {
    return new static(
      $container->get('entity_type.manager')->getStorage('entity_type')
    );
  }

  /**
   * {@inheritdoc}
   */
  public function getDerivativeDefinitions($base_plugin_definition) {

    return $this->derivatives;
  }
}
```

Créer un bloc par annonce

- Ajouter un nouveau bloc et son *deriver*.
- Chaque annonce doit générer son propre bloc. Injecter pour se faire le service *Entity Type Manager* et charger toutes les annonces.
- Créer une annonce puis vérifier la disponibilité du bloc correspondant. Comment faire pour automatiquement créer les blocs pour chaque annonce?
- Faites-en sorte d'afficher le titre de l'annonce comme titre du bloc et l'image de l'annonce dans le corps du bloc.
- Les blocs d'annonce ne prennent pas en compte le contrôle d'accès du type d'entité annonce. Comment faire pour s'assurer que seuls les utilisateurs ayant la permission de voir les annonces voit les blocs correspondants?
- Ajouter les informations de cache sur ces blocs.
- Comment faire en sorte que lors d'un ajout ou de la suppression d'une annonce la liste des blocs correspondants soit à jour?

Trained People

Trained People c'est aussi :

- des **formations Drupal spécialisées** Themeur, Développeur, Sécurité & Pe Déploiement & Industrialisation).
- des **certifications** à **Drupal** (Webma Intégrateur, Développeur et Expert).
- un programme de **e-learning Symfo** partenariat avec **SensioLabs**.
- de **l'accompagnement** durant vos projets (audit, régie...)
- des **recommandations** (freelances, agences, hébergement).

TRAINEDPEOPLE

SensioLabs UNIVERSITY

Merci !