



# Formation Drupal 8 Développeur Back

Animée par Romain JARRAUD

# Objectifs de la formation

- Créer **son propre module**.
- Comprendre et savoir utiliser les **objets** des APIs de Drupal.
- Utiliser Drupal comme un **framework** plutôt que comme un CMS.
- Créer des formulaires avec la **Form API**.
- **Adapter** le fonctionnement d'autres modules sans toucher à leur code.
- **Protéger l'accès** à un bloc et une page.

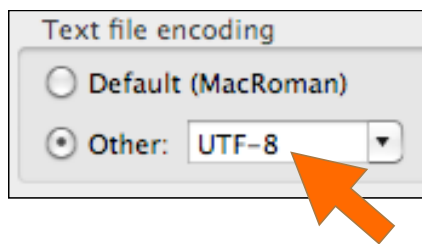
# Que fait mon module ?

- Crée **3 pages sous forme d'onglets** :
  - Accueil avec message
  - Liste des noeuds
  - Calculatrice (formulaire avec AJAX)
- Crée **2 blocs** :
  - Message de bienvenue
  - Décompte du nombre de sessions
- Crée **1 page d'administration** :
  - Modification de la couleur des blocs
- **Ajoute un onglet** à chaque noeud indiquant l'historique des modifications.
- Modifie le **formulaire de contact** du site.




# Développer avec Drupal

# Développement - Rappels



Encodage des  
fichiers en **UTF-8**  
(sans BOM)



```
<?php
function toto() {
    return 'titi';
}
```

Ne pas fermer les  
balises PHP

# API

Fonctions de l'API et hooks.

# API de Drupal

- **Drupal** est écrit en **PHP**, en utilisant les concepts de **Programmation Orientée Objet** :
  - objet
  - classe
  - interface
  - namespace
  - PSR-4
  - ...
- **Drupal fait usage d'objet** très largement, par exemple pour les noeuds ("*nodes*"), les utilisateurs, les blocs...
- Il reste cependant une partie **procédurale** héritée des versions précédentes. Ce sont les fonctions de **hook**.
- La documentation de l'API est disponible sur *api.drupal.org*.
- **Remarque** : *l'API de Drupal a évolué avec ses différentes version (6,7,8.0, 8.4...). Vérifier que vous êtes bien sur la documentation correspondante.*

# Fonctions de base

- Rendre traduisible du texte :

***`t("Message to display")`***

- Le message passé à la fonction *t()* doit être en anglais. Si nécessaire on fournit sa traduction avec l'ensemble des chaînes pour le module.

- Afficher un message d'erreur dans la zone réservée

***`\Drupal::messenger()->addMessage("Mon message", "error")`***

- Vous pouvez déclarer un message de type *status* (défaut), *warning* ou *error*.

- Récupérer l'utilisateur courant (sous forme d'objet) :

***`$account = \Drupal::currentUser()`***.

- Tester si un utilisateur possède la permission "ma permission" :

***`$account->hasPermission("ma permission")`***



# Fonctions de “hook”

- Chaque fois que le système effectue une **tâche précise** (lancement des tâches planifiées Cron, affichage d'un noeud, connexion d'un utilisateur,...), vous pouvez demander à Drupal d'**exécuter votre propre code**.
- On utilise des fonctions particulières de Drupal appelées des “**hooks**”.
- C'est pour vous l'opportunité de faire faire quelque chose en plus à Drupal à un moment donné.
- Les fonctions de hook commencent par “hook\_” (pour la recherche sur l'API).
- Lorsque vous utiliser le hook “*hook\_nomduhook()*” dans votre module “*monmodule*”, votre fonction doit s'appeler “*monmodule\_nomduhook()*”.

# Hook - Exemples

Nom	Exécuté quand...	Argument(s)	Code / Valeur de retour
<b>hook_cron</b>	Drupal s'apprête à lancer les tâches planifiées	NULL	On exécute le code de son choix : envoi d'un e-mail, incrémentation d'un compteur...
<b>hook_requirements</b>	on installe ou met à jour un module	\$phase	On exécute le code de son choix : test de la version du PHP...
<b>hook_form_alter</b>	Avant qu'un formulaire soit affiché	\$form, \$form_state, \$form_id	On modifie la variable \$form passée par référence.

# Hook - utilisation

- Lorsque l'on utilise une fonction de hook, il faut **respecter sa syntaxe et sa signature**. Tenez compte des arguments et de la valeur de retour (si pertinent).
- Pour chaque hook, il y a un exemple sur *api.drupal.org*.

```
/**
 * Implements hook_user_login().
 */
function hello_user_login($account) {
  // Votre code ici.
}
```

# Hooks - Généralités

- Tous les hooks sont facultatifs (il n'y a pas de hooks obligatoires dans un module).
- L'**ordre** dans lequel les hooks sont **déclarés** n'a aucune importance. Ce sont des fonctions PHP, c'est Drupal qui se chargera de les appeler au bon moment.
- Le **même** hook peut être utilisé plusieurs fois dans des modules différents :
  - `titi_form_alter()` = `hook_form_alter()` dans le module **titi**.
  - `toto_form_alter()` = `hook_form_alter()` dans le module **toto**.
- Pour un hook donné, ses différentes implémentations sont exécutées dans l'ordre alphabétique : `titi_form_alter` puis `toto_form_alter`.

# Affichage - Render Array

- Tout le contenu à afficher doit être transmis au thème sous forme de tableaux PHP que l'on appelle des **Render Array**.
- Ces tableaux contiennent le type de formatage HTML à utiliser et les données à formater.
- Il existe de nombreux formatages prêts à l'emploi :
  - tableau
  - liste à puces
  - image
  - ...
- Exemple :

```
];  
$row[] = [  
  'data' => [  
    '#type' => 'operations',  
    '#links' => $links,  
  ],  
];  
$rows[] = $row;  
}  
return [  
  '#type' => 'table',  
  '#header' => $headers,  
  '#rows' => $rows,  
  '#empty' => t('No books available.'),  
];
```

# Module

Les fichiers de base

# 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* : ce fichier est **OBLIGATOIRE**. Il regroupe les métadonnées (nom, description, version...) du module. Ce fichier suffit pour figurer dans la liste des modules dans le back-office de Drupal.
  - *toto.module* : ce fichier contient le code PHP du module (principalement les fonctions de hook).
  - *toto.install* : ce fichier contient le code d'installation, de désinstallation et de mise à jour du module.
- Le nom machine d'un module ne devrait contenir que des **lettres minuscules** et des **underscores**. Pas de chiffres !

# Fichier *.info.yml*

```
# 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 vers d'autres modules.  
dependencies:  
- views:views
```

## ▼ TRAINING



Hello

▼ Hello module for D8 developer training.

Machine name: hello

Version: 8.x-1.0

Requires: Views, Filter, User, System



# 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-dev).
- Les propriétés « description » et « package » sont rendues automatiquement **traduisibles**.

# Créer un module

- **Créer** le module *hello* avec le fichier *hello.info.yml*.
- **Installer** votre module sur *Admin > Extension*.
- Créer le fichier *hello.module* et utilisez le *hook\_help()* pour vous insérez dans la section aide de votre site.
- Implémenter le *hook\_cron()* pour ajouter une tâche planifiée, n'affichant pour le moment qu'un message d'erreur à l'utilisateur.
- Afficher le message "*Bienvenue cher Nom\_utilisateur !*" chaque fois qu'un utilisateur se connecte au site. Utilisez pour cela le hook *hook\_user\_login()*.

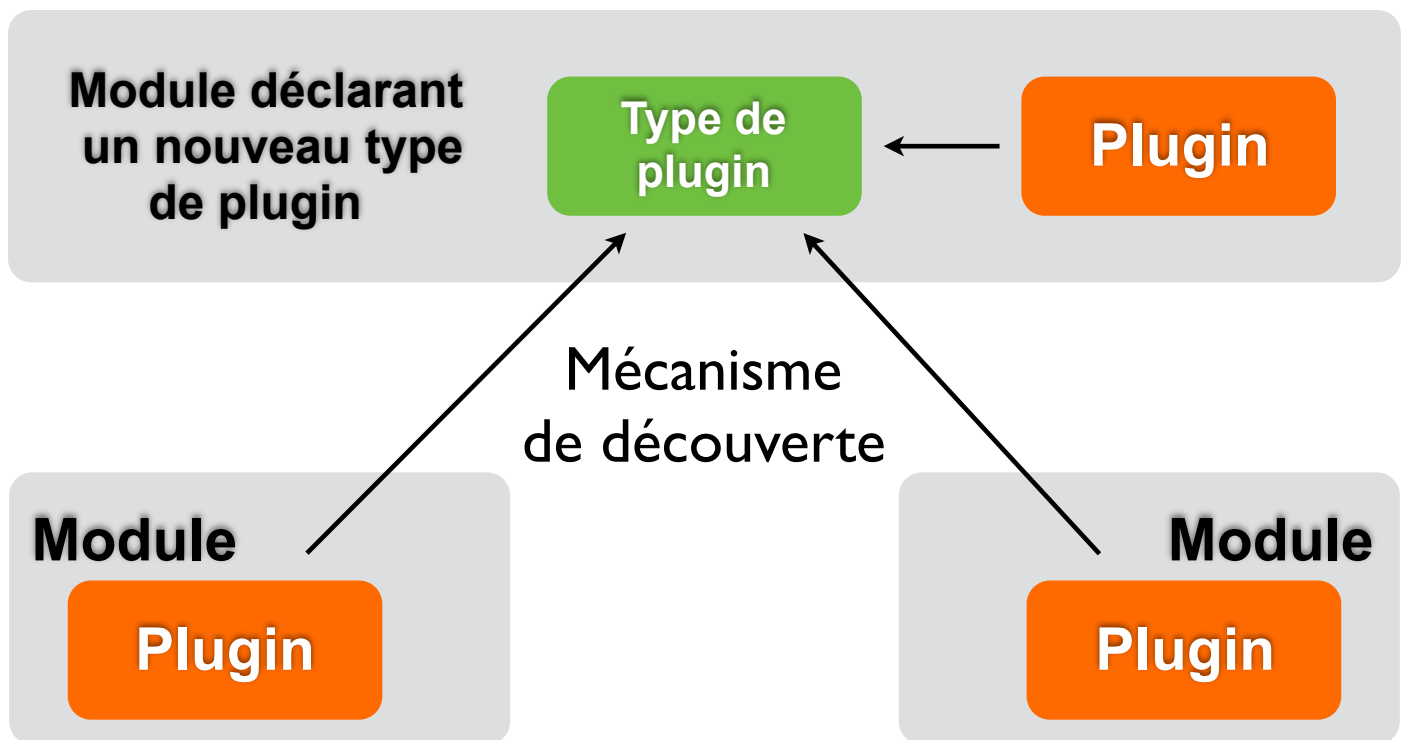


# Les plugins

# Les plugins

- Un **plugin** est un ensemble de code permettant d'obtenir une ou plusieurs fonctionnalités. Ces fonctionnalités peuvent être réutilisées par d'autres modules, en implémentant le type de plugin.
- On a donc différents **types de plugin**, chacun étant associé à un **mécanisme de découverte** et à une **factory** responsable d'instancier les différents objets.
- Un module peut utiliser des plugins de types existants ou bien ajouter son propre type de plugin au système.
- Exemple de plugins :
  - les blocs
  - les types de champ (fichier, texte...)
  - les actions (publier un noeud, bloquer un utilisateur...)
  - les effets sur les images (mise à l'échelle, recadrage...)
  - les types d'entité (noeuds, views...)

# Plugin



# Mécanisme de découverte

## - les annotations PHP

- Les fichiers déclarant des plugins doivent être placés dans le répertoire

*/src/Plugin/Nom\_du\_type\_de\_plugin*

- Ils sont ainsi automatiquement **chargés** et **parsés**.
- Les métadonnées sont indiquées sous forme d'annotations dans les commentaires de la classe déclarant le plugin.
- **Remarque** : *Il ne doit pas y avoir de saut de ligne entre les commentaires contenant les annotations et la classe correspondante.*

# Exemple du plugin *source* de migration

```
<?php

namespace Drupal\action\Plugin\migrate\source;

use ...

/**
 * Drupal action source from database.
 *
 * @MigrateSource(
 *   id = "action",
 *   source_module = "system"
 * )
 */
class Action extends DrupalSqlBase {

  /**
   * {@inheritdoc}
   */
  public function query() {
    return $this->select('actions', 'a')
      ->fields('a');
  }

  /**
   * {@inheritdoc}
   */
  public function fields() {
    $fields = [
      'aid' => $this->t('Action ID'),
      'type' => $this->t('Module'),
      'callback' => $this->t('Callback function'),
      'parameters' => $this->t('Action configuration'),
    ];
    if ($this->getModuleSchemaVersion('system') >= 7000) {
      $fields['label'] = $this->t('Label of the action');
    }
    else {
      $fields['description'] = $this->t('Action description');
    }
    return $fields;
  }

  /**
   * {@inheritdoc}
   */
}
```

# Mécanisme de découverte - les fichiers YAML

- Afin de **découvrir** qu'un module souhaite implémenter un plugin, *Drupal* propose un certain nombre de **fichiers YAML** chargés en fonction de leurs noms.
- Ce mécanisme de découverte est réalisé au tout début du bootstrap de *Drupal*. On l'utilise pour déclarer des éléments de bas niveaux.

*Fichier /toto.links.task.yml*

```
hello.hello:  
  route_name: hello.hello  
  base_route: hello.hello  
  title: 'Hi!'  
  weight: 1
```





# **Injection de dépendance et container**

# Injection de dépendance

- Une classe peut avoir besoin d'instancier un objet d'une autre classe, et dépend donc intrinsèquement de cette dernière. Le code est fortement couplé car un changement sur une classe entraîne de nombreuses adaptations sur toutes les autres qui l'utilise. De plus Il faut être sûr que tous les types d'**objet** sont bien **disponibles**.
- Afin de pouvoir écrire du code facilement maintenable, évolutif et testable, une classe doit **déclarer explicitement ses dépendances** vis à vis d'autres classes.
- Il est donc nécessaire d'avoir un système de gestion de toutes les définitions et dépendances entre les classes. C'est le rôle du container.

# Container

- Un **service** est une **tâche globale** ré-utilisable au sein d'une application.
- Exemples de service :
  - gestionnaire de contrôle d'accès
  - mode de maintenance
  - fil d'Ariane
  - formatage de date
- Le container est une sorte de registre de définition/dépendance entre les différents services disponibles.
- Chaque service est une classe qui déclare les paramètres nécessaires à son fonctionnement. Chaque paramètre est un autre service.

```
router:  
  class: Drupal\Core\Routing\AccessAwareRouter  
  arguments: ['@router.no_access_checks', '@access_manager', '@current_user']
```

# Injection de dépendance avec *Drupal*

- Lorsque l'on a besoin d'un service, on peut toujours utiliser la méthode statique `\Drupal::service('nom_du_service')`.
- Cette méthode est un wrapper et fait appel en réalité au container.
- Son utilisation doit être la plus limitée possible car elle pose principalement 2 problèmes :
  - on fait appel indirectement au container en utilisant la classe *Drupal*, donc le code PHP ne peut plus fonctionner sans cette dernière. Il est préférable de **faire appel directement au container** sans ajouter de dépendance inutile.
  - toute classe PHP qui nécessite un service doit **déclarer explicitement sa dépendance au container**. En passant par la classe *Drupal*, on charge un service à la volée.
- Notons que les fonction de *hook()* ne permettent pas l'injection de dépendance. Cela fait partie des limitations de la programmation procédurale héritée de *Drupal 7*.

# Injection de dépendance avec *Drupal*

- La classe **ControllerBase** (utilisée pour les contrôleurs - cf le routing) définit la méthode *create()*, qui permet d'injecter n'importe quel service du container.
- Toute classe de plugin implémentant l'interface **ContainerFactoryPluginInterface** dispose également de la méthode *create()*.

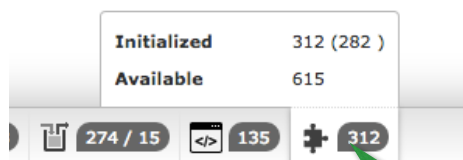
```
/**
 * The block manager.
 *
 * @var \Drupal\Core\Block\BlockManagerInterface
 */
protected $blockManager;

/**
 * Constructs a new CategoryAutocompleteController.
 *
 * @param \Drupal\Core\Block\BlockManagerInterface $block_manager
 *   The block manager.
 */
public function __construct(BlockManagerInterface $block_manager) {
  $this->blockManager = $block_manager;
}

/**
 * {@inheritdoc}
 */
public static function create(ContainerInterface $container) {
  return new static(
    $container->get('plugin.manager.block')
  );
}
```

# Liste des services

- Le module **Web Profiler** permet de connaître la liste des services disponibles.
- Aller sur *Admin > Configuration > Développement > Devel settings*, puis l'onglet *Webprofiler*. Activer l'outil *Services*.



Services			
SERVICES		MIDDLEWARE	
ID	Class	Tags	
ID	Class	Tags	
ID class_loader	Initialized	Yes	
ID kernel	Class <a href="#">S\C\H\KernelInterface</a>	Initialized	Yes
ID service_container	Class <a href="#">S\C\D\ContainerInterface</a>	Initialized	Yes
ID cache_context.ip	Class <a href="#">D\C\C\IpCacheContext</a>	Initialized	No



# Générer l'affichage

# Introduction

- Vous savez maintenant créer un module, mais comment afficher du contenu sur le site ?
- Un module peut générer 2 types d'affichages principaux :
  - **HTML** : page ou bloc.
  - **Non-HTML** : flux RSS, PDF, image...
- Dans cette partie, nous allons voir comment générer les affichages **HTML Page** et **Bloc**.
- ***Remarque** : nous commencerons par voir comment faire pour créer une page ou un bloc, nous aborderons plus tard la question du **contenu** de cette page ou ce bloc.*



# Afficher une page

# Le système de routing

- Chaque site *Drupal* propose un **ensemble de chemins** (exemple: */node* ou */user/12*), chacun renvoyant du contenu à afficher.
- Chaque **chemin** interne à Drupal correspond à un **contrôleur**, responsable de gérer ce qui se passe lorsque ce chemin est appelé.
- Pour déclarer ces chemins on crée le fichier ***MODULE\_NAME.routing.yml*** à la racine du module.
- Chaque **contrôleur** correspond à un fichier contenant une unique **classe**. Ce fichier doit être placé dans le répertoire ***/src/Controller*** du module, par convention.

# Créer une route

*Fichier /hello.routing.yml*

```
hello.hello:
  path: '/hello'
  defaults:
    _title: 'Hello'
    _controller: '\Drupal\hello\Controller\HelloController::content'
  requirements:
    _access: 'TRUE'
```

*Fichier /src/Controller/HelloController.php*

```
<?php

namespace Drupal\hello\Controller;

use Drupal\Core\Controller\ControllerBase;

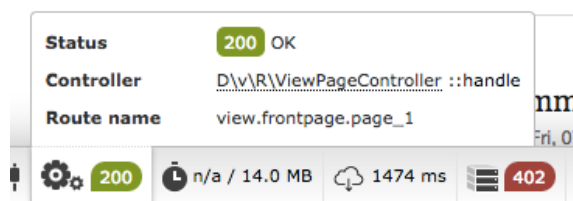
class HelloController extends ControllerBase {

  public function content() {
    return ['#markup' => $this->t('Hello!')];
  }

}
```

# Remarques

- Les noms de routes doivent être préfixé par le nom du module : **MODULE\_NAME.ma\_route**.
- Les chemins déclarés **ne doivent pas** se terminer par un slash ('/').
- La chaine passée à la propriété “\_title” (“defaults”) est **automatiquement** rendue **traduisible**.
- Il est possible d'indiquer d'autres valeurs que contrôleur :
  - **\_form** : formulaire.
  - **\_entity\_view** : affichage d'une entité.
  - **\_entity\_list** : affichage d'une liste d'entités.
  - **\_entity\_form** : formulaire d'une entité (ajout, édition...).
- Pour connaître le nom de la route courante, utiliser le module **Web Profiler** :



# Attention au cache !

- Les définitions de **routes** déclarées dans le fichier **/MODULE\_NAME.routing.yml** sont **mises en cache**; pensez donc à vider le cache après chaque modification de ce fichier.
- Pour vider le cache, cliquer sur le bouton “*Effacer tous les caches*” sur la page *Admin > Configuration > Performance*. Vous pouvez créer un raccourcis vers cette page (module **Shortcut** indispensable) ou bien utiliser le module **Admin Toolbar Extra Tools**.

# Liens de menu

- Lorsque l'on déclare à *Drupal* un chemin associé à un contrôleur, **il n'y a aucun lien de menu par défaut**.
- Il est donc nécessaire de **déclarer explicitement ces liens** pointant vers des chemins internes.
- La déclaration des liens de menu est faite via le fichier ***/MODULE\_NAME.links.menu.yml***.
- Exemple :

```
hello.page:  
  title: 'Hello!'  
  route_name: hello.hello  
  description: 'Hello page'  
  menu_name: main
```

# Afficher une page

- Déclarer le chemin */hello* affichant une page web avec les caractéristiques suivantes :
  - **Titre** "*Hello*"
  - **Contenu** "*Vous êtes sur la page Hello. Votre nom d'utilisateur est NOM.*". Où NOM est le nom de l'utilisateur courant que vous pouvez récupérer via la méthode *\$this->currentUser()*.
  - **Access** "*TRUE*" que tous les utilisateurs (même anonymes) possèdent par défaut.
- **Créer un lien** "*Hello!*" dans la navigation principale (propriété « *menu\_name* » égale à « *main* ») vers cette page.

# Types de lien

- Liens de menu déclarés dans le fichier ***MODULE\_NAME.links.menu.yml***.
- Liens contextuels déclarés dans le fichier ***MODULE\_NAME.links.contextual.yml***.
- Liens de tâches (sous forme d'onglets) déclarés dans le fichier ***MODULE\_NAME.links.task.yml***.
- Liens d'actions (souvent des boutons d'ajout comme pour les noeuds ou les vues) déclarés dans le fichier ***MODULE\_NAME.links.action.yml***.





# Types de lien

- Liens de menu déclarés dans le fichier  
***MODULE\_NAME.links.menu.yml.***
- Liens contextuels déclarés dans le fichier  
***MODULE\_NAME.links.contextual.yml.***
- Liens de tâches (sous forme d'onglets) déclarés dans le fichier  
***MODULE\_NAME.links.task.yml.***
- Liens d'actions (souvent des boutons d'ajout comme pour les noeuds ou les vues) déclarés dans le fichier  
***MODULE\_NAME.links.action.yml.***

```
1 hello.page:  
2   title: 'Hello!'  
3   route_name: hello.hello  
4   description: 'Hello page'  
5   menu_name: main  
6
```

```
1 entity.node.hello:  
2   title: 'Hi!'  
3   group: node  
4   route_name: hello.hello
```

```
1 hello.hello:  
2   route_name: hello.hello  
3   base_route: hello.hello  
4   title: 'Hi!'  
5
```

```
1 hello.action:  
2   route_name: hello.target  
3   title: 'Hi!'  
4   appears_on:  
5     - hello.page
```

# Liens dynamiques

- Il est possible de définir un ensemble de liens **dynamiquement**, par exemple pour obtenir des liens en fonction d'autres données (types d'entité...).
- Pour ce faire il faut utiliser dans le fichier de déclaration (au format YAML) la propriété *derivative* faisant référence à une classe étendant le type de plugin *DeriverBase*.

```
mon_module.mon_lien:  
  derive: 'Drupal\mon_module\Plugin\Derivative\DynamicLink'
```

```
<?php  
  
namespace Drupal\mon_module\Plugin\Derivative;  
  
use Drupal\Component\Plugin\Derivative\DeriverBase;  
  
/**  
 * Defines dynamic links.  
 */  
class DynamicLink extends DeriverBase {  
  
  /**  
   * {@inheritdoc}  
   */  
  public function getDerivativeDefinitions($base_plugin_definition) {  
    $this->derivatives['mon_module.mon_lien'] = $base_plugin_definition;  
    $this->derivatives['mon_module.mon_lien']['title'] = 'My title';  
    $this->derivatives['mon_module.mon_lien']['route_name'] = 'mon_module.ma_route';  
    $this->derivatives['mon_module.mon_lien']['base_route'] = 'mon_module.ma_route.base';  
  
    return $this->derivatives;  
  }  
}
```

# Arguments de page

- Il est possible de récupérer des **arguments** en définissant des **paramètres dynamiques** dans le chemin.
- Ces paramètres sont ensuite **passés au contrôleur** (la méthode indiquée au niveau du routing) et peuvent être ainsi exploités dans le code.

/hello.routing.yml

```
hello.hello:  
  path: '/hello/{param}'  
  defaults:  
    _title: 'Hello'  
    _controller: '\Drupal\hello\Controller\HelloController::content'  
    param: 'no parameter'  
  requirements:  
    _access: 'TRUE'
```

/src/Controller/HelloController.php

```
public function content($param) {  
  return array('#markup' => $param);  
}
```

# Arguments de page

## Remarques :

- Le **nom du paramètre** déclaré dans le fichier de routing doit correspondre à **celui de la variable en argument de la méthode** du contrôleur.
- On peut déclarer jusqu'à 9 paramètres différents.
- Si le nom du paramètre correspond à celui d'une entité, alors c'est **l'objet** complet (par exemple le noeud) qui est **passé à la méthode du contrôleur**.
- La propriété « *defaults: param: ''*; » est obligatoire lorsque l'on crée un lien de menu vers la route correspondante. Cela permet de donner une valeur par défaut s'il n'y a pas de paramètre passé dans l'URL.

# Afficher une page avec arguments

- Reprenez le chemin */hello*.
- Faites en sorte que le contenu devienne dépendant d'un paramètre passé dans l'URL tel que :
  - **Contenu** *"Vous êtes sur la page Hello. Votre nom d'utilisateur est NOM, et voici le paramètre dans l'URL : PARAM."*. Où PARAM est l'argument du chemin */hello/PARAM*.
- **Remarque** : Vous ferez attention à ne pas interpréter le paramètre de l'URL, car c'est une donnée utilisateur dont il faut se méfier !

# Afficher un bloc

# Que peut-on faire sans coder ?

- Sans coder, on peut créer un bloc qui contient :
  - Du **HTML saisi manuellement** : Menu "*Ajouter un bloc*" dans le back-office.
  - Une **vue** : module **Views**.
  - Un **menu** : automatique (tout menu génère un bloc).
  - Un **formulaire** : par exemple avec le module *Webform*.
  - ...

# Alors pourquoi coder ?

- Pour afficher des **données "*hors Drupal*"** dans le bloc :
  - Résultat d'une requête SQL sur une base externe.
  - Données renvoyées par un web service.
- Pour avoir de **meilleures performances**.
  - La génération d'un bloc peut être coûteuse en ressources.
- Pour avoir **plus de flexibilité**.
  - Exemple : afficher des données très spécifiques, comme les 3 dernières pages consultées par l'internaute lors de sa précédente visite (stockées dans un cookie).



# Déclarer un bloc

- Les **blocs** sont des **plugins**, que l'on déclare via un fichier contenant une classe. Ce fichier doit obligatoirement être placé dans le dossier  
*/src/Plugin/Block.*
- Les **métadonnées** de bloc (son identifiant, son titre...) sont indiqués sous forme d'**annotations**.
- Les blocs ainsi générés sont ensuite disponibles sur *Admin > Structure > Mise en page des blocs*, et peuvent être assigné à une région comme n'importe quel autre bloc.
- **IMPORTANT** *N'oubliez pas de vider le cache après avoir créé de nouveaux blocs.*

# Déclarer un bloc

```
<?php

namespace Drupal\hello\Plugin\Block;

use Drupal\Core\Block\BlockBase;

/**
 * Provides a hello block.
 *
 * @Block(
 *   id = "hello_block",
 *   admin_label = @Translation("Hello!")
 * )
 */
class Hello extends BlockBase {

  /**
   * Implements Drupal\Core\Block\BlockBase::build().
   */
  public function build() {
    $build = array('#markup' => $this->t('Welcome!'));

    return $build;
  }
}
```

# Injection de dépendance dans un bloc

- Afin d'utiliser un service dans un bloc, il est préférable de l'injecter dans le constructeur de la classe, plutôt que d'utiliser la méthode statique `\Drupal::service()`.
- Il faut pour ce faire implémenter l'interface `\Drupal\Core\Plugin\ContainerFactoryPluginInterface`.
- Ainsi notre classe bénéficie de la méthode `create()` permettant d'injecter n'importe quels services dans le constructeur.

```
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
    );
}
```

# Afficher un bloc

- Dans votre module ajouter un fichier `.php` dans le répertoire `/src/Plugin/Block` permettant de déclarer un bloc avec les caractéristiques suivantes :
  - **Titre** « *Hello!* ».
  - **Contenu** "*Bienvenue sur notre site. Il est 13h42 14s.*". Vous rendrez l'heure dynamique en utilisant les services `datetime.time` et `date.formatter` du container.
- Faites-en sorte que ce bloc n'apparaisse que sur la page d'accueil de votre site dans la région centrale (« *Contenu* »).

# Front-office ou back-office ?

- Les affichages HTML de vos modules peuvent apparaître aussi bien :
  - sur le **front-office** (ex : bloc contenant les 3 derniers billets de blog publiés).
  - sur le **back-office** (ex : page contenant un formulaire de paramétrage de votre module).
- C'est à **vous** de faire en sorte que la page ou le bloc généré s'affiche dans le front ou le back.
- Il est possible également de déclarer une page dans le back-office, sans que le chemin commence par */admin*, en le précisant comme option dans le fichier */MODULE\_NAME.routing.yml* comme suit :

```
hello.page:  
  path: '/hello-page'  
  defaults:  
    _controller: '\Drupal\hello\Controller\HelloPageController::content'  
  requirements:  
    _access: 'TRUE'  
  options:  
    _admin_route: 'TRUE'
```

# Gestion du cache

# Caches

- De **nombreux niveaux de caches** existent :
  - cache **PHP** (exemples de *OPcache* ou *APC*)
  - cache d'**exécution** (au niveau de la requête)
  - cache **statique** (calculs nécessitant des ressources importantes et que l'on ne veut donc pas refaire systématiquement)
  - cache de **rendu** (HTML)
  - cache **HTTP**
  - ...
- Voyons comment fonctionne les différents caches de *Drupal 8*.

# Cache d'exécution

- Le **cache d'exécution** permet de ne pas refaire un calcul long ou une requête lourde au sein d'une même page (requête).
- Par exemple, si vous affichez un noeud à plusieurs endroits d'une page, il n'est pas nécessaire de faire plusieurs fois la même requête en base pour le récupérer, alors qu'il n'a pas changé.
- Ainsi si vous faites appel plusieurs fois à un objet qui demande des ressources importantes, vous devez le mettre en cache.

```
function on_m_appelle_souvent($id) {  
    $cache = &drupal_static(__FUNCTION__);  
  
    if (empty($cache[$id])) {  
        $cache[$id] = calcul_long($id);  
    }  
    return $cache[$id];  
}
```



# Cache statique

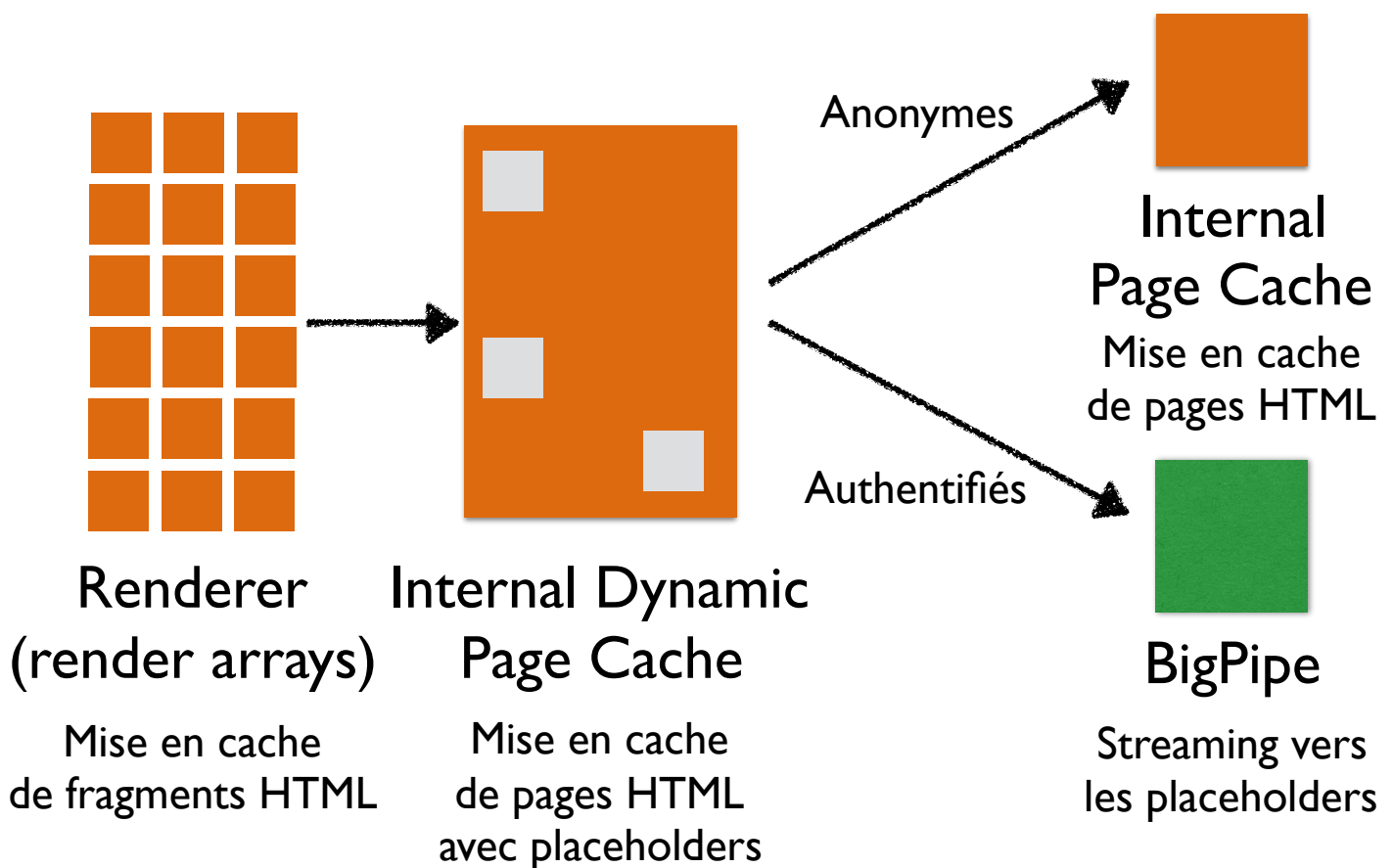
- Le *cache statique* permet de stocker en base, et donc de façon pérenne, une donnée nécessitant beaucoup de traitement pour la calculer.
- On dispose du service *cache* et de ses différentes *bins* (correspondantes à différentes tables en base).

```
function on_m_appelle_souvent($cid) {  
    $cache = \Drupal::cache();  
  
    if ($cache = $cache->get($cid)) {  
        $data = $cache->data;  
    }  
    else {  
        $data = calcul_long($cid);  
        $cache->set($cid, $data, '3600');  
    }  
    return $data;  
}
```

# Les modules de cache

- Pour tous les utilisateurs le module **Dynamic Internal Page Cache** permet de mettre en cache une page incomplète (placeholders). La page peut contenir des parties variables en fonction des utilisateurs. Seules ces sections sont reconstruites. Le cache est contextualisé (contexts).
- Pour les anonymes on dispose du module **Internal Page Cache** : la page complète est mise en cache avec une seule variante invalidée éventuellement si elle expose des données qui ont été modifiées depuis le dernier affichage (tags).
- Afin d'améliorer les performances perçues, on dispose du module **Bigpipe**. En utilisant ce qui précède, la page peut être affichée dans le navigateur alors que toutes ses parties n'ont pas été encore calculées. Elles alimentent des placeholders dans le squelette HTML. Le cache est contextualisé (contexts). La page est mise en cache avec ses placeholders avant leurs substitutions.

# Les modules de cache



# Cache API

- Chaque render array doit définir son propre cache identifié par des cache **keys**.
- On dispose de 3 paramètres pour gérer la validité du cache :
  - le **contexte** : le rendu dépend de paramètres tels que l'utilisateur connecté, ou la langue du site.
  - les **tags** : ce que l'on affiche dépend d'autres entités du système ou de la configuration. Par exemple lorsque l'on affiche le nom de l'auteur du noeud, le rendu de ce dernier doit être invalidé si l'utilisateur en question modifie son nom.
  - la **durée** : arbitraire, on invalide le cache au bout d'un certain temps.
- Par défaut, un bloc est mis en cache de façon permanente (*max-age: Cache::PERMANENT*) et doit donc définir ses **keys**.

# Cache API

- Cache **context** : ex. 'timezone', 'session' (liste des contextes possibles sur [drupal.org/developing/api/8/cache/contexts](http://drupal.org/developing/api/8/cache/contexts)).
- Cache **tags** :
  - entity (ex. : 'node:4', 'node\_list').
  - config (ex. : 'config:filter.format.basic\_html').
  - custom.
- Cache **max-age** : nombre de secondes minimum avant invalidation. Une valeur de 0 indique qu'il n'y a aucune mise en cache.

```
$build_1 = [  
  '#markup' => $markup,  
  '#cache' => [  
    'keys' => ['build_1'],  
    'contexts' => ['user'],  
  ],  
];  
  
$build_2 = [  
  '#markup' => $markup,  
  '#cache' => [  
    'keys' => ['build_2'],  
    'tags' => ['node:4'],  
  ],  
];  
  
$build_3 = [  
  '#markup' => $markup,  
  '#cache' => [  
    'keys' => ['build_3'],  
    'max-age' => '100',  
  ],  
];
```

# Debugger le cache

- Les informations de cache sont visibles dans les entêtes HTTP *X-Drupal-Cache-Tags* et *X-Drupal-Cache-Context*.
- Afin de vérifier le contenu de ces entêtes il est nécessaire de modifier le paramètre *http.response.debug\_cacheability\_headers* du container.
- Pour ce faire, dupliquer le fichier */sites/default/default.services.yml* en */sites/default/services.yml* et adapter comme suit (n'oublier pas de vider le cache pour que le changement soit pris en compte) :

```
#  
# Not recommended in production environments  
# @default false  
http.response.debug_cacheability_headers: true  
factory.keyvalue:  
  {}  
# Default key/value storage service to use.
```

# Cache API

- Modifier le bloc précédent afin que l'heure soit mise à jour automatiquement lors du rafraichissement de la page. Vous utiliserez un cache *max-age* de 10 secondes.
- Ajouter un nouvel utilisateur *toto* (mot de passe « *toto* ») et connecter-le au site sur un autre navigateur. Le bloc *Hello!* est-il mis en cache pour chaque utilisateur ?
- Le bloc est-il mis en cache pour les anonymes ? Tester avec et sans le module ***Internal Page Cache*** activé.

# Ajouter un cache context

- Afin d'ajouter un nouveau contexte de cache, il faut **déclarer un tagged service** ('cache\_context').
- La classe correspondante doit implémenter l'interface **CacheContextInterface** et ses 3 méthodes :
  - **getLabel()** : chaîne définissant le nom du contexte.
  - **getContext()** : renvoie le contexte (URL, noeud courant, utilisateur...).
  - **getCacheableMetadata()** : renvoie un objet de type **CacheableMetadata**.



# Cache API - contexte

- Modifier le contenu du bloc « Hello! » comme suit : "*Bienvenue NOM\_UTILISATEUR. Il est 13h42 14s.*".
- Utiliser un cache *max-age* de 1000, ou bien indiquer comme valeur *Cache::PERMANENT*.
- Editer le nom de toto. Le contenu du bloc est-il changé ?
- Ajouter un cache de contexte sur l'utilisateur. Maintenant le contenu du bloc est-il modifié ?
- En créant un second compte utilisateur, comment est géré le cache ?

# Module *Bigpipe* - Auto-placeholder

- Le module *Bigpipe* permet d'afficher une page sans que tous les fragments HTML aient déjà été calculés. Cela améliore les performances perçues, mais pas le temps de chargement complet de la page.
- Pour pouvoir produire le HTML d'une page incomplète, *Bigpipe* utilise un système de *placeholder*. Le squelette HTML est chargé par le navigateur, mais il y a des « trous » qui ne sont comblés que lorsque le bloc correspondant est calculé.
- Dans certains cas où le contenu d'un bloc varie considérablement, un placeholder est créé automatiquement. Ces conditions sont définies dans le fichier */sites/default/services.yml* comme suit :

```
# For more information about rendering optimizations see
# https://www.drupal.org/developing/api/8/render/arrays/cacheability#optimizing
auto_placeholder_conditions:
  # Max-age at or below which caching is not considered worthwhile.
  #
  # Disable by setting to -1.
  #
  # @default 0
  max-age: 0
  # Cache contexts with a high cardinality.
  #
  # Disable by setting to [].
  #
  # @default ['session', 'user']
  contexts: ['session', 'user']
  # Tags with a high invalidation frequency.
  #
  # Disable by setting to [].
  #
  # @default []
  tags: []
# Cacheability debugging:
```

# Placeholders

- Il est possible de créer des placeholders en dehors des conditions vues précédemment.
- On n'utilise pas un render array, mais un tableau PHP particulier avec les clés *#lazy\_builder* et *#create\_placeholder*.
- Le contenu que l'on souhaite affiché doit être défini dans une méthode à part (celle d'une classe définie comme service). On ne peut passer comme argument à cette méthode que des chaînes, correspondant aux sections « variables » du contenu.

## Fichier /src/Controller/CacheController.php

```
<?php

namespace Drupal\formation\Controller;

use Drupal\Core\Controller\ControllerBase;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Drupal\webprofiler\Config\ConfigFactoryWrapper;
use Drupal\Core\Session\AccountProxy;

class CacheController extends ControllerBase {

    protected $config_factory;

    protected $current_user;

    public function __construct(ConfigFactoryWrapper $config_factory, AccountProxy $current_user) {
        $this->config_factory = $config_factory;
        $this->current_user = $current_user;
    }

    public static function create(ContainerInterface $container) {
        return new static(
            $container->get('config.factory'),
            $container->get('current_user')
        );
    }

    public function content() {
        return [
            '#create_placeholder' => TRUE,
            '#lazy_builder' => [
                'formation.lazy_builders:page',
                [$this->current_user->getAccountName()],
            ],
        ];
    }

}
```

## Fichier /formation.services.yml

```
services:
  formation.lazy_builders:
    class: Drupal\formation\LazyBuildersService
    arguments: []
```

## Fichier /src/LazyBuildersService.php

```
<?php

namespace Drupal\formation;

/**
 * Class LazyBuildersService.
 *
 * @package Drupal\formation
 */
class LazyBuildersService {

  public function page($userName) {
    return [
      '#markup' => 'Your name is ' . $userName,
    ];
  }

}
```

# Afficher du contenu non- HTML

# Afficher des données brutes

- Lorsque l'on appelle un chemin interne, la plupart du temps **Drupal** renvoie du HTML correspondant au contenu de la page.
- Il est possible de renvoyer des données brutes, par exemple :
  - une image
  - un fichier PDF
  - un flux JSON ou XML
- Pour ce faire il faut ajouter une nouvelle route via le fichier de routing (*MODULE\_NAME.routing.yml*) et faire en sorte que le contrôleur renvoie un objet de type *response* de **Symfony**.

# Exemple de route non-HTML

```
class HelloRssController extends ControllerBase {  
    public function content() {  
        // Afin de passer outre le thème et renvoyer des données brute, il faut retourner  
        // un objet de type Symfony\Component\HttpFoundation\Response.  
        $response = new Response();  
        return $response;  
    }  
}
```



# Afficher un service JSON

- Dans le module *hello*, déclarez une nouvelle route et son chemin **hello-json** de sorte qu'il renvoie un **service json**. Cette route ne doit apparaître dans aucun menu.
- Le contenu est laissé à votre libre choix. Le plus simple est d'avoir un flux statique...

# Récapitulons les types d'affichages

- **Afficher une page (HTML) :**
  - **MODULE\_NAME.routing.yml** : déclare le chemin de la page et le contrôleur associé.
  - **contrôleur** : retourne le contenu de la page sous forme de render array.
  - **N.B.** : le contenu retourné est automatiquement "encadré" par le thème courant.
- **Afficher du contenu NON-HTML (RSS, pdf, jpg..)** :
  - **MODULE\_NAME.routing.yml** : déclare le chemin de la page et le contrôleur associé.
  - **contrôleur** : retourne le contenu sous forme de données brutes.
- **Afficher un bloc (HTML) :**
  - **/src/Plugin/Block/monblock.php** : définit les méta-données du bloc (nom, identifiant...) et son contenu.



# La base de données

# Se connecter à la BDD

- La chaîne de connexion à la BDD est stockée dans le fichier */sites/default/settings.php* (variable *\$databases*).
- Pour se connecter à plusieurs bases différentes :
  - **déclarer plusieurs chaînes de connexion** dans settings.php (1) :
    - `$databases['default'] = array(...);`
    - `$databases['archives'] = array(...);`
  - ou bien **dans le code php** :
    - `Database::addConnectionInfo('cle_unique', 'default', $database_info);`
- Puis invoquer la connexion appropriée avec :  
`db_set_active('archives')` ou  
`\Drupal\Core\Database\Database::setActiveConnection('archives')`
- **N.B. :** *Quand vous codez un module, la connexion à la BDD est déjà faite, vous ne devez pas vous connecter manuellement.*

# Manipuler les données via l'API Drupal plutôt qu'avec du SQL maison

- Inconvénients du **SQL maison** :
  - Il faut **comprendre le modèle de données** de Drupal. Un contenu est parfois stocké dans plusieurs tables (ex : les noeuds).
  - **Échappe à la mécanique des hooks**. Par exemple, une insertion dans la table *node* ne déclenche pas le *hook\_entity\_insert()*.
  - Risque d'être **spécifique à un moteur de BDD**.
- **L'API Drupal** permet de contourner ces obstacles. Il faut l'utiliser !

# Principales tables en base

- La base de donnée contient un nombre de tables dépendant du profile d'installation.
- Les principales sont :
  - **block\_content** : liste des blocs du site.
  - **node** et **node\_field\_data** : propriétés système des noeuds.
  - **router** : routes du site.
  - **sessions** : liste des sessions.
  - **users** et **users\_field\_data** : liste de tous les comptes du site.
  - **users\_roles** : rôles utilisateurs.
  - **watchdog** : liste des logs.
  - ...

# Lire des données de la base

# Données de contenu

- **Drupal 8** propose principalement 2 façons de stocker des données en base :
  - utiliser des tables dédiées à gérer soi-même.
  - utiliser le système d'entité de contenu et son système de stockage automatique.
- Le choix dépend de la complexité des données et de l'interaction souhaitée avec les utilisateurs.
- Lorsque l'on a des données « simples », par exemple des chaînes à enregistrer en base, il est possible de prendre en charge leur gestion manuellement. On utilise alors les **Schema API** et **Database API**.
- Au contraire si ces données nécessitent des formulaires pour leur création/édition, un affichage particulier, une extensibilité structurelle (ajout de champs), un contrôle d'accès particulier, etc... alors il est préférable de choisir le système d'entité de contenu. On utilise **Entity API**.



# Manipuler la base de données avec *Database API*

- L'API de requêtes en base est inspirée du **PDO** (*PHP Data Objects*).
- On crée des **requêtes objets** constituées de méthodes.
- Seules les données qui ne sont pas des entités (node, user, views...) doivent être récupérées par cette méthode. Les entités bénéficient d'une API dédiées (voir la suite...).

```
$database = \Drupal::database();  
  
$session_num = $database->select('watchdog', 'w')  
    ->fields('w', array('message'))  
    ->execute();
```

# Faire une requête

- Créer un deuxième bloc avec les paramètres suivants :
  - **Titre** « Sessions actives »
  - **Contenu** « Il y a actuellement N sessions actives », N étant le nombre de sessions en base.
- Ce bloc doit apparaitre dans la barre latérale de droite.
- **Remarque** : vous pouvez utiliser les méthodes « `countQuery()` » et « `fetchField()` ».

# Entity API

- Avec **Drupal 8** tout ou presque est structuré sous forme d'**entité** : les noeuds, les utilisateurs, les blocs, les styles d'image...
- On a ainsi 2 familles distinctes d'entité, suivant la nature des données :
  - *Entité de contenu* : node, user, taxonomy term, comment, File...
  - *Entité de configuration* : views, rôles, style d'image, breakpoint, format de date...
- On dispose de classes de base dédiées :
  - Drupal\Core\Entity\ConfigEntityBase***
  - Drupal\Core\Entity\EntityBase***

# Entity API

- Dès que l'on a besoin de manipuler des entités, il faut utiliser le service *entity\_type.manager*.
- Chaque type d'entité à son propre système de stockage. Il faut l'utiliser, et ne pas faire de chargement/modification à la main.

```
// Manipuler le type d'entité TYPE.  
$storage = \Drupal::entityTypeManager()->getStorage('TYPE');  
  
// Faire des requêtes pour le type d'entité TYPE.  
$query = \Drupal::entityTypeManager()->getStorage('TYPE')->getQuery();  
  
// Requête avec condition.  
$query->condition('field', 'value');  
  
// Récupérer les identifiants des objets correspondants.  
$ids = $query->execute();  
  
// Récupérer les objets correspondants.  
$entities = $storage->loadMultiple($ids);
```

# Affichage sous forme d'onglet

- Les onglets (tâches) de page sont déclarés dans le fichier  
*/MODULE\_NAME.links.task.yml.*
- Ces onglets n'apparaissent qu'à partir du moment où il y a au minimum **deux onglets déclarés**.
- L'onglet par défaut correspond à l'élément dont la propriété "*route\_name*" est identique à la propriété "*base\_route*".

/hello.links.task.yml

```
hello.hello:  
  title: Hello!  
  route_name: hello.hello  
  base_route: hello.hello  
  weight: 1  
  
hello.hello2:  
  title: Hello 2  
  route_name: hello.hello2  
  base_route: hello.hello  
  weight: 2
```

# Créer une route dynamique

- Créez un onglet au chemin **hello/liste-noeuds** qui liste tous les noeuds du site et accepte en argument le type de noeud à lister. Par exemple :
  - `monsite.com/hello/liste-noeuds/page` affichera tous les noeuds de type **page**.
  - `monsite.com/hello/liste-noeuds/article` affichera tous les noeuds de type **article**, et ainsi de suite.
- Sur cet onglet, vous afficherez :
  - uniquement les titres de noeuds,
  - chaque titre devra être cliquable et mener sur la page du noeud correspondant.
  - la liste de titres sera formatée en liste à puces.
- Ajouter les informations adéquates de cache sur cette liste de noeuds.

## Training

Welcome Node list

You are on the Formation page. Your name is *admin*

## Training

Welcome Node list

- Abbas Velit
- Abigo Secundum Voco
- Ad Fere Importunus Iriure
- Amet Probo Venio
- Appellatio Inhibeo Plaga Quia
- Aptent Odio
- At Conventio Praesent Proprius
- At Pala Sagaciter
- Bene Cogo Pala
- Bene Interdico Olim

1 2 3 4 5 suivant » dernier »

# Enregistrer des données

Créer et mettre à jour  
ses propres tables  
avec la Schema API

# Fichier *.install*

Le fichier */MODULE\_NAME.install* contient le code d'**installation**, de **désinstallation** et de **mise à jour** du module, avec les hooks suivants :

- **hook\_schema()** : permet de **définir la ou les tables de BDD** nécessaires à un module, en utilisant une syntaxe spéciale appelée **Schema API**.
- **hook\_install()** : code à exécuter à l'**installation** d'un module
- **hook\_uninstall()** : code à exécuter à la **désinstallation** d'un module (par exemple le nettoyage de tables en base de données).
- **hook\_update\_N()** : code à exécuter à la **mise à jour** d'un module (par exemple l'ajout d'un champ dans une table en base de données).



# Schema API

## Créer ses propres tables

```
<?php
/** @file ... */

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

# Créer une table en base

- Ajoutez le fichier *hello.install* permettant de créer automatiquement une table **hello\_node\_history** lorsque votre module est installé. (Préfixer le nom des tables avec le nom du module est une bonne pratique).
- Cette table comporte les champs suivants :
  - hid : SERIAL - not null - clé primaire
  - nid : INT(10) - not null - UNSIGNED
  - update\_time : INT(11) - not null
- Vérifiez que la table *hello\_node\_history* est bien automatiquement supprimée quand le module **hello** est désinstallé.

# Mettre à jour ses tables

- Le *hook\_update\_N()* contient les mises à jour d'un module ; il doit être situé dans un fichier **MODULE\_NAME.install**.
- Ce hook diffère légèrement des hooks standard :
  - Il faut **remplacer le N par une séquence numérique de 4 chiffres**, le premier chiffre correspondant à la version de Drupal (7000, 8001...).
  - Si un module contient plusieurs mises à jour, toutes doivent être conservées dans le fichier .install en créant **un hook\_update\_N() par mise à jour**, la valeur de N étant incrémentée à chaque fois :
    - hook\_update\_8100() pour la version 8.x-1.0
    - hook\_update\_8101() pour la version 8.x-1.01
    - ...
- Pour **modifier le schéma de base de données** lors d'une mise à jour :
  - **Modifier le schéma original** dans le *hook\_schema()*.
  - Utiliser les méthodes de l'objet schema, documentées sur [api.drupal.org](http://api.drupal.org).

```
use Drupal\Core\Database\Database;  
  
$schema = Database::getConnection()->schema();  
$schema->addField('hello_table', 'field_name', $field_spec);
```

# Mettre à jour ses tables

- A l'**installation d'un module Drupal** appelle les fonctions *hook\_schema()* et *hook\_install()*, puis enregistre le numéro le plus élevé des fonctions *hook\_update\_N()*, par exemple 8005. **Drupal** n'appelle pas les fonction *hook\_update\_N()*.
- Lors de la **mise à jour d'un module** (nouveaux fichiers), **Drupal** appelle les fonctions *hook\_update\_N()*, dont le N est supérieur strictement au N du dernier *hook\_update\_N()* qu'il a parcouru (par exemple 8005).
- Toutes les fonctions *hook\_update\_N()* doivent être présentes dans le fichier ***MODULE\_NAME.install***.

# Créer une mise à jour

- Dans le fichier *hello.install* créé précédemment, utilisez le hook *hook\_update\_N()* pour déclarer une mise à jour qui ajoute le champ “*uid*” (int de 10) à la table *hello\_node\_history*.
- Mettez à jour la structure de la table *hello\_node\_history* dans le *hook\_schema()*.
- Allez sur *Admin > Rapports > Tableau de bord d'administration*, vous devez avoir un message signalant une mise à jour nécessaire en base.

# Les noeuds

Manipuler les noeuds  
programmatically

# L'objet `$node`

- Dans le code PHP, un noeud est représenté par un **objet PHP** `$node` (ou `$entity`).
- L'objet `$node` possède des **champs** (propriétés) :
  - Champs **système** (voir slide suivant).
  - Champ **spécifiques** à votre site, c'est à dire ceux que l'on ajoute via l'interface du back-office sur les différents types de contenu.

# Champs système des noeuds

- Champs de base (table *node*) :
  - **nid** : ID du noeud.
  - **vid** : ID d'une révision du noeud (si révisions activées).
  - **type** : Type du noeud (type interne).
  - **uuid** : universal unique ID.
  - **langcode** : Code de la langue (fr, en...).
- Champ de base (table *node\_field\_data*) :
  - **title** : Titre du nœud.
  - **uid** : ID de l'utilisateur ayant créé le noeud.
  - **status** : Statut du nœud (1 = publié ; 0 = pas publié).
  - **created** : Date de création (timestamp).
  - **changed** : Date de modification (timestamp).
  - **promote** : Nœud *Promu en page d'accueil*.
  - **sticky** : Nœud *Epinglé en haut des listes*.
  - **revision\_translation\_affected** : Code de la langue par défaut.
  - **default\_langcode** : Code de la langue par défaut.



# Node API

```
// Récupérer le stockage des noeuds :  
$storage = \Drupal::entityTypeManager()->getStorage('node');  
  
// Création d'un noeud :  
$node = $storage->create(['type' => 'article', 'title' => 'Mon article']);  
  
// Chargement :  
$node = $storage->load('27');  
$nodes = $storage->loadMultiple(['27', '42']);  
  
// Mise à jour / enregistrement :  
$node->save();  
  
// Suppression :  
$node->delete();
```

# Réagir aux manipulations des noeuds

- Les *hook\_entity\_X()* sont une famille de hooks qui permettent de **réagir aux actions effectuées sur chaque entité** (et donc sur les noeuds) du site :
  - **hook\_node\_delete()** : appelé quand un noeud est supprimé.
  - **hook\_node\_load()** : appelé quand un noeud est chargé.
  - **hook\_node\_view()** : appelé quand un noeud est affiché.
  - **hook\_node\_create()** : appelé quand un noeud est créé.
  - ...
- Ces hooks reçoivent tous en **argument** l'objet **\$entity** concerné par l'action.
- En implémentant l'un de ces hooks, notre module peut **exécuter son propre code** lorsque d'autres modules (ou un utilisateur) manipulent les noeuds.

# Créer un historique de mise à jour

- On souhaite se servir de la table *hello\_node\_history* pour enregistrer la date et l'auteur chaque fois qu'un noeud est mis à jour.
- Vous utiliserez le *hook\_entity\_XXX()* adéquat pour détecter une modification de noeud.
- Vérifier, en modifiant un article, que la table est bien alimentée.
- Que faire lorsque l'on supprime un noeud ? Utiliser le *hook\_node\_delete()* afin de mettre en place votre solution.

# Créer un historique de mise à jour

- Déclarer une nouvelle route avec comme chemin `/node/{node}/history`.
- Faites-en sorte de récupérer l'objet *node* en argument de la méthode associée à la route.
- Ajouter un onglet vers cette route et vérifier qu'il apparaît automatiquement sur chaque page de noeud.
- Utiliser le service *database*, pour récupérer les données de votre table pour chaque noeud et afficher les comme suit :

## Article update history

Voir

Update history

Modifier

Supprimer

Update author	Update time
romain	lun, 19/10/2015 - 12:28
admin	lun, 19/10/2015 - 10:56



# Les formulaires

Créer des formulaires avec la Form API

# Que peut-on faire sans coder ?

- Créer des **formulaires de noeuds** avec le module **Field UI** (core).
- Créer des **formulaires de contact** avec le module **Contact** (core).
- Créer des **questionnaires** avec le module **Webform** (la destination du formulaire est "privée", les données sont visibles uniquement par le webmaster).
- Créer un **formulaire de sondage** avec le module **Poll**.

# Alors pourquoi coder ?

- Pour créer un **formulaire d'admin** (formulaire d'import, formulaire de paramétrage... ).
- Pour avoir un **formulaire avec une apparence sur mesure**, plutôt que de devoir modifier intensivement un formulaire existant avec **hook\_form\_alter()** ou **CSS**.
- Pour gérer des **validations conditionnelles complexes** (Champ B obligatoire si champ A est coché...).
- Pour gérer des **formulaires complexes** (passage de token d'identification, AJAX...).

# Créer un formulaire

Déclarer et afficher un formulaire



# Form API

Chaque formulaire est une classe PHP (avec un namespace du type `\Drupal\nom_module\Form`). On étend la classe `\Drupal\Core\Form\FormBase` avec des méthodes imposées (implémentation de l'interface `\Drupal\Core\Form\FormInterface`) :

- **`getFormID()`** : renvoie l'identifiant du formulaire.
- **`buildForm(array &$form, FormStateInterface $form_state)`** : renvoie le tableau `$form` représentant le formulaire (structure).
- **`submitForm(array &$form, FormStateInterface $form_state)`** : traite le formulaire.

# Formulaire - déclaration

```
<?php
namespace Drupal\hello\Form;

use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;

/**
 * Implements a hello form.
 */
class HelloForm extends FormBase {

    /**
     * {@inheritdoc}.
     */
    public function getFormID() {
        return 'hello_form';
    }

    /**
     * {@inheritdoc}.
     */
    public function buildForm(array $form, FormStateInterface $form_state) {
        return $form;
    }

    /**
     * {@inheritdoc}
     */
    public function submitForm(array &$form, FormStateInterface $form_state) {
    }
}
```

# Formulaire - construction

- La **structure** (les champs) du formulaire est définie dans la méthode ***buildForm()***. Cette dernière renvoie un tableau PHP *\$form*.
- Chaque **champ** du formulaire correspond à une **clé** du tableau *\$form* et est lui même un tableau PHP associatif.
- Toutes les propriétés du formulaire sont contenues dans le tableau *\$form*.

# Formulaire - champs

- On déclare ici un champ texte et un bouton submit.
- Les **identifiants** des champs sont les **clés** du tableau \$form.

```
/**
 * {@inheritdoc}
 */
public function buildForm(array $form, FormStateInterface $form_state) {
    $queued_number = $this->queue->numberOfItems();

    $form['queued_items'] = array(
        '#type' => 'markup',
        '#markup' => $this->t('%number items waiting to be processed.', array('%number' => $queued_number)),
    );

    $form['items_number'] = array(
        '#type' => 'select',
        '#options' => array(
            '10' => '10',
            '100' => '100',
            '1000' => '1000',
        ),
        '#default_value' => '10',
        '#attributes' => ['class' => ['items-number']]
    );

    $form['create_items'] = array(
        '#type' => 'submit',
        '#value' => $this->t('Add items'),
    );

    return $form;
}
```

# Formulaire - imbrication

- Il est possible d'**imbriquer les champs** du formulaire, c'est à dire de regrouper des champs dans un **fieldset** par exemple. D'autres types de champ sont disponibles comme *vertical\_tabs* ou *container*.
- Notez le **#** qui signale les **attributs** de chaque champ.
- C'est ce **#** qui permet de différencier un attribut d'un sous-champ :

```
$form['ds_fields_error'] = [  
  '#type' => 'fieldset',  
  '#title' => $this->t('Fields error'),  
];  
  
$form['ds_fields_error']['disable'] = [  
  '#type' => 'html_tag',  
  '#tag' => 'p',  
  '#value' => $this->t('In case you get an error after configuring a layout printing a message like "Fatal error:  
];  
  
$form['ds_fields_error']['submit'] = [  
  '#type' => 'submit',  
  '#value' => ($this->state->get('ds.disabled', FALSE) ? $this->t('Enable attaching fields') : $this->t('Disable a  
  '#submit' => [ '::submitFieldAttach'],  
  '#weight' => 1,  
];
```

# Formulaire - types de champ disponibles

- Tous les **types de champs disponibles** (checkbox, radio, textfield...) sont documentés sur **[api.drupal.org](https://api.drupal.org/api/drupal/namespaced/Drupal!Core!Render!Element/8)** ([api.drupal.org/api/drupal/namespaced/Drupal!Core!Render!Element/8](https://api.drupal.org/api/drupal/namespaced/Drupal!Core!Render!Element/8)).
- On peut **créer ses propres types de champs** pour la Form API. Consulter par exemple le module *Datetime* du coeur de **Drupal**.
- Certains modules implémentent les **Field API** et **Form API**, par exemple le module *Datetime*...

# Formulaire - affichage

- Pour afficher un formulaire sur une **page**, il est nécessaire d'indiquer au niveau du **routing** que la classe cible définit un formulaire.

```
hello.form:  
  path: '/hello-form'  
  defaults:  
    _title: 'Hello form'  
    _form: '\Drupal\hello\Form\HelloForm'  
  requirements:  
    _access: 'TRUE'
```

- Pour afficher un formulaire dans un autre contexte, on dispose du service dédié **form\_builder**. Ce service appelle la méthode *formBuild()* de la classe passée en argument :

```
$form = \Drupal::formBuilder()->getForm('\Drupal\exemple\Form\ExempleForm');
```

# Créer un formulaire

Ajouter une page *Calculatrice* sous forme d'onglet, comme vu avec l'onglet **Node list** (l'onglet par défaut est la page **Hello**) contenant le formulaire suivant :

## Calculator

Hi!

Node list

Calculator

First value \*

Enter first value.

Operation

☒ Ajouter

☐ Soustrait

☐ Multiply

☐ Divide

Choose operation for processing.

Second value \*

Enter second value.

Calculate



# Valider un formulaire

- Si la classe du formulaire définit la méthode ***validateForm()***, alors cette dernière est appelée lors de la validation du formulaire.
- Dans cette méthode, pour récupérer les valeurs saisies par l'utilisateur, on dispose de :

```
$field_value = $form_state->getValue('field');
```

- Afin d'indiquer une erreur de validation, on procède comme suit :

```
$form_state->setErrorByName('field', $this->t('Error message.'));
```

# Valider un formulaire

- La méthode de validation *validateForm()* prend 2 arguments :
  - *array &\$form*
  - *FormStateInterface \$form\_state*
- Drupal ne procède au **traitement** du formulaire qu'à partir du moment où **aucune erreur n'est soulevée** lors de la validation.
- Il ne faut donc **jamais traiter** le formulaire dans la méthode *validateForm()*.

# Exemples de fonction de validation

```
/**
 * {@inheritdoc}
 */
public function validateForm(array &$form, FormStateInterface $form_state) {
    $value_1 = $form_state->getValue('value1');
    if (!is_numeric($value_1)) {
        $form_state->setErrorByName('value1', $this->t('Value 1 must be numeric!'));
    }
}
```

# Traiter un formulaire

- Le traitement du formulaire est fait via la méthode **submitForm()**; cette dernière est **obligatoire**.
- La méthode de traitement **submitForm()** prend 2 arguments :
  - **array &\$form**
  - **FormStateInterface \$form\_state**
- Dans cette méthode, on récupère les valeurs saisies par l'utilisateur comme dans la méthode **validateForm()**.
- Afin d'**indiquer une redirection**, on utilise la méthode **setRedirect()** sur l'objet **\$form\_state** comme suit :

```
$form_state->setRedirect('hello.hello');
```

- Pour **réaffirmer le formulaire en tenant compte des valeurs de l'utilisateur**, on dispose de la méthode **setRebuild()** de l'objet **\$form\_state** :

```
$form_state->setRebuild();
```

# Exemples de fonction de traitement

```
/**
 * {@inheritdoc}
 */
public function submitForm(array &$form, FormStateInterface $form_state) {
    $value_1 = $form_state->getValue('value1');
    $form_state->setRedirect('hello.hello');
}
```

# Valider et traiter la calculatrice

- Implémentez les **règles de validation** suivantes sur le formulaire créé précédemment :
  - Les valeur 1 et valeur 2 doivent être numériques.
  - La valeur 2 doit être différente de 0 si l'opération est la division.
- Si le formulaire passe la validation avec succès :
  - **Affichez le résultat du calcul** avec *drupal\_set\_message()*, en restant sur la page du formulaire.
  - Puis créez les variantes suivantes :
    - Affichez le résultat du calcul sur **une page distincte** (à créer) en utilisant une redirection.
    - Affichez le résultat **dans le formulaire** avec un champ de type "markup".

# Formulaire avec AJAX

Utiliser l'AJAX

# Utiliser AJAX en pratique

- Pour utiliser l'AJAX dans un formulaire, il faut indiquer :
  - Le champ qui déclenche l'AJAX.
  - La fonction de callback à appeler.
- Lorsqu'un événement est déclenché la **fonction de callback** AJAX est appelée. Elle doit renvoyer un objet de type *Drupal\Core\Ajax\AjaxResponse*. Ce dernier dispose d'une méthode *addCommand* prenant en paramètre des objets par exemple de type *Drupal\Core\Ajax\CssCommand* ou *Drupal\Core\Ajax\HtmlCommand*.
- Le **code HTML/CSS** du formulaire est alors **modifié** en fonction.



# AJAX dans le code

```
$form['text'] = array(  
    '#type'      => 'textfield',  
    '#title'     => t('Text field'),  
    '#ajax' => array(  
        'callback' => array($this, 'validateTextAjax'),  
        'event' => 'change',  
    ),  
    '#suffix' => '<span class="text-message"></span>',  
);
```

```
public function validateTextAjax(array &$form, FormStateInterface $form_state) {  
    $css = ['border' => '2px solid green'];  
    $message = 'Ajax message: ' . $form_state->getValue('text');  
  
    $response = new AjaxResponse();  
    $response->addCommand(new CssCommand('#edit-text', $css));  
    $response->addCommand(new HtmlCommand('.text-message', $message));  
  
    return $response;  
}
```

Text field

Ajax message: hello

# Ajouter de l'AJAX à un formulaire

- On souhaite valider le formulaire en AJAX, afin que l'utilisateur n'ait pas besoin de soumettre le formulaire pour déclencher sa validation.
- Faites en sorte que les champs numériques soient associés à une fonction de callback. Celle-ci doit procéder à la validation et ajouter un message d'erreur (voir la capture).
- **Remarque** : la validation « classique » doit toujours se faire normalement.

Value not numeric!

First value \*

er

Enter first value.

# Créer des pages d'administration

*ConfigFormBase()*

# Formulaire de configuration

- Un formulaire pour une page de l'administration est **identique** à n'importe quel autre formulaire.
- Par rapport à un formulaire "*classique*" on veut **enregistrer de façon persistante l'état du formulaire**. On utilise la Simple Configuration API pour stocker ces valeurs.
- On étend la classe **ConfigFormBase** à la déclaration du formulaire d'administration. Cette fonction permet d'**ajouter un bouton submit** intitulé "*Enregistrer la configuration*" en bas du formulaire et d'afficher le message « *La configuration a bien été enregistrée* » (entre autres).

# Formulaire de configuration

```
<?php

namespace Drupal\hello\Form;

use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;

/**
 * Implements an admin form.
 */
class AdminForm extends ConfigFormBase {

  /**
   * {@inheritdoc}.
   */
  public function getFormID() {
    return 'admin_form';
  }

  /**
   * {@inheritdoc}
   */
  protected function getEditableConfigNames() {
    return ['hello.config'];
  }

  /**
   * {@inheritdoc}.
   */
  public function buildForm(array $form, FormStateInterface $form_state) {
    $form = [];
    return parent::buildForm($form, $form_state);
  }

  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {
    parent::submitForm($form, $form_state);
  }
}
```

# Config API

- La configuration par défaut est stockée dans le fichier `/config/install/MODULE_NAME.config.yml`. « config » est arbitraire ici.
- Le nom du fichier de configuration correspond au nom de la configuration : `exemple.config.yml` => `exemple.config`.
- On peut avoir autant de fichiers de config que nécessaire.
- Pour charger une valeur :  
`\Drupal::config('MODULE_NAME.config')->get('variable');`
- Pour mettre à jour une valeur :  
`\Drupal::config('MODULE_NAME.config')->set('variable', 'valeur 1');`
- Pour enregistrer une valeur :  
`\Drupal::config('MODULE_NAME.config')->save();`

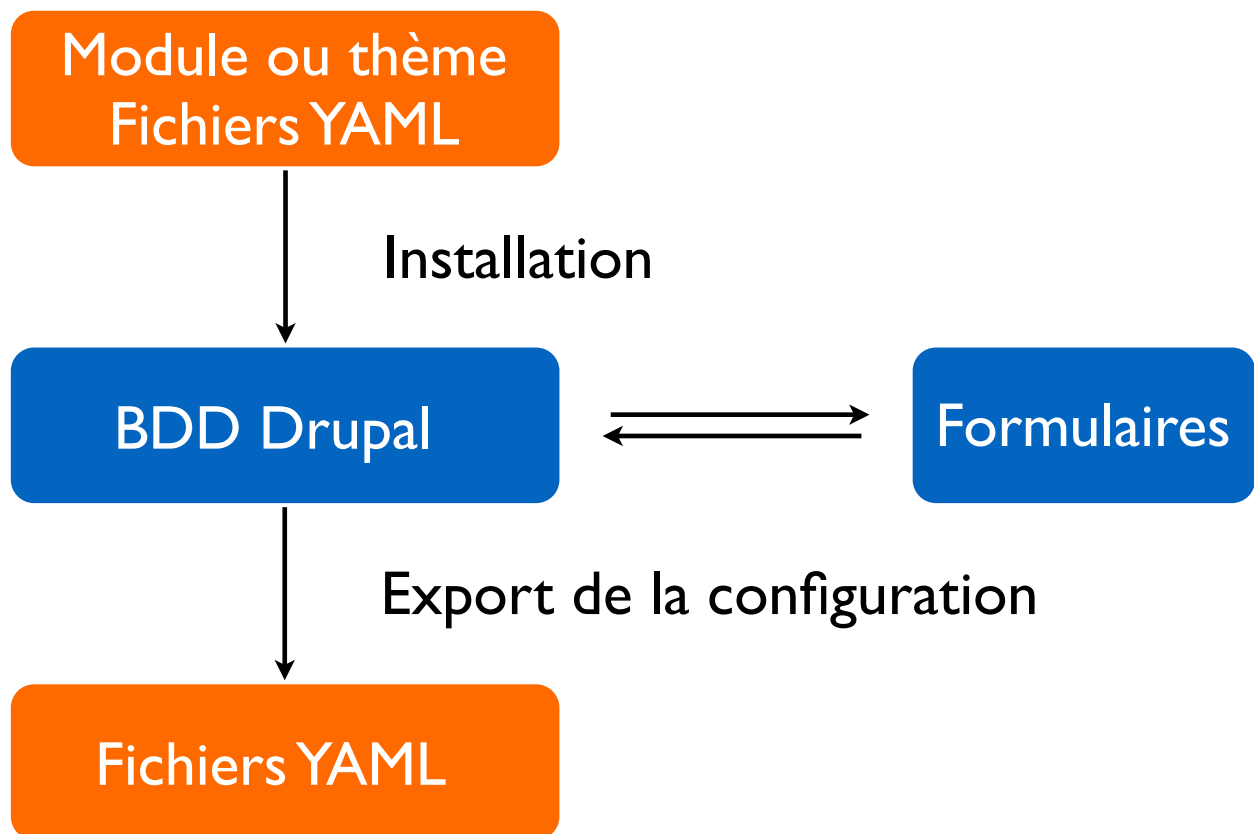
`/config/install/hello.config.yml`

`valeur: 'default'`

# Config API

- La configuration d'un module est chargée lors de son installation. C'est la configuration par défaut.
- Les différents **paramètres** par défaut sont alors **stockés en base**.
- Drupal ne **modifie jamais le contenu des fichiers** du module (il n'en a pas forcément la possibilité de toutes les façons).
- Il est par contre possible alors d'exporter la configuration du site (c'est à dire avec les choix de l'utilisateur) via le back-office. Aller pour ce faire sur **Admin > Configuration > Développement > Configuration synchronization**, onglet **Simple Export/import**.

# Gestion de la configuration





# Config API et ConfigFormBase

- La classe **ConfigFormBase** permet de récupérer directement la configuration :

```
$value = $this->config('config_name')->get('value');
```

- L'enregistrement se fait alors de façon similaire :

```
$this->config('config_name')->set('value', 'valeur')->save();
```

- Egalement on dispose de la méthode *currentUser()* permettant de récupérer l'utilisateur courant :

```
$user = $this->currentUser();
```

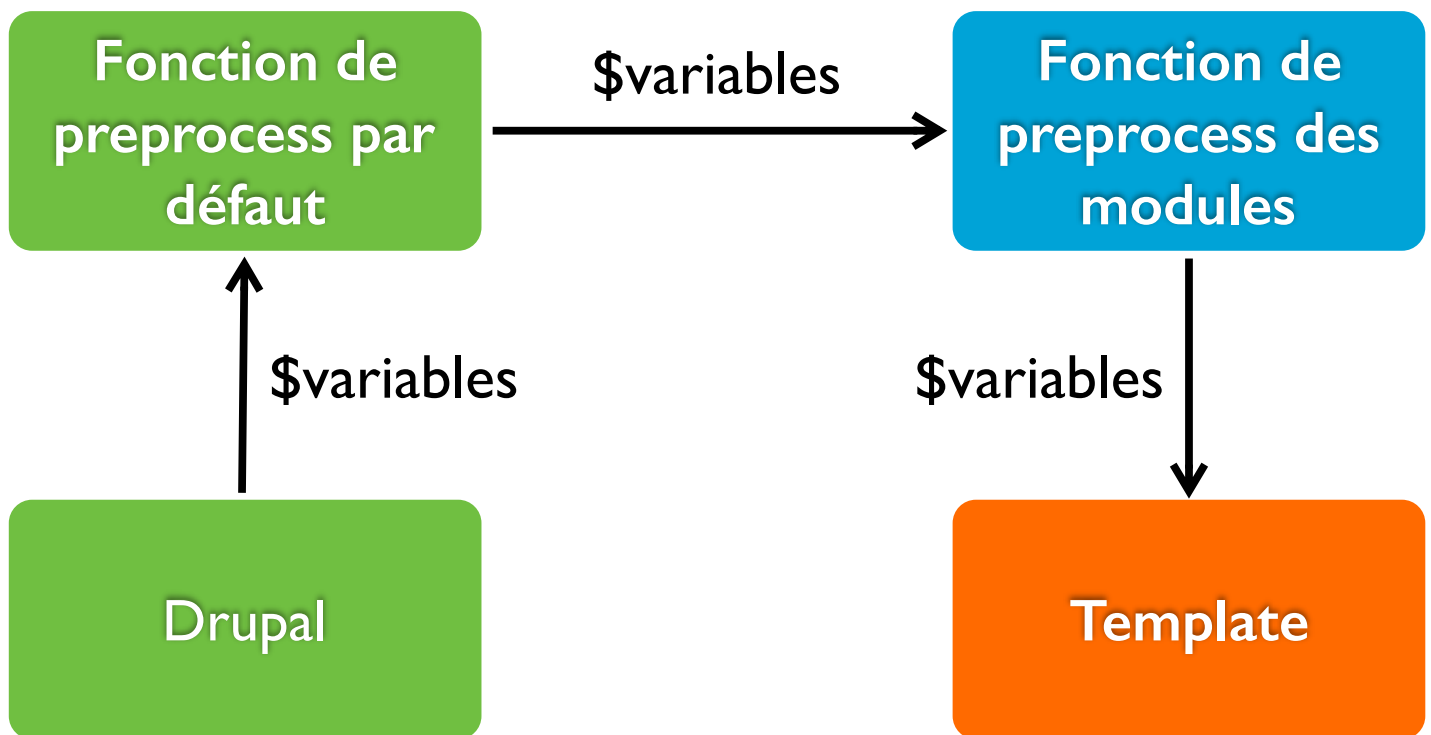
# Créer une page d'admin

- Créer le fichier */config/install/hello.config.yml*, définissant le paramètre 'color' et sa valeur par défaut 'blue-class'.
- Ré-installer le module *Hello*, et vérifier que la configuration est bien reconnue en allant sur *Admin > Configuration > Développement > Synchronisation de configuration > Export > Élément individuel*.
- Créer une page sous forme d'onglet sur *Admin > Apparence > Couleur des blocs*.
- Sur cette page, affichez un formulaire avec :
  - Un champ **liste déroulante** *<select>* avec 3 couleurs (vert, orange, et bleu).
  - Un bouton submit "*Enregistrer la configuration*".
- Faites en sorte que la valeur choisie soit bien pérenne.

# Fonction de preprocess

- Pour **chaque template** on **dispose de variables** que l'on affiche avec le formatage voulu.
- Ces **variables sont créées en amont des templates** dans lesquelles elles sont utilisées.
- On dispose de **fonctions** dites **de preprocess** qui permettent de fabriquer les variables qui sont utilisées dans les templates.
- Il est possible d'**étendre ces fonctions pour modifier les variables** envoyées aux templates **ou en créer de nouvelles**.
- Pour ajouter une fonction de preprocess il faut la nommer comme suit : **MODULE\_preprocess\_HOOK()**. Par exemple *hello\_preprocess\_page()*.

# Fonctions de preprocess



# Ajouter des fichiers CSS

- L'ajout de fichier de CSS se fait via un **système de librairie (bibliothèque)**.
- Il faut tout d'abord déclarer une librairie dans le fichier **MODULE\_NAME.libraries.yml**, puis la référencer depuis le hook **MODULE\_NAME\_page\_attachments()** du fichier **MODULE\_NAME.module**.

*/hello.libraries.yml*

```
1 base:
2   version: VERSION
3   css:
4     theme:
5       css/layout.css: {}
6       css/style.css: {}
7       css/colors.css: {}
8       css/print.css: { media: print }
9
10 maintenance_page:
11   version: VERSION
12   css:
13     theme:
14       css/maintenance-page.css: {}
15   dependencies:
16     - system/maintenance
17
```

*/hello.module*

```
function hello_page_attachments(array &$page) {
  $page['#attached']['library'][] = 'hello/base';
}
```

# Vider le cache

- Chaque entité (noeud, bloc, utilisateur...) est mise en cache individuellement.
- Pour vider le cache correspondant à un type d'entité dans son ensemble on utilise le service *entity\_type.manager* :

```
\Drupal::entityTypeManager()->getViewBuilder('entity_type')->resetCache();
```

# Créer une page d'admin

Faites en sorte que la couleur choisie dans la liste déroulante soit utilisée comme couleur de fond pour tous les blocs de la colonne de gauche. Pour ce faire :

- Grâce à la fonction de *preprocess* adéquate, ajouter une classe dynamiquement sur tous les blocs. Vérifier avec *Firebug* ou équivalent.
- **Déclarer une bibliothèque** contenant la feuille de style *hello.css*.
- **Charger la librairie** depuis le fichier */hello.module*.
- Trouver un sélecteur permettant de modifier la couleur d'arrière plan des blocs de la première barre latérale.
- Dans la méthode *submitForm()* de votre formulaire de configuration, faites-en sorte de vider le cache de rendu des blocs.
- Ajouter la propriété *configure* dans le fichier */hello.info.yml*, afin d'avoir un lien vers le formulaire de configuration sur la page des module.



# State API

Données d'environnement



# State API

- *Drupal* stocke un certain nombres de **données d'environnement**. Ce sont typiquement des données que l'on ne souhaite pas déployer entre des instances différentes d'un même site (par exemple entre DEV, STAGING, PROD).
- On a par exemple :
  - le timestamp du dernier lancement des tâches planifiées (Cron).
  - le timestamp de la dernière recherche de mise à jour (si le module Update Manager est activé).
  - la liste des routes du front office.
  - la liste des fichiers de JS agrégés.
  - L'activation ou non du mode de maintenance.
  - ...
- Il est possible d'accéder à toutes ces données et également d'en créer de nouvelles avec la **State API**.

# State API

- Pour utiliser la *State API*, on dispose du service **state** et de ses différentes méthodes :
  - `$state->get($key)`
  - `$state->getMultiple($keys)`
  - `$state->set($key, $value)`
  - `$state->setMultiple(array($key1 => $value1, $key2 => $value2)`
  - `$state->delete($key)`
  - `$state->deleteMultiple($keys)`
  - `$state->resetCache()` :
- Il est possible de visualiser/modifier l'ensemble des variables avec le module **Devel** et **Admin Toolbar Extra Tools** en allant sur *Admin > Drupal8 > Development > State editor*.

State editor ☆		
<a href="#">Home</a>		
This is a list of state variables and their values. For more information read online documentation of <a href="#">State API in Drupal 8</a> .		
Search		
<input type="text" value="cron"/>		
NAME	VALUE	OPERATIONS
system.cron_key	i3J2JKP2P5UavE3sihLgLVeqPBRWuNy_C9R44d1REd407V8LjIfu74Wh7VMccoW2UMLKH1M6g	<button>Edit</button>
system.cron_last	1472220002	<button>Edit</button>

# Utiliser la *State API*

- On souhaite enregistrer la date et l'heure de la dernière soumission de notre formulaire calculatrice.
- Utiliser le service *state* pour ce faire (à injecter avec la méthode *create()*), en modifiant la méthode *submit()* du formulaire. Vous pouvez récupérer le timestamp correspondant à la requête avec le service *datetime.time*.
- Vérifier via l'interface du site que le timestamp est bien enregistré.
- Comment faire en sorte que cette donnée soit supprimée à la dés-installation du module ***Hello*** ?



# Développement et thème

Templates et render arrays

# Pourquoi se soucier du thème ?

- C'est une bonne pratique de **séparer le code de présentation, du code applicatif** d'un module.
- C'est aux développeurs de faire en sorte que leur module soit **modifiable/adaptable par les themers, sans devoir en changer le code**.
- Il faut donc **utiliser** les hooks de thème de Drupal **ou créer** ses propres **templates** afin que le themer puisse les surcharger.
- Le moteur de thème de Drupal **TWIG**, propose 1 **seul emplacement** pour le code html : des **fichiers de template .html.twig**.

# Invoquer un formatage

- Pour que **Drupal** utilise son mécanisme de surcharge des templates, il faut utiliser des **Render Arrays**.
- Ainsi **Drupal** détermine quel est le candidat à utiliser, en vérifiant si le thème surcharge ou non un des templates. Si aucune surcharge n'est trouvée, alors il utilise le template de base (celui défini par défaut).
- Par exemple ici, **Drupal** appelle le fichier de template *item-list.html.twig*.

```
$items = [];  
foreach ($nodes as $node) {  
    $items[] = $node->toLink();  
}  
$list = [  
    '#theme' => 'item_list',  
    '#items' => $items,  
];
```

# Déclarer un formatage et le template associé

On utilise le *hook\_theme()* pour définir un formatage :

```
function mon_module_theme($existing, $type, $theme, $path) {  
  return array(  
    'mon_module' => array(  
      'template' => 'mon-module',  
      'variables' => array('data' => NULL),  
    ),  
  );  
}
```

/templates/mon-module.html.twig

```
<span>  
  {% trans %}  
    {{ data }}  
  {% endtrans %}  
</span>
```

On invoque le formatage avec un *render array* :

```
$output = array(  
  '#theme' => 'mon_module',  
  '#data' => 'demo',  
);
```

# Créer un template

- On souhaite ajouter le nombre d'update de chaque noeud sur l'onglet *Node history* avec le message suivant : « Le noeud `NODE_NAME` a été modifié X fois. » (voir la capture ci-dessous)
- **Créer** le hook de thème "*hello\_node\_history*" avec le **hook\_theme()**. Vous devez passer les variables "*count*" et "*node*" au template associé (***hello-node-history.html.twig***).
- **Créez** le fichier correspondant dans le répertoire ***/templates*** de votre module affichant le message correspondant au compteur d'update de chaque noeud.
- **Appeler** ce hook de thème dans le code de votre module pour afficher le message de l'onglet "*Update history*" sur chaque page de noeud.

## *Ideo Vereor* update history

Voir	Modifier	Update history	Supprimer
The node <i>Ideo Vereor</i> has been updated 2 time(s).			
Update author	Update time		
admin	ven 20/11		
admin	mar 17/11		





# Contrôle d'accès

# Contrôle d'accès

- Afin de contrôler l'accès aux pages/blocs, il existe différents systèmes :
  - Permissions
  - Rôles
  - Utilisateur connecté ou non
  - ...
- Il est possible d'utiliser d'autres types de contrôle, par exemple en fonction de la date/heure, de la météo... Ces contrôle d'accès sont à créer soi-même.

# Contrôle d'accès - permissions

- La déclaration de ses propres permissions au système est faite via le fichier */MODULE\_NAME.permissions.yml*.
- La liste des permissions se trouve sur *Admin > Personnes > Permissions*.
- La déclaration d'une permission ne sert à rien en elle-même. Il faut tester dans le code si l'**utilisateur courant possède le(s) droit(s)** correspondants.
- Exemple :

/hello.permissions.yml

```
ma permission:
  title: 'Ma permission'
  description: 'Description de ma permission.'
  restrict access: 'true'
```

# Protéger l'accès à un bloc

# Protéger l'accès à un bloc

- Utiliser d'abord les **paramètres de visibilité** du bloc dans le back-office. Ils vous permettent de restreindre l'affichage d'un bloc :
  - À certaines **URLs** du site.
  - À certains **types de contenu**.
  - À certains **rôles utilisateur**.
  - À certaines **langues** (si pertinent).
- Si cela ne convient pas, il faut alors ajouter la méthode ***blockAccess()*** dans la classe définissant le bloc :

```
protected function blockAccess(AccountInterface $account) {  
    return AccessResult::allowed();  
}
```

# Protéger l'accès à un bloc

- **Créer le fichier** */hello.permissions.yml* déclarant la permission “Access *hello*”. Vérifier que votre permission est bien listée sur *Admin > Personnes > Droits*.
- Dans votre module, **utilisez cette permission**, et faites en sorte que seuls les utilisateurs possédant cette permission puissent voir le bloc “*Compteur de sessions*” de votre module.
- Par défaut les anonymes n'ont pas cette permission, donc vous pouvez tester en vous déconnectant de votre site.

# Protéger l'accès à une page

# Protéger l'accès à une route

- Pour sécuriser une route en utilisant le système de permissions natif de *Drupal*, il suffit d'indiquer le ***nom machine de la permission*** requise pour y accéder.
- Ceci se fait donc dans le fichier `/MODULE_NAME.routing.yml`.
- S'il est nécessaire de définir de nouvelles permissions, on procède comme vu précédemment avec le fichier `/MODULE_NAME.permissions.yml`.

```
hello.hello:  
  path: '/hello'  
  defaults:  
    _title: 'Hello'  
    _controller: '\Drupal\hello\Controller\HelloController::content'  
  requirements:  
    _permission: 'ma permission'
```



# Protéger l'accès à une route

- On peut définir le contrôle d'accès en fonction de l'utilisateur soit en testant s'il a une permission, soit s'il appartient à un ou plusieurs rôles. On utilise la propriété **requirements** sur la route correspondante :
  - **\_permission**: 'permission1, permission2' (AND)
  - **\_role**: 'role1+role2' (OR)
  - **\_access**: 'TRUE'
  - **\_user\_is\_logged\_in**: 'TRUE'
- Il est possible également de définir son propre contrôle d'accès en déclarant un nouveau **tagged service** via le fichier **/MODULE\_NAME.services.yml**.

# Protéger l'accès à une page

- Reprenez le fichier */hello.permissions.yml* pour **déclarer une nouvelle permission** appelée “administrer hello”. Vérifier que vous la retrouvez bien dans le back-office (*Admin > Personnes > Permissions*).
- Faites en sorte que :
  - seuls les utilisateurs possédant la permission “*administrer hello*” puissent accéder à la page d’administration (couleur des blocs).
  - seuls les utilisateurs possédant la permission “*access hello*” puissent accéder à la page “*calculatrice*”.

# Contrôle d'accès custom

- Il est possible de définir d'**autres types d'accès** que `_permission` et `_role`.
- Il faut pour cela déclarer au container un nouveau **tagged service** (classe PHP) correspondant.
- C'est ce service qui définit comment se fait le contrôle.
- La méthode `access()` de la classe du service (implémentant `AccessCheckInterface()` doit renvoyer un object `AccessResult`) en utilisant au choix les méthodes statiques suivantes :
  - `AccessResult::neutral()`
  - `AccessResult::allowed()`
  - `AccessResult::forbidden()`
  - `AccessResult::allowedIf()`
  - `AccessResult::forbiddenIf()`
  - `AccessResult::allowedIfHasPermission()`
  - `AccessResult::allowedIfHasPermissions()`

# Contrôle d'accès custom

/example.services.yml

```
services:
  access_check.example:
    class: Drupal\example\Access\ExampleAccessCheck
    tags:
      - { name: access_check, applies_to: _access_example }
```

/example.routing.yml

```
example.access:
  path: '/example-access'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleAccessController::content'
    _title: 'Example access'
  requirements:
    _access_example: 'param'
```

# Contrôle d'accès custom

/src/Access/ExampleAccessCheck.php

```
<?php

namespace Drupal\example\Access;

use Drupal\Core\Access\AccessCheckInterface;
use Drupal\Core\Session\AccountInterface;
use Symfony\Component\Routing\Route;
use Symfony\Component\HttpFoundation\Request;
use Drupal\Core\Access\AccessResult;

class ExampleAccessCheck implements AccessCheckInterface {

  public function applies(Route $route) {
    return NULL;
  }

  public function access(Route $route, Request $request = NULL, AccountInterface $account) {
    $param = $route->getRequirement('_access_example');

    return AccessResult::allowed();
  }
}
```

# Contrôle d'accès et cache

- **Le résultat d'un contrôle d'accès est mis en cache.**  
Ainsi si on évalue l'accès en fonction d'une caractéristique de l'utilisateur, par exemple, il est nécessaire d'ajouter une dépendance vers l'utilisateur.
- De façon générique il est possible d'ajouter une dépendance vers un autre objet.
- L'objet `\Drupal\Core\Access\AccessResult` dispose de méthodes pour ce faire :
  - `addCacheableDependency()`
  - `addCacheContexts()`
  - `cachePerPermission()`
  - `cachePerUser()`
  - ...

# Contrôle d'accès custom

- **Créer le fichier** de services */hello.services.yml* déclarant un nouveau service de contrôle d'accès (« *access\_check.hello* »).
- **Créer le fichier** contenant une classe et sa méthode *access()*, permettant de limiter l'accès en fonction de l'ancienneté de l'utilisateur : l'accès est accordé seulement aux utilisateurs ayant créé leur compte depuis plus de N heures. N est un paramètre dynamique renseigné lors de l'appel à ce service.
- **Modifier la route** *hello.node\_history* (onglet listant les updates d'un noeud) via le fichier */hello.routing.yml*, puis le contrôleur correspondant. L'accès doit être contrôlé par le service précédent : on limite l'accès aux comptes créés il y a plus de 48 heures.



# Modifier l'existant



# Pourquoi modifier l'existant ?

- Souvent, un module contribué fait 80% de ce que vous voulez. Comment prendre la main sur les 20% restants ?
  - Soit vous **repartez de 0** et codez votre propre module.
  - Soit vous **personnalisez le module contrib**, grâce aux hooks d'altération (*hook\_XXX\_alter()*), à la surcharge de classes ou à celle des templates.
- L'avantage de ces techniques est qu'elles modifient le comportement ou l'apparence d'un module sans toucher au code du module lui-même !

# Modifier une route

# Modifier une route existante

- Il est possible de modifier n'importe quelle propriété d'une route définie par un autre module.
- Il faut pour se faire créer un **tagged service event\_subscriber**. La classe correspondante doit étendre la classe *RouteSubscriberBase*.
- On dispose alors dans notre service de la méthode *alterRoutes()* permettant de modifier les définitions de toutes les routes déclarés sur le site.

# Empêcher l'accès à des pages du back-office

- Ajouter un tagged service via le fichier `/hello.services.yml`. La classe correspondante peut être déclarée dans le namespace `Drupal\hello\Routing`.
- Vérifier, grâce au module **Web Profiler**, que votre service est bien reconnu.
- Faites-en sorte alors d'empêcher l'accès à la liste des modules, la page de dés-installation de ces derniers et la page de configuration des permissions.
- Que pensez-vous de cette technique ?

# Modifier un formulaire

# Modifier un formulaire existant

- On peut utiliser 2 techniques différentes pour modifier un formulaire existant :
  - **intercepter le formulaire** avant son affichage avec la fonction *hook\_form\_alter()*.
  - **modifier la route** sur laquelle est affiché le formulaire, afin de remplacer la classe originale par une nouvelle classe étendant la précédente.
- Cette seconde technique n'est pas toujours possible, par exemple si le formulaire est affiché dans un bloc.

# Modifier un formulaire existant avec le *hook\_form\_alter()*

- On utilise le *hook\_form\_alter(array &\$form, \Drupal\Core\Form\FormStateInterface &\$form\_state, \$form\_id)* pour modifier un formulaire existant :
  - **\$form** - Définition du formulaire  
Array PHP identique à un \$form qu'on aurait créé nous-mêmes.  
Cette variable est passée par référence : si on la modifie dans le hook, ces modifications se répercutent à l'extérieur du hook sur le formulaire affiché.
  - **\$form\_state** - État du formulaire  
On peut s'en servir pour récupérer les valeurs saisies dans `$form_state->getValues()`.
  - **\$form\_id** - Identifiant du formulaire  
Rappel : l'identifiant du formulaire est le nom de la fonction qui déclare ses champs.  
Cet identifiant nous permet de tester quel formulaire est en train d'être passé au hook.
- Tous les formulaires de *Drupal* passent par ce hook avant d'être affichés, c'est pour nous l'opportunité de les modifier (ajouter/supprimer/modifier des champs).
- **Remarque** : il existe aussi le *hook\_form\_FORM\_ID\_alter()*, dédié au formulaire portant l'identifiant **FORM\_ID**.

# Validation et traitement dans le *hook\_form\_alter()*

- Tout formulaire Drupal a une fonction de validation et une fonction de traitement par défaut.
- Mais, le *hook\_form\_alter()* modifiant des formulaires existants, il est possible que ces méthodes de validation/traitement par défaut soit **déjà utilisées par le formulaire original**.
- On peut associer des **fonctions supplémentaires** de validation/traitement à un formulaire existant en modifiant les tableaux **`$form['#validate']`** et **`$form['#submit']`** de ce formulaire. Les fonctions/méthodes de ces tableaux sont appelées par ordre d'apparition.

```
// On ajoute une fonction de validation.  
$form['actions']['submit']['#validate'][] = 'ma_fonction_de_validation';  
// On remplace la méthode de traitement par défaut.  
$form['actions']['submit']['#submit'] = array('ma_fonction_de_traitement');
```



# Modifier le formulaire de contact

- Dans le module **hello**, implémentez le **hook\_form\_alter()** de sorte que le formulaire de contact du site (visible au chemin */contact*) subisse les modifications suivantes :
  - Ajoutez un champ “*Téléphone*”.
  - Réordonnez les champs pour leur donner un ordre adapté.
  - Renommez le champ “*Envoyer un message*” en “*Envoyer le message maintenant*”.
  - Supprimer le champ “*Sujet*”.
- Comment feriez-vous pour que le champ “***Téléphone***” que nous avons ajouté soit **inclus dans le message** qui sera envoyé par le formulaire de contact ? Installer le module **Mail Log** afin de visualiser le contenu des mails envoyés par Drupal.
- Quelle remarque pouvez-vous faire sur cette façon de procéder pour modifier les formulaires ?

**Merci !**