



Documentation du framework sabo

Documentation du framework PHP sabo

Table of Contents

Framework Sabo	2
Structure des dossiers	3
Les éléments de configuration	5
Routing	8
Pages par défaut	18
Gestion de la maintenance	19
Controllers	21
Class de traitement	23
Base de données	25
Construction de requêtes customisées	33
Créer un nouveau système	40
Scripts crons	43
La class Application	44
Les class Request et Response	45
Stockage protégé	50
Mailing	52
Accès aux routes	54
Utilitaires	55
Apis	56
Csrf	58
Gestionnaire de fichier	60
Liste	62
La class SessionStorage	63
Server Send Event	65
Chaine de caractères	67
Vérifications	68
Commandes cli	69

Framework Sabo



Sabo

⚠ Sabo est un **framework *PHP*** designé pour ressentir le fait d'écrire du PHP malgré l'utilisation d'un Framework.

Description rapide

- Logique de rendu MVC
- Vues (Blade, Twig ou classique)
- Adapté aux petits et moyens projets

Créateurs

⚠ Le framework a été créé et maintenu par **Yahaya Bathily**, lien du profil Github (<https://github.com/yahvya>)

Ce projet est parti d'un simple test de compétence personnel avant de se développer :)

Structure des dossiers

Structure globale

i Un projet comporte deux parties distinctes, le code du framework ainsi que le code source de l'utilisateur.

- `sabo-core` dossier contenant l'entièreté du framework
- `src` dossier contenant les sources utilisateur
- `.htaccess` fichier servant à la redirection de routing
- `composer.json` + `composer.lock` fichiers du gestionnaire composer

i Gestion de l'auto-loading et des dépendances

- `sabo` utilitaire de ligne de commandes

i ce fichier est un fichier php sans l'extension

Focus sur le dossier src

- `configs` contient les différents fichiers de configuration du framework
- `controllers` contient les différents 'controllers' de l'application

i L'emplacement des 'controllers' dépend de l'auto-loading psr-4

- `models` contient les différents 'models' de l'application

i L'emplacement des 'models' dépend de l'auto-loading psr-4

- `public` contient les ressources publiques de l'application
- `routes` contient les fichiers d'enregistrement des routes de l'application
- `storage` dossier de stockage interne de l'application
- `treatment` contient les class de traitement d'actions
- `views` contient les éléments de (vue, mails)

Les éléments de configuration

⚠ Le dossier **src/configs** contient différents fichiers de configurations, permettant de gérer du framework, aux fonctions globales de l'application.

[functions.php]

i Ce fichier contient l'ensemble des fonctions globales à l'application. Une fonction qui y est définie est accessible à tout endroit du code.

La liste suivante peut changer en fonction des modifications rapides apportées

Liste des fonctions actuellement définies

- **debug** fonction à paramètres multiples, permet de debugger les variables fournies
- **debugDie** fonction à paramètres multiples, permet de debugger les variables fournies puis stoppe l'exécution **die**
- **route** fonction permettant de récupérer le lien d'une route nommée

```
route(requestMethod: "get ou post ou put ...",routeName:
"nom_de_la_route",replaces: ["generic1" => 10]);
```

- **generateCsrf** génère un code csrf et fourni le gestionnaire créé

```
generateCsrf()->getToken();
```

- **checkCsrf** vérifie un token csrf et retourne son état de validité

```
if(!checkCsrf(token: $_POST["token"]))
    throw new \Exception(message: "Token invalide");
```

[framework.php]

i Ce fichier contient les configurations modulables du framework (chemins, formats ...). Il permet également d'ajouter des configurations qui seront chargées globalement dans l'application.

- Définition du timezone par défaut
- Définition du chemin du dossier des ressources publiques
- Définition du chemin du dossier de stockage
- Définition du chemin du dossier des routes
- Définition de la liste des extensions autorisées à l'accès publique
- Définition de la regex permettant la définition du format d'un paramètre générique. (`{generic_url_param}` ou `:generic_url_param` ou `format_personnalisé`)

⚠ Cette regex doit permettre de capturer le nom du paramètre générique tout en reconnaissant les symboles liés

ex: `{generic_url_param}` = `"\{([a-zA-Z]+)\}"` Le nom du paramètre est capturé et les accolades détectées.

[twig-config.php]


i Ce fichier contient la fonction d'enregistrement des extensions twig.

[blade-config.php]

i Ce fichier contient la fonction d'enregistrement des directives blade ainsi que la fonction de récupération de route formatée pour javascript.

Référez-vous aux commentaires des fonctions

[env.php]

 Configuration d'environnement

Routing

Etapes de routing

Format procédure


Première étape

1. Accès à un lien de l'application par l'utilisateur
2. redirection via `.htaccess` sur le point d'entrée `sabo-core/index.php`
3. Lancement de l'application via `Application::launch`
4. Insertion et vérification des configurations dans l'ordre
 - fonctions globales
 - blade + twig
 - environnement
 - framework
5. Initialisation de la base de données si requis
6. Lancement du router

Routing

Gestion de la maintenance

1. Vérification dans l'environnement
2. Si accès déjà autorisé le routing continue
3. Sinon affichage de la page de maintenance ou lien spécial d'accès

 Si la maintenance laisse le routing se poursuivre uniquement

- Si le lien est lié à une ressource accessible alors, rendu de la ressource

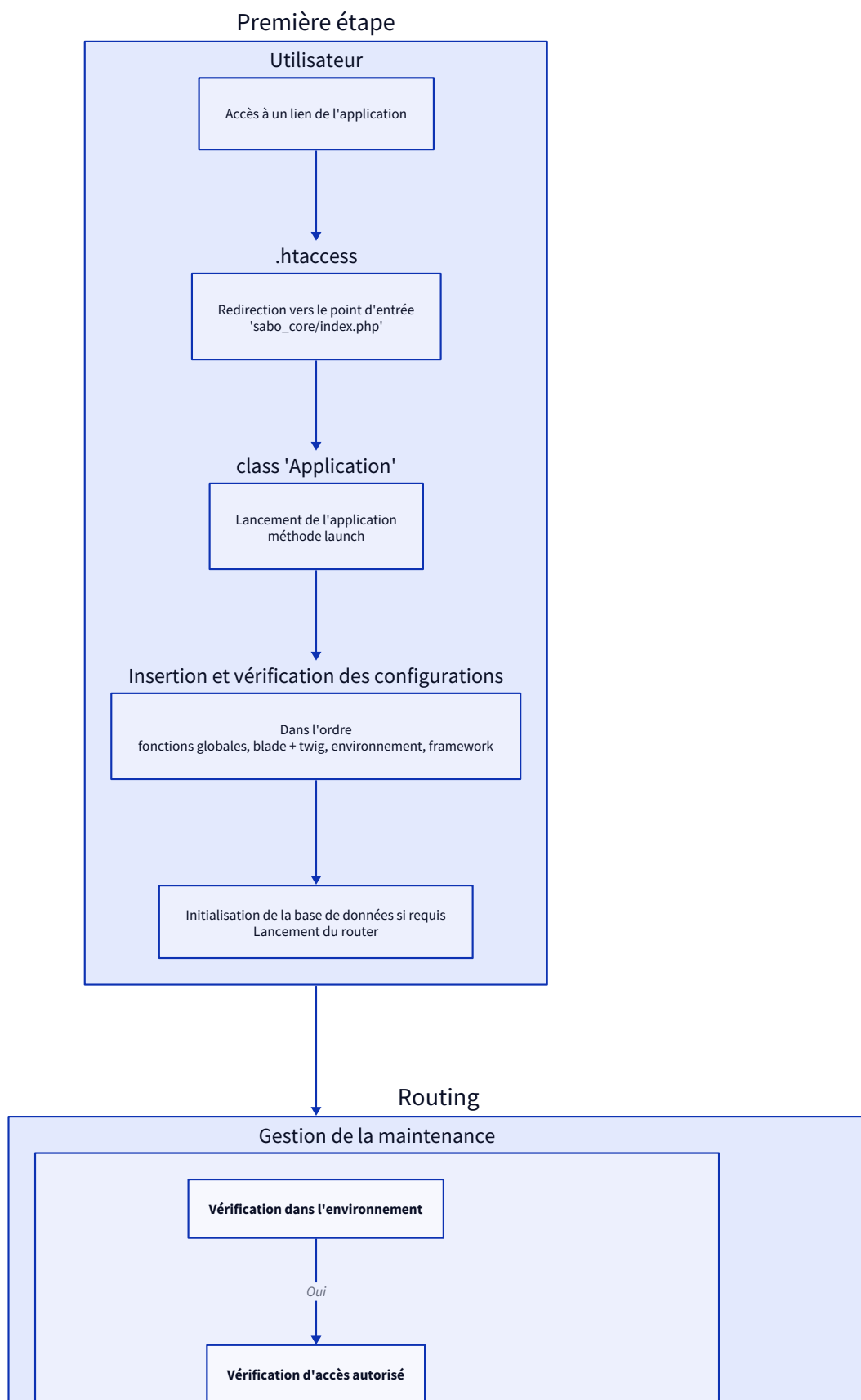
i Noter qu'en état de maintenance même l'accès aux ressources est bloqué si l'accès n'est pas acquis

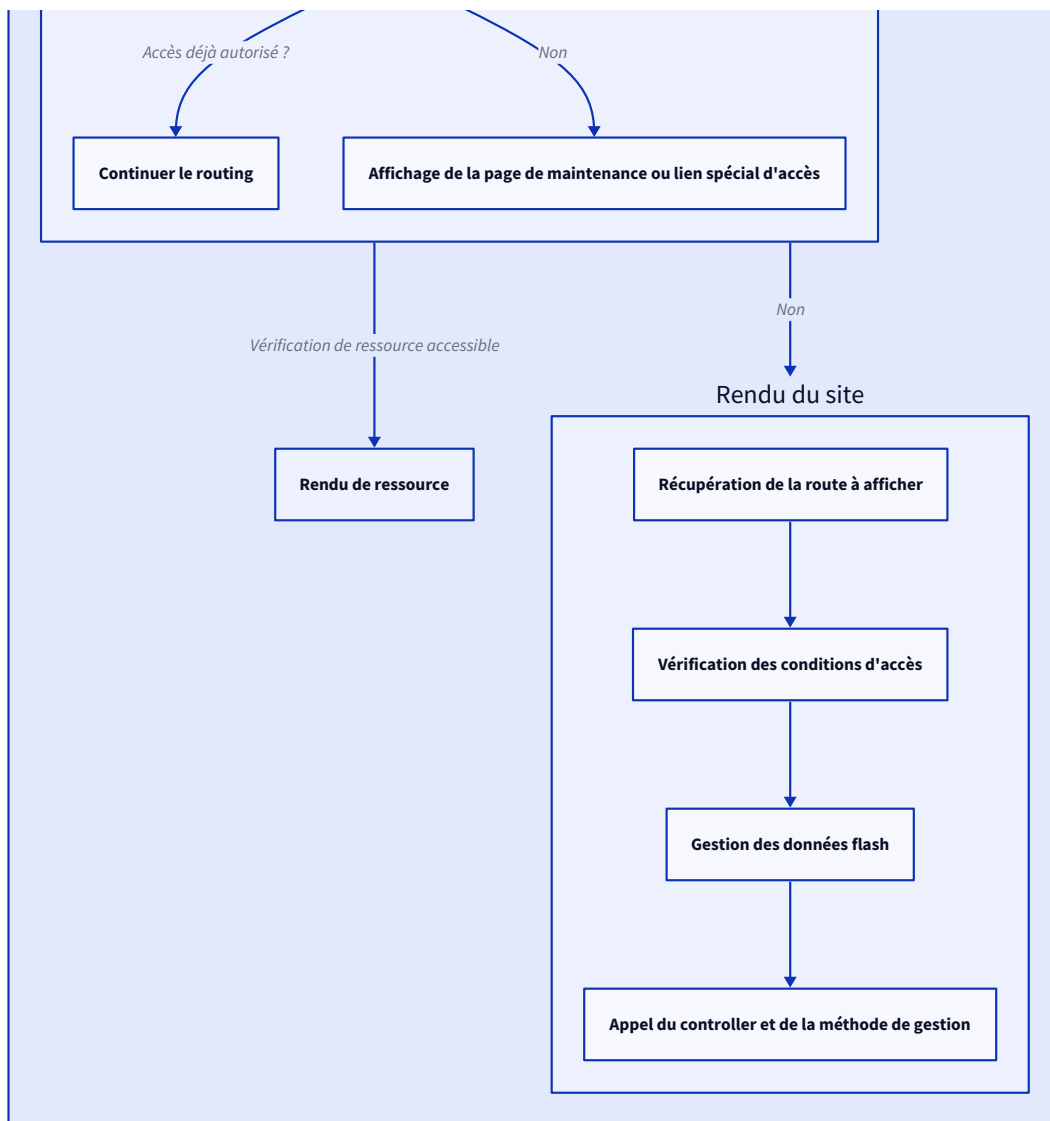
Rendu du site

1. Récupération de la route à afficher
2. Vérification des conditions d'accès
3. Gestion des données flash
4. Appel du controller et de la méthode de gestion

Format schéma







Enregistrement des routes

⚠ L'enregistrement des routes passe par le dossier **src/routes**.

- `routes.php` fichier utilisé par le framework permettant d'inclure les documents d'enregistrement d'api et web

⚠ Il n'est pas recommandé de le modifier bien qu'il puisse servir à

l'enregistrement des routes

- `web.php` sert à l'enregistrement des routes web
- `api.php` sert à l'enregistrement des routes d'api

i La séparation des deux fichiers n'est que sémantique. Il est recommandé pour les routes d'api de débiter par la création d'un groupe `/api`

```
<?php
# routes d'api
use SaboCore\Routing\Routes\RouteManager;
RouteManager::registerGroup(
    linksPrefix: "/api/:apiVersion",
    routes: [],
    genericParamsConfig: ["apiVersion" => "[0-9_]+"],
    groupAccessVerifiers: []
);
# ou
RouteManager::registerGroup(
    linksPrefix: "/api",
    routes: [],
    groupAccessVerifiers: []
);
```

⚠ Il faut différencier les class `RouteManager` et `Route`. La class `Route` permet de créer une route, la class `RouteManager` quant à elle sert à la gestion des routes par le framework, mais également à les enregistrer.

Enregistrement d'une route

```
<?php
// routes web
use SaboCore\Routing\Response\BladeResponse;
```

```

use SaboCore\Routing\Routes\Route;
use SaboCore\Routing\Routes\RouteManager;
RouteManager::registerRoute(
    Route::get(
        link: "/",
        toExecute: fn():BladeResponse => new BladeResponse("sabo",
["websiteLink" => "https://yahvya.github.io/sabo-final-doc/"]),
        routeName: "sabo"
    )
);

```

i Les méthodes statiques de la class `Route` représentent les différentes méthodes de requête couramment utilisées. [get, post, put, delete, patch, options, head, trace]

Enregistrement d'un groupe de routes

```

<?php
# routes d'api
use SaboCore\Routing\Routes\RouteManager;
use SaboCore\Routing\Response\BladeResponse;
use SaboCore\Routing\Routes\Route;
RouteManager::registerGroup(
    linksPrefix: "/api",
    routes: [
        Route::get(
            link: "/",
            toExecute: fn():BladeResponse => new BladeResponse("sabo",
["websiteLink" => "https://yahvya.github.io/sabo-final-doc/"]),
            routeName: "sabo"
        ),
        Route::get(
            link: "/test",
            toExecute: fn():BladeResponse => new BladeResponse("sabo",
["websiteLink" => "https://yahvya.github.io/sabo-final-doc/"]),

```

```

        routeName: "sabo"
    )
],
groupAccessVerifiers: []
);

```

Ce code donnera les routes, accessible en get :

- /api
- /api/test

Paramètres de création de route

```

Route::method(string $link, Closure|array $toExecute, string $routeName,
array $genericParamsRegex = [], array $accessVerifiers = [])

```

- `$link` lien associé. Ce lien peut contenir des paramètres génériques au format défini par framework.php. Voir la définition de format (["\[framework.php\]" in "Les éléments de configuration"](#))
- `$toExecute` Callable à appeler pour traiter la requête

i Le callable doit renvoyer une `Response` et peut se faire injecter un type `Request` ainsi que les paramètres génériques via leur nom

```

<?php
# pour la route /test/:username/test2
function traitementFunction(Request $requestManager,string
$username):JsonResponse{
    return ["username" => $username];
}

```

ou

```

[Controller::class,"methodName"]

```


- `$routeName` nom unique de la route (*format libre*)
- `$genericParamsRegex` configuration des regex associés aux paramètres génériques

```
<?php
# pour la route /test/:articleName/:id
["articleName" => "[a-z\-\0-9]+", "id" => "[0-9]+"]
```

- `$accessVerifiers` Conditions de vérification d'accès à la route

```
<?php
[new Verifier(verifier:
[CustomApiController::class, "checkApiAccess"], onFailure:
[CustomApiController::class, "refuseApiAccess"])]
# ou
[new Verifier(verifier: fn(Request $requestManager):bool =>
false, onFailure: fn():RedirectResponse => ...)]
```

i La classe de vérification permet de définir toute une étape de vérification. La fonction de condition de vérification retournant un booléen, si false la fonction de gestion d'échec qui dans le cas des routes renvoi une `Response`

i Les mêmes éléments sont posés sur la création de groupe de route, les conditions sont appliquées à toutes les sous routes.

! Sur un groupe de routes les règles de paramètres génériques sont encapsulés sur le prefix des liens

Sous fichiers de route

! Pour apporter une meilleure organisation, des sous fichiers routes peuvent être créés via l'utilisation de `RouteManager::fromFile`

Cette fonction prend en paramètre le chemin du fichier sans son extension qui doit être **.php** avec comme racine le dossier **routes**.

- routes
 - web.php
 - routes.php
 - api.php
 - custom-routes
 - authentication.php

Pour charger le fichier **authentication.php**, vous pouvez utiliser la fonction spécifiée ci-dessus dans le fichier **web.php**.

```
RouteManager::fromFile(filename: "custom-routes/authentication");
```

Mode de développement

⚠ Lors de la phase de routing le mode de développement défini si oui ou non la récupération d'exception non gérée est capturée ou non. Si dev mode à **true** alors les erreurs ne sont pas capturées sinon elles le sont et la page d'erreur interne est affichée.

Mise à jour du mode dans l'environnement (["\[env.php\]" in "Les éléments de configuration"](#))

Pages par défaut

⚠ Dans certaines étapes de la phase de routing, certaines erreurs ou événements peuvent se produire, pour les traiter 3 fichiers html par défaut sont définis. Ces fichiers sont des fichiers html statiques sans accès aux ressources du site ni valeurs dynamiques pour garantir leur affichage sans soucis.

Ces fichiers se situent dans le répertoire `src/views/default-pages`

- `internal-error.html` affiché en cas d'erreur interne durant la phase de routing ou d'exception non traitée par le code utilisateur
- `maintenance.html` affiché lorsque l'application est en état de maintenance
- `not-found.html` affiché quand le lien saisi n'est pas trouvé

Gestion de la maintenance

⚠ La mise à jour de l'état de maintenance se gère via la configuration d'environnement (["\[env.php\]" in "Les éléments de configuration"](#))

Etat de l'application durant une maintenance

⚠ Durant une maintenance comme expliqué sur le schéma du processus routing (["Etapas de routing" in "Routing"](#)), toute ressource hormis la page d'affichage de maintenance est bloqué.

Accès à l'application

Dans la configuration d'environnement la ligne `->setConfig(name: MaintenanceConfig::SECRET_LINK->value,value: "/maintenance/dev/access/")` permet de définir un lien secret d'accès pouvant être changé.

Ce lien donne même à l'appel d'un controller permettant la gestion de l'accès à la maintenance

Définition d'un controller de gestion de maintenance

1. Créer une class
2. Extends la class `SaboCore\Controller\MaintenanceController`
3. Implémenter les méthodes requises : `showMaintenancePage` affiche la page de saisie des identifiants - `verifyLogin` vérifie si l'accès est autorisé (notamment par le traitement du formulaire reçu)
4. Enregistrer le gestionnaire dans l'environnement via la ligne `->setConfig(name: MaintenanceConfig::ACCESS_MANAGER->value,value: MyCustomManager::class)`

Le framework arrive avec un processus d'accès par défaut pré-défini via les éléments suivant :

- `src/controllers/DefaultMaintenanceController` ce controller par défaut implémente l'affichage d'une page et la vérification via code secret d'accès
- `src/views/maintenance/authentication.blade.php` vue de saisie de code d'accès
- `src/storage/maintenance/maintenance.secret` fichier contenant un mot de passe haché. Par défaut `motdepasse`

```
echo password_hash(password: "votre_mot_de_passe", algo:
PASSWORD_BCRYPT);
```



Il est important de modifier ce mot de passe avant déploiement

Processus d'accès

1. Se rendre sur le lien secret
2. Le controller par défaut affiche la vue d'authentification
3. À la validation du formulaire la vérification via `verifyLogin` va comparer le mot de passe fourni à celui contenu dans le fichier `maintenance.secret`
4. Si conforme return true, et débloque l'accès via session sinon false

Controllers

⚠ Les controllers sont des class permettant d'effectuer les actions de traitement à l'accès de liens précis.

Ils sont situés par défaut dans le dossier `src/controllers` sous la méthode de chargement `psr-4`.

Définition

Créer une classe dans le dossier des controllers qui **extends** de la class `CustomController`. Cette classe sert d'intermédiaire entre le framework et l'utilisateur.

```
<?php
namespace Controllers;
use SaboCore\Routing\Response\BladeResponse;
use SaboCore\Routing\Response\JsonResponse;
use SaboCore\Routing\Request\Request;
class MyController extends CustomController{
    public function showHomePage():BladeResponse{
        return new BladeResponse(pathFromViews: "sabo",datas:
["websiteLink" => "https://yahvya.github.io/sabo-final-doc/"]);
    }
    public function getDatasOf(Request $requestManager,string
$username):JsonResponse{
        return new JsonResponse(json: ["method" => $requestManager-
>getMethod(),"username" => $username]);
    }
}
```

Utilisation

Les controllers peuvent être associés aux liens dans la définition des routes

```
<?php
use Controllers\MyController;
```

```

use SaboCore\Routing\Routes\Route;
use SaboCore\Routing\Routes\RouteManager;
RouteManager::registerRoute(
    route: Route::get(
        link: "/",
        toExecute: [MyController::class,"showHomePage"]),
    routeName: "app.home"
)
);
RouteManager::registerRoute(
    route: Route::get(
        link: "/datas/:username",
        toExecute: [MyController::class,"getDatasOf"]),
    routeName: "app.get-datas"
)
);

```

Fonctions par défaut

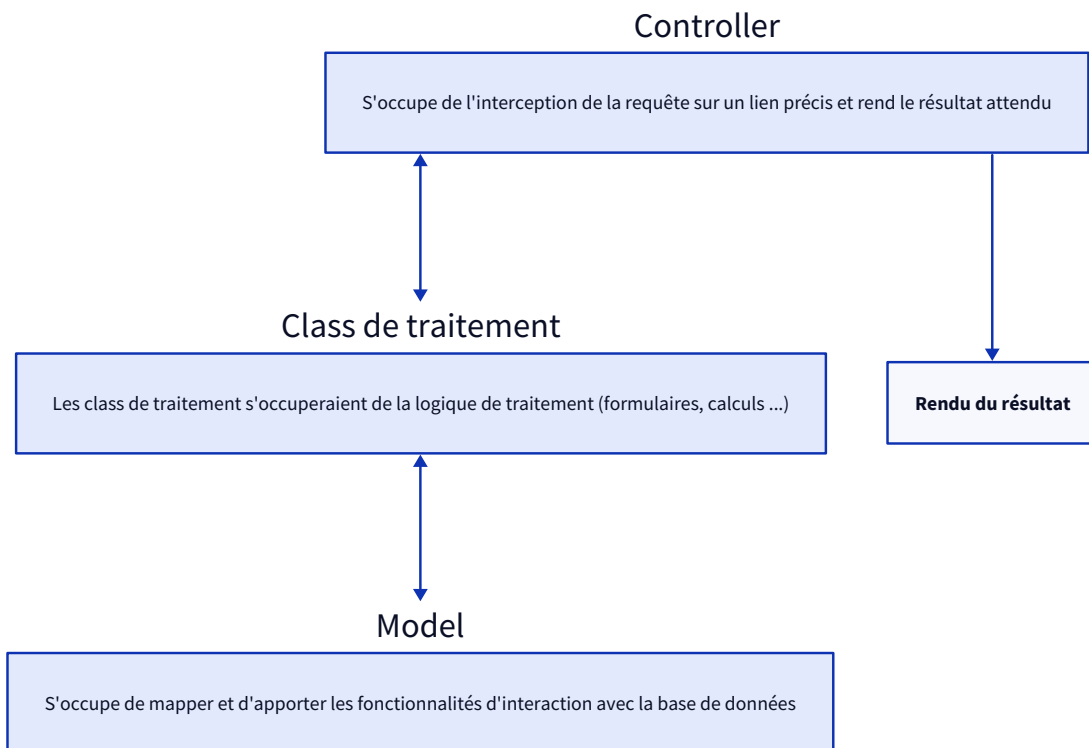
⚠ La class d'héritage des controllers sert actuellement de marqueur et n'embarque pas de fonctions par défaut

Ajouter des fonctions

⚠ La classe intermédiaire `CustomController` permet à l'utilisateur d'apporter des customisations à ses controllers, les méthodes ajoutées sont héritées des controllers.

Class de traitement

⚠ Le principe des class de traitement sert à ne pas surcharger les controllers. La logique de fonctionnement MVC attendue par le framework fonctionne ainsi



Utilisation

Les class de traitement se situent par défaut dans `src/treatment` et peuvent étendre de la classe `CustomTreatment`.

i La logique recommandée est de fonctionner par renvoi d'exceptions pour marquer une erreur

Deux méthodes sont ajoutées par défaut à cet effet, `throwModelException` et `throwException`

Exemple d'implémentation

1.

```
class MyController{
    public function sendContactMessage(Request
$request):RedirectResponse{
        try{
            ContactTreatment::sendMessage(request: $request);
            return new RedirectResponse(link:
"https://github.com/yahvya");
        }
        catch(TreatmentException $e){
            $request
                ->getSessionStorage()
                ->storeFlash(storeKey: "contact.error",toStore: $e-
>getErrorMessage());
            return new RedirectResponse(route(requestMethod:
"get",routeName: "contact.page"));
        }
    }
}
```
2.

```
class ContactTreatment extends CustomTreatment{
    public static function sendMessage(Request $request):void{
        self::throwException(errorMessage: "Erreur de test");
    }
}
```



Vous pouvez ajouter des méthodes customisées dans `CustomTreatment`

Base de données

⚠ Le framework embarque un mini ORM permettant l'interaction avec la base de données. Le système par défaut intégré se base sur MySQL, mais la structure est faite de manière à pouvoir implémenter un système personnalisé.

Configuration de la base de données

La configuration de la base de données se base sur le fichier d'environnement `src/configs/env.php` au niveau de `// configuration de la base de données`.

- Mettre le booléen d'initialisation à `true`, permet de spécifier à l'étape de routing de lancer la gestion de la base de données
- Spécifier le provider, une class qui extends de `SaboCore\Database\Default\Provider\DatabaseProvider` (par défaut `MysqlProvider`) permet d'avoir une class pouvant se charger d'initialiser les outils requis.
- Spécifier les données de connexion

Les models

⚠ Un model dans le contexte de *sabo* est une class permettant de représenter sous forme de code PHP une table / une vue de la base de données. Cette description passe via une liste d'attributs (["Les attributs de description" in "Base de données"](#)).

Les models sont chargés par défaut dans le dossier `src/models` et extends de la class `CustomModel` servant d'intermédiaire avec le framework.

Cette class permet également de définir des méthodes personnalisées pour les models.

Les attributs de description

i Ces attributs servent à décrire les différents éléments de la class, afin de les associer à la base de données et se situent dans le namespace SaboCore\Database\Default\Attributes

- `TableName` attribut associé à la class permettant de spécifier le nom de la table représentée
- `Binary|Bool|Char|Decimal|Enum|Int|Json|Text|TinyInt|VarBinary|Varchar` avec comme suffixe `Column`. Ces types permettent de mapper les différentes colonnes possibles sur un type `Mysql`. *Référez-vous aux paramètres et commentaires de ces fonctions*

```
use SaboCore\Database\Default\Attributes\VarcharColumn;  
#[VarcharColumn(columnName: "user_name",maxLen: 255,)]  
protected string $username;
```

- `TimestampColumn` ce type spécial du framework (bien qu'existant sur MYSQL) sert de passage pour toutes données de temps (datetime, time ...) via timestamp. Le type associé derrière est un `INT`. Un élément portant cet attribut doit utiliser le type customisé `Timestamp`.

```
use SaboCore\Database\Default\Attributes\TimestampColumn;  
use SaboCore\Database\Default\CustomDatatypes\Timestamp;  
use Override;  
#[TimestampColumn(columnName: "joined_at")]  
protected Timestamp $joinedAt;  
#[Override]  
protected function beforeCreate(mixed $datas = []): self{  
    parent::beforeCreate();  
    $this->joinedAt = new Timestamp();  
    return $this;  
}
```

ou

```
use SaboCore\Database\Default\Attributes\TimestampColumn;  
use SaboCore\Database\Default\CustomDatatypes\Timestamp;
```

```
#[TimestampColumn(columnName: "joined_at")]
protected Timestamp $joinedAt = new Timestamp();
```

- **JoinedColumn** ce type spécial permet de représenter une jointure entre deux tables. Un élément portant cet attribut doit être associé au type **SaboList** des utilitaires qui contiendra la liste des éléments associés.

```
use SaboCore\Database\Default\Attributes\JoinedColumn;
use SaboCore\Database\Default\CustomDatatypes\JoinedList;
#[JoinedColumn(
    classModel: BannedUserModel::class,
    joinConfig: ["id" => "userId"]
)]
protected JoinedList $bannedUsers;
```

- Le paramètre **classModel** représente la class de table associée - Le paramètre **joinConfig** ce tableau permet de définir les clés de jointures. Il est indicé par le nom des attributs de la class actuelle pointant sur le nom des attributs de la class liée.
Cette structure permet la liaison via de multiples attributs - Le troisième paramètre **loadOnGeneration** permet de définir si les models liés doivent être chargés automatiquement à la génération du model. *Cela permet notamment d'éviter le chargement infini entre deux models.*

i Dans le cas où le chargement se veut manuel, la création d'une nouvelle instance `new JoinedColumn()->loadContent()`.

i Pour créer un nouvel attribut il vous faut créer une class qui extends de `SaboCore\Database\Default\Attributes\TableColumn` dont les méthodes et le constructeur peuvent être redéfinies au besoin. Se baser sur un type existant est recommandé.

Les conditions d'affectation

⚠ Les conditions d'affectation permettent d'ajouter des barrières à passer lors de l'affectation d'une valeur sur un attribut lié à une colonne. Ces conditions peuvent être posées via chaque attribut de description. Elles doivent toutes être validées afin que l'affectation opère. À l'échec d'une condition une exception est levée avec le message d'erreur associé.

Les conditions par défaut se situent dans le namespace `SaboCore\Database\Default\Conditions`.

- `RegexCond` permet d'associer une regex pour valider la donnée fournie.
- `LenCond` permet d'associer une limitation en taille sur une donnée de type `chaine`.
- `JsonValidityCond` permet de vérifier que la donnée est un tableau pour la conversion json.
- `FilterCond` permet d'appliquer une vérification `filter_var`
- `EnumValidity` permet de vérifier que la donnée est acceptée dans l'énumération liée
- `DatetimeCond` vérifie que la chaîne fournie est une `Datetime` correcte.
- `CallableCond` permet d'appeler une condition encapsulée dans une méthode statique

i Pour créer une nouvelle condition personnalisée, veuillez créer une class implémentant l'interface `SaboCore\Database\Default\Conditions\Cond`.

Les utilitaires de formatage de données

⚠ Le formatage de donnée permet d'ajouter une couche d'abstraction entre le format fourni et récupéré. Les utilitaires de formatage bien qu'ayant la même structure se distinguent en deux groupes : **reformer** groupe qui permet à partir de la donnée stockée de formater à la récupération, **formater** formater une donnée avant de la stocker dans l'attribut. Ces conditions peuvent être posées via chaque attribut de description.

Les 'formater' par défaut se trouvent dans le namespace
`SaboCore\Database\Default\Formatters`.

- `JsonFormatter` formate un tableau en chaine json pour le stockage en base de données.
- `JsonReformer` reformate une chaine json formatée en tableau.

i Pour créer un nouveau formater, veuillez créer une class implémentant l'interface `SaboCore\Database\Default\Formatters\Formatter`

Les types personnalisés

⚠ Les types customisés mentionnés plus haut, sont associé à des types d'attributs particuliers. L'attribut est donc l'élément qui défini le type à utiliser.

- `JoinedList` lié à `JoinedColumn`
- `Timestamp` lié à `TimestampColumn`

i Pour créer un type customisé, il suffit de créer une class, pour pouvoir l'utiliser, il faut également créer un nouvel attribut de description de colonne qui attendra une donnée du type de la class créée.

Les hooks

⚠ Les hooks sont des méthodes appelés à des moments précis du cycle de vie d'un model (des évènements). Redéfinir ces méthodes permet d'intercepter et d'effectuer les actions à l'appel de ces évènements.

⚠ Pensez à appeler les méthodes parentes, lors des redéfinitions

```
parent::beforeCreate();
```

L'énumération `SaboCore\Database\System\DatabaseActions` liste les hooks implémentées par les systèmes.

- `[before|after]Create` actions pre et post création
- `[before|after]Update` actions pre et post mise à jour
- `[before|after>Delete` actions pre et post suppression
- `[before|after]Generation` actions pre et post génération du model. *L'étape de génération consiste à l'affectation des valeurs de la base de données et autres actions de création du model.*

Les méthodes par défaut

⚠ En plus des hooks certaines méthodes utilitaires sont implémentées par défaut.

Cette liste ne contient pas toutes les méthodes définies, seulement les plus utiles.

- `create` enregistre le model
- `update` met à jour le model (basé sur les éléments déclarés comme clés primaires)
- `delete` supprime le model (basé sur les éléments déclarés comme clés primaires)
- `findOne` méthode statique permettant de récupérer un seul élément matchant les conditions
- `findAll` méthode statique permettant de récupérer les éléments matchant les conditions

i Les fonctions `findOne` et `findAll` se basent sur la même logique de construction de requête que le `QueryBuilder`

- `setAttribute` met à jour la valeur d'un attribut en vérifiant en amont les conditions associées puis en appliquant les formateurs associés
- `getAttribute` récupère la donnée d'un attribut en appliquant les reformers associés
- `getAttributeOriginal` permet de récupérer la valeur originale stockée sans reformer
- `getColumnsConfig` fourni la configuration récupérée des colonnes.
- `getColumnConfig` fourni la configuration d'une colonne particulière via le nom de l'attribut de class associé
- `getJoinedColumnsConfig` fourni la configuration des colonnes de jointure
- `getTableNameManager` fourni une instance de l'attribut `TableName` du nom de la table
- `getFromArray` enregistre les données de la class dans un tableau indicé par le nom d'attribut de class
- `setAttributesOriginalValues` met à jour le tableau des valeurs originales




Ne mets pas à jour les valeurs dans la class

- `lastInsertId` fourni le dernier id inséré (peut être utile notamment pour l'affectation des clés primaires après insertion)
- `createFromDatabaseLine` génère un model à partir d'une ligne de la base de données (*via fetch*)
- `createFromDatabaseLines` génère les models à partir des lignes de la base de données (*via fetch*)
- `createFromDatabaseLines` génère les models à partir des lignes de la base de données (*via fetch*)
- `newInstanceOfModel` génère une nouvelle instance du model à partir de la class fournie
- `execQuery` exécute une requête via le constructeur de requête
- `loadJoinedColumns` charge les colonnes jointes

- `createModelFromLine` génère un nouveau model à partir d'une ligne directe (sous forme de tableau) de la base de données
- `buildPrimaryKeysCondOn` ajoute les conditions de clés primaires sur un constructeur de requête
- `getDatabaseConfig` fournit la configuration sous forme de class `Config` de base de données définie dans l'environnement

Générer le SQL de création d'un model

 L'utilitaire de génération permet à partir d'une instance d'un modèle, de générer le SQL de création de la table

```
<?php
use SaboCore\Database\Default\System\MysqlTableCreator;
echo MysqlTableCreator::getTableCreationFrom(model: new
YourModelInstance);
```

Construction de requêtes personnalisées

⚠ Les requêtes personnalisées passent par l'utilisation de plusieurs utilitaires différents permettant de construire via des méthodes PHP une requête SQL différente des méthodes classiques CRUD.

MysqlQueryBuilder

L'utilitaire principale, le constructeur de requête

`SaboCore\Database\Default\QueryBuilder\MysqlQueryBuilder`

```
<?php
use SaboCore\Database\Default\QueryBuilder\MysqlQueryBuilder;
MysqlQueryBuilder::createFrom(modelClass: YourModel::class);
```

- `reset` remet à 0 les éléments internes du constructeur
- `prepareRequest` prépare la requête dans l'instance de PDO fournie
- `as` définit l'alias dans la requête

⚠ Un alias par défaut est associé par défaut à chaque constructeur

- `getRealSql` fournit le SQL interne sans modification
- `getSql` fournit le SQL généré
- `getBaseModel` fournit le modèle lié au constructeur
- `getBindValues` fournit une liste de classes gestionnaires de *bind*
- `joinBuilder` permet de joindre le constructeur fourni au constructeur actuel (utilisé pour l'imbrication de requête)
- `staticRequest` permet d'écrire manuellement une requête SQL

⚠ Si possible mieux vaut utiliser les fonctions de constructions

i Les fonctions suivantes permettent de construire la requête SQL partie par partie. Les commentaires associés décrivent plus en détails chacun des paramètres.

Select

⚠ La méthode `select` permet de définir les éléments à sélectionner dans la requête. Elle attend des noms d'attribut ou des fonctions SQL encapsulées dans `MysqlFunction`. Si aucun paramètre n'est fourni alors `SELECT *`

Cet exemple montre uniquement les possibilités

```
$builder->select("username",MysqlFunction::COUNT(numberGetter:
"``{price}``"));
```

Insert

⚠ La méthode `insert` permet de définir une requête d'insertion. Elle attend un tableau indicé par les noms d'attributs de class associé aux colonnes avec comme valeur, les données à insérer.

Les données à insérer peuvent être sous la forme : `brute`, `MysqlFunction` ou `MysqlQueryBuilder`

Cet exemple montre uniquement les possibilités

```
$builder->insert(insertConfig: [
    "username" => "yahvya",
    "github" => "https://github.com/yahvya",
    "fullname" => MysqlFunction::UPPER("{username}"),
    "queryResult" => MysqlQueryBuilder::createFrom(modelClass:
YourModel::class)
```

```
->select()  
->limit(count: 1)  
]);
```

Update

⚠ La méthode `update` permet de définir une requête de mise à jour. Elle attend un tableau indicé par les noms d'attributs de class associé aux colonnes avec comme valeur, les données à mettre à jour.

Les données à mettre à jour peuvent être sous la forme : `brute`, `MysqlFunction` ou `MysqlQueryBuilder`

Cet exemple montre uniquement les possibilités

```
$builder->update(insertConfig: [  
    "username" => "yahvya",  
    "github" => "https://github.com/yahvya",  
    "fullname" => MysqlFunction::UPPER("{username}"),  
    "queryResult" => MysqlQueryBuilder::createFrom(modelClass:  
YourModel::class)  
    ->select()  
    ->limit(count: 1)  
]);
```

Delete

⚠ La méthode `delete` permet de définir une requête de suppression.

```
$builder->delete();
```

Where

⚠ La fonction `where` permet d'ajouter le mot clé `WHERE` sur la requête

```
$builder->select()->where(); # équivaux à SELECT * from ... WHERE
```

Cond

- ⚠ La fonction `cond` permet de paire avec `where`, de définir les conditions `where` via un ensemble de méthodes. Elle attend `MysqlCondition` la classe de gestion des conditions ou `MysqlCondSeparator` permettant de séparer les conditions, créer des groupes ...

Cet exemple montre uniquement les possibilités

```
$builder
    ->select()
    ->where()
    ->cond(
        new MysqlCondition(condGetter: "username", comparator:
MysqlComparator::EQUAL(), conditionValue: "yahvya"),
        MysqlCondSeparator::AND(),
        new MysqlCondition(condGetter: "email", comparator:
MysqlComparator::EQUAL(), conditionValue: "sabo.framework@github.com"),
    );
```

Having

- ⚠ La fonction `having` permet d'ajouter la clause `HAVING` à la requête, sa construction se base sur la même logique de construction que la fonction `cond` (["Cond" in "Construction de requêtes personnalisées"](#))

Order By

- ⚠ La fonction `orderBy` permet d'ajouter la clause `orderBy` à la requête. Elle prend en paramètre un tableau contenant une liste au format [nom d'attribut, "ASC|DESC"]

```
$builder
    ->select()
    ->orderBy(["price","ASC"],["id","DESC"]);
```

Group By

⚠ La fonction `groupBy` permet d'ajouter la clause `GROUP BY` à la requête. Elle prend en paramètre la liste des noms d'attributs de classe.

Cet exemple montre uniquement les possibilités

```
$builder
    ->select(MySqlFunction::COUNT(numberGetter: "{price}"))
    ->groupBy("price","id");
```


Limit


⚠ La fonction `limit` permet d'ajouter la clause `LIMIT` à la requête. Elle prend en paramètre le nombre d'éléments et optionnellement l'offset.

```
$builder
    ->select()
    ->limit(count: 1);
$builder
    ->select()
    ->limit(count: 15,offset: 30);
```


MysqlComparator

⚠ Cette class sert à ajouter un symbole de comparaison. Elle contient par défaut un panel de fonctions représentant différents modes de comparaison. Fiez-vous aux commentaires de ces fonctions pour l'utilisation.

 Certains comparateurs attendent un type de données implicites, ex: `IN()` attend que la donnée fournie derrière soit un tableau contenant les valeurs à vérifier. Les commentaires définissent les requis

 Veuillez utiliser le plus possible les fonctions déjà définies et éviter d'utiliser le constructeur.

Focus sur la méthode spéciale `REQUEST_COMPARATOR`

 Cette méthode permet de comparer une valeur avec le résultat d'une requête ex: `WHERE ID = (SELECT id from table)`.

Elle prend en paramètre une chaîne comparator spéciale, car permettant de choisir l'emplacement de la requête.

Pour une requête du style `SELECT * FROM table_1 where id = IN(SELECT id from table_2 LIMIT 4)`

```
$tableOneBuilder = MysqlQueryBuilder::createFrom(modelClass:
TableOneModel::class);
$tableTwoBuilder = MysqlQueryBuilder::createFrom(modelClass:
TableTwoModel::class);
$tableTwoBuilder
    ->select("id")
    ->limit(count: 4);
$tableOneBuilder
    ->select();
    ->where()
    ->cond(new MysqlCondition(
        condGetter: "id",
        comparator: MysqlComparator::REQUEST_COMPARATOR(comparator:
"IN({request})", queryBuilder: $tableTwoBuilder),
        conditionValue: $tableTwoBuilder
    ));
```

MysqlCondition

- ⚠ Cette class permet de définir une condition sql basé sur trois paramètres, `condGetter` le nom d'un attribut ou `MysqlFunction`, `comparator` le comparateur, `conditionValue` la valeur requise par le comparateur utilisé

MysqlCondSeparator

- ⚠ Cette class permet d'ajouter des bouts de chaine SQL, notamment des mots clés de séparation. Elle peut être donc être utilisé pour créer des groupes de condition en ajoutant (et) via `GROUP_START` et `GROUP_END` ou des mots clés customisés

```
$builder
->select()
->where()
->cond(
    MysqlCondSeparator::GROUP_START(),
    new MysqlCondition(...),
    new MysqlCondSeparator::AND(),
    new MysqlCondition(...),
    MysqlCondSeparator::GROUP_END()
    new MysqlCondSeparator::OR(),
    new MysqlCondition(...)
);
```


Créer un nouveau système

⚠ Le framework arrive par défaut avec un système permettant de gérer le système MYSQL. Toutefois, il est possible d'intégrer un nouveau système en l'implément ou en intégrant un paquet externe.

Etapes de création d'un système

1. Créer une class qui extends `SaboCore\Database\Providers\DatabaseProvider`.

Cette class permettra a minima d'initialiser le système dans le cycle de vie du framework via la méthode abstraite `initDatabase`. Cette méthode reçoit en paramètre une configuration provenant de l'environnement qui sera traité plus bas.

i Il est recommandé d'implémenter cette class en utilisant le pattern singleton
([https://fr.wikipedia.org/wiki/Singleton_\(patron_de_conception\)](https://fr.wikipedia.org/wiki/Singleton_(patron_de_conception))).

2. Enregistrer le système dans l'environnement ainsi que son mode de configuration. Pour ce faire:

- Fournir une instance du provider sur la clé `DatabaseConfig::PROVIDER->value`
- Remplir la configuration attendue par le système dans la méthode `initDatabase` via la clé `DatabaseConfig::PROVIDER_CONFIG->value`


3. La suite de l'implémentation est libre en fonction du fonctionnement, toutefois des class et interfaces de structure sont prévus dans le namespace `SaboCore\Database\System` dans le chemin de même nom.

- `DatabaseModel` permettant de définir la structure d'un model ainsi que des méthodes natives attendues
- `DatabaseCondition|DatabaseCondSeparator|DatabaseComparator` permettant l'écriture de conditions de recherche via méthodes.
- `DatabaseActions` énumération des différents événements possibles.

- DatabaseActionException exception liée à un évènement.

Exemple d'implémentation

Cet exemple va se baser sur un système fonctionnant à base d'un seul document JSON sans toutefois aller jusqu'à l'implémentation des conditions dynamiques ...

 L'exemple est à but représentatif

Implémentation du provider

```
class JsonSystemProvider extends DatabaseProvider{
    protected static array|null $jsonDocument = null;
    protected static string|null $documentPathFromStorage = null;
    #[Override]
    public function initDatabase(Config $providerConfig):void{
        $providerConfig->checkConfigs("documentPathFromStorage");
        self::$documentPathFromStorage = $providerConfig-
>getConfig(name: "documentPathFromStorage");
        $fileContent = @file_get_contents(filename:
AppStorage::buildStorageCompletePath(pathFromStorage:
self::$documentPathFromStorage) );
        if($fileContent == null)
            return;
        self::$jsonDocument = json_decode(json:
$fileContent,associative: true);
    }
    #[Override]
    public function getCon():array|null{
        return self::$jsonDocument;
    }
}
```

Enregistrement dans l'environnement

```
->setConfig(
    name: EnvConfig::DATABASE_CONFIG->value,
```

```

        value: Config::create()
        ->setConfig(name: DatabaseConfig::INIT_APP_WITH_CONNECTION-
>value,value: true)
        ->setConfig(name: DatabaseConfig::PROVIDER->value,value: new
JsonSystemProvider() )
        ->setConfig(
            name: DatabaseConfig::PROVIDER_CONFIG->value,
            value: Config::create()
                ->setConfig(name: "documentPathFromStorage",value:
"/database/json-doc.json")
            )
    )
)

```




On pourrait dans ce cas imaginer des systèmes de modèles représentant un format d'une partie du document ...

Scripts crons

⚠ Les tâches crons sont une partie importante d'une application web. Pour initialiser le framework et ses utilitaires dans un script cron, veuillez copier la ligne suivante dans le script.

```
require_once("CHEMIN_RACINE_PROJET" . "/sabo-core/cron-launcher.php");
```

La class Application

 Cette class gère et représente l'état général de l'application.

Elle permet notamment

- l'accès global aux configurations de l'application `Application::getApplicationConfig`
- l'accès et la mise à jour des données de configuration d'environnement `Application::getEnvConfig` ou `Application::setEnvConfig`
- l'accès aux données de configuration du framework `Application::getFrameworkConfig`

Les class Request et Response

⚠ Les objets `Request` et `Response` sont respectivement un objet permettant de récupérer des informations sur la requête et d'interagir avec la session ainsi qu'un objet permettant de rendre une réponse HTTP.

L'objet Request

Il existe deux manières de récupérer une instance de cet objet.

La première est de l'injecter via les paramètres de la méthode d'un `callable` gestionnaire de requête (ex: la méthode d'un controller) en ajoutant le type `Request` sur le paramètre.

La deuxième est simplement de créer une nouvelle instance la class, ***toutefois mieux vaud utiliser la première méthode pour éviter des duplicatas d'instance.***

```
function showHomePage(Request $myRequest):BladeResponse{
    return ...;
}
```

```
class MyController extends CustomController{
    public function showHomePage(Request $myRequest):BladeResponse{
        return ...;
    }
}
```

Elle propose quelques méthodes utilitaires telles que :

- `getSessionStorage` fourni l'instance interne de `SessionStorage` l'utilitaire d'interaction avec la session
- `getPostValues` fourni un tableau des valeurs `POST` demandées. Dans le cas où une des clés requises n'est pas trouvée
 - si le message d'erreur n'est pas `NULL` alors une `TreatmentException` est levée

- sinon null est retournée
- `getGetValues` même principe que `getPostValues` sur les valeurs `GET`
- `getCookieValues` même principe que `getPostValues` sur les valeurs `COOKIES`
- `getFilesValues` même principe que `getPostValues` sur les valeurs `FILES`
- `getMethod` fourni la méthode de requête utilisée formatée en minuscule (`GET`, `POST`, `PUT` ...)
- `getValuesFrom` utilitaire de récupération de donnée à partir d'un conteneur sous forme de tableau

L'objet Response

⚠ L'objet `Response` est une class abstraite implémentant par défaut la logique de rendu d'une réponse `HTTP` le framework implémente certains modèles de réponse par défaut

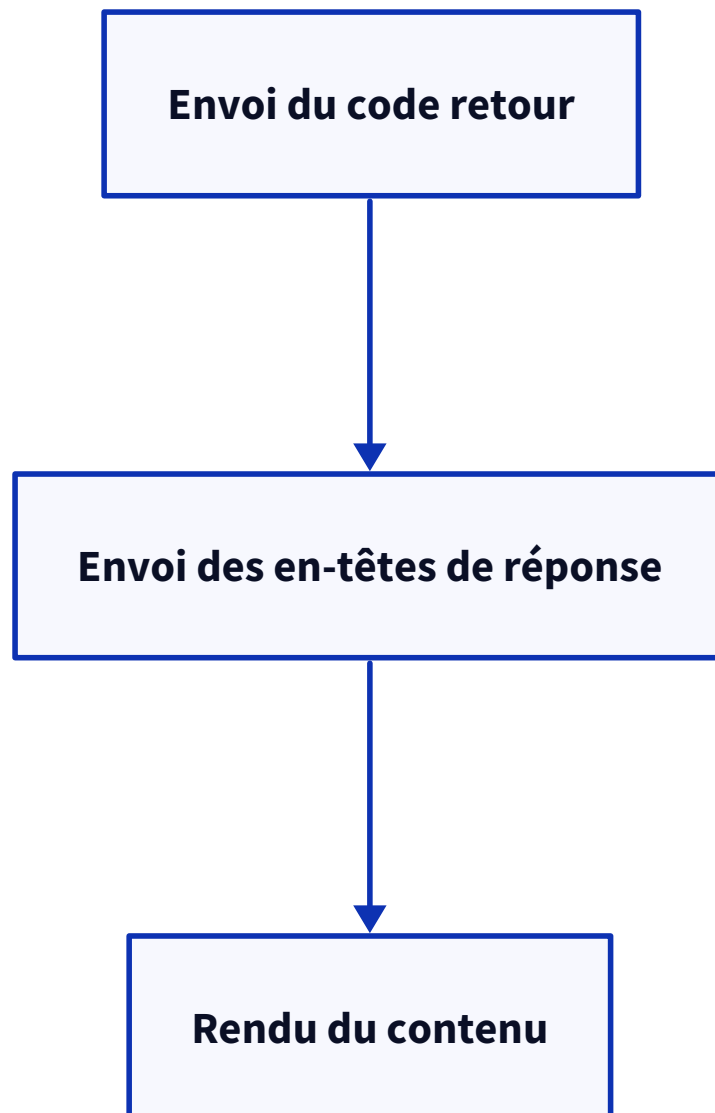
- `BladeResponse` pour rendre un visuel à partir d'un template blade
- `TwigResponse` pour rendre un visuel à partir d'un template twig
- `HtmlResponse` pour rendre un visuel à partir d'un contenu html (utilisé par blade et twig)
- `DownloadResponse` pour rendre une ressource à télécharger
- `JsonResponse` pour rendre du contenu `JSON`
- `RedirectResponse` pour rediriger sur un lien fourni
- `ResourceResponse` pour rendre une ressource *utilisé par le framework pour rendre les ressources*

Procédure de rendu d'une réponse

Procédure


1. Envoi du code retour
2. Envoi des en-têtes de réponse
3. Rendu du contenu

Schéma



Liste des en-têtes par défaut défini par le framework :

- `X-Content-Type-Options` à `nosniff`
- `Cache-Control` à `no-cache, no-store, must-revalidate`
- `Strict-Transport-Security` à `max-age=31536000; includeSubDomains`

 Des fonctions permettent toutefois de mettre à jour le code de retour ainsi que de modifier les en-têtes

- `setHeader` permet de mettre à jour un en-tête précis
- `setContent` permet de mettre à jour le contenu textuel à rendre
- `setResponseCode` permet de mettre à jour le code de retour HTTP

Cette fonction utilise l'énumération `SaboCore\Routing\Response\ResponseCode`

 Il n'est pas recommandé de mettre à jour manuellement le rendu textuel

Stockage protégé

- ⚠ En plus du dossier `public` permettant de stocker des ressources accessibles publiquement, le framework vient avec un dossier `src/storage` offrant un espace de stockage accessible seulement via l'application.

L'accès à ce stockage peut se faire via l'utilitaire `SaboCore\Utils\Storage\AppStorage`.
Référez-vous aux commentaires des fonctions pour plus de détails.

Cette class offre des fonctions de stockage et d'accès :

- `storeClassicFile` stocke une copie d'un fichier dans le dossier de stockage. Le paramètre `storagePath` prend comme racine le dossier `storage`

```
AppStorage::storeClassicFile(storagePath:
"/articles/new.md",fileBasePath: "chemin_absolue_vers_fichier.txt");
```

- `storeContent` stocke un contenu textuel dans le dossier de stockage. Le paramètre `storagePath` prend comme racine le dossier `storage`

```
AppStorage::storeContent(storagePath: "/articles/new.md",content: "#
Sabo framework doc");
```


- `storeFormFile` stocke un fichier de formulaire à partir du tmp name. Le paramètre `storagePath` prend comme racine le dossier `storage`

```
AppStorage::storeFormFile(storagePath: "/articles/new.md",fileTmpName:
$_FILES["f"]["tmp_name"]);
```

- `buildStorageCompletePath` construit le chemin absolu vers le dossier stockage en fournissant un chemin l'ayant comme racine. En partant d'un dossier souhaité `src/storage/articles` pour avoir le chemin absolu via cette fonction

```
AppStorage::buildStorageCompletePath(pathFromStorage: "/articles");
```

Storable

 L'interface `Storable` du même namespace permet la description d'un élément pouvant être stocké

Mailing

⚠ Parmi les utilitaires, le framework embarque un utilitaire de mail crée autour de la class `PHPMailer`

La class `SaboCore\Utils\Mailer\SaboMailer` permet l'envoi de mail via templates (blade, twig) en plus des fonctionnalités natives offertes par `PHPMailer`

Configuration du mailer

⚠ La configuration de mail est désactivée par défaut dans l'environnement ("[env.php](#)") in "[Les éléments de configuration](#)". Il faut décommenter les lignes et remplir les configurations requises.

Envoi de mail classique

⚠ La fonction `sendBasicMail` permet l'envoi d'un mail classique, il prend le sujet, le contenu textuel ainsi que les destinataires du mail.

```
$mailer = new SaboMailer();

$sendSuccess = $mailer->sendBasicMail(subject: "Sabo
framework",mailContent: "Mail content",recipients:
["sabo.framework@github.com"]);
```

Envoi de mail via template

⚠ La fonction `sendMailFromTemplate` permet l'envoi de mail via template, il prend le sujet, les destinataires ainsi que le gestionnaire de template

Focus sur le gestionnaire de template

Les gestionnaires de template extends de la class

`SaboCore\Utils\Mailer\MailerTemplateProvider`

Le framework en fourni deux par défaut permettant de rendre respectivement des mails à partir de templates `twig` ou `blade` ayant comme racine `src/views/mails`

`SaboCore\Utils\Mailer\BladeMailProvider` - `SaboCore\Utils\Mailer\TwigMailProvider`

```
$mailer = new SaboMailer();
$sendSuccess = $mailer->sendMailFromTemplate(
    subject: "Sabo framework",
    recipients: ["sabo.framework@github.com"],
    templateProvider: new BladeMailProvider(
        templatePath: "mail", # src/views/mail/mail.blade.php
        altContent: "Contenu alternatif",
        templateDatas: []
    )
);
$sendSuccess = $mailer->sendMailFromTemplate(
    subject: "Sabo framework",
    recipients: ["sabo.framework@github.com"],
    templateProvider: new TwigMailProvider(
        templatePath: "mail.twig", # src/views/mail/mail.twig
        altContent: "Contenu alternatif",
        templateDatas: []
    )
);
```

Accès aux routes

⚠ L'accès aux routes dans l'application passe par des fonctions définies globalement, accessible via le programme, blade mais aussi le passage de routes nommées à Javascript.

i L'implémentation pour twig peut se faire en définissant une nouvelle extension pouvant être enregistré dans la configuration en copiant l'équivalent blade.

Route nommée

`route` fonction permettant de récupérer le lien d'une route nommée

```
route(requestMethod: "get ou post ou put ...",routeName:
"nom_de_la_route",replaces: ["generic1" => 10]);
```

Passage de route à javascript

- `bladeJsRoutes` pour blade.

i Référez-vous aux commentaires des fonctions.

Utilitaires



Le framework embarque un certain nombre d'utilitaires par défaut enregistrés dans le namespace et dossier du même nom `SaboCore\Utils`

Apis

⚠ SaboApi du namespace SaboCore\Utils\Api est l'utilitaire d'appel d'api du framework.

Fonctions offertes

- `constructeur` le constructeur prend en paramètre un préfixe. Ce préfixe est stocké afin de pouvoir être utilisé pour chaque appel à la méthode `apiUrl`
- `apiUrl` fourni la concaténation du préfixe d'api enregistré avec la suite de lien fourni
- `request` permet de lancer une requête vers l'api à partir des données fournies. *Référez-vous aux commentaires pour plus de détails sur les paramètres*


⚠ L'URL demandée est une url complète (pouvant être obtenue via `apiUrl`)


- `getLastRequestResult` fourni le résultat de la dernière requête lancée accessible sous 2 formats
 - tableau via `SaboApiRequest::RESULT_AS_JSON_ARRAY`
 - chaîne de caractères via `SaboApiRequest::RESULT_AS_STRING` *cette version renvoi le résultat original de la requête*
- `ifArrayContain` cette méthode est un utilitaire permettant notamment à la suite d'une récupération au format `SaboApiRequest::RESULT_AS_JSON_ARRAY` de vérifier la présence des clés attendues *Référez-vous aux commentaires pour plus de détails sur le format*
- `createFromConfig` cette méthode statique fournit une instance de SaboApi à partir de la configuration fournie

Utilisation de la class

La class est abstraite de base, l'idée dans son implémentation est de créer une class par Api utilisée.

```
class SaboOnlineApi extends SaboApi{  
    public function __construct(){  
        parent::construct(apiUrlPrefix: "https://sabo-false-  
url.com/api");  
    }  
}
```

 L'utilisation de la méthode statique `createFromConfig` permet de créer une instance utilisable, mais il est recommandé pour une structure propre de créer une class autour de l'api utilisé

 Actuellement l'implémentation de cet utilitaire se base autour des fonctions natives de `curl`, il est prévu de l'orienter autour du package `Http`

Csrf

⚠ La protection <csrf Cross Site Request Forgery permet de protéger un site de requête non défini par elle. Plus d'informations (https://fr.wikipedia.org/wiki/Cross-site_request_forgery)

La méthode de protection commune contre ce genre d'attaque passe par l'utilisation d'un token re-généré pour les requêtes.

Afin de récupérer un token la fonction `generateCsrf()` est définie de manière globale dans `src/config/functions.php`.

Cette fonction est donc accessible dans le code ainsi que dans les templates de rendu `blade` et `twig`.

Utilisation de la fonction

La fonction retourne un objet `CsrfManager` dont la méthode `getToken` fourni le token généré.

```
{# exemple twig #}  
{{ generateCsrf().getToken() }}
```

```
{-- exemple blade --}  
{{ generateCsrf()->getToken() }}
```


```
# exemple dans le code php  
generateCsrf()->getToken()
```

Vérification du token

Afin de vérifier un token, la seconde fonction `checkCsrf` est définie de manière globale dans `src/config/functions.php`


Cette fonction prend en paramètre une chaîne supposée `token` et retourne si le token généré est valide.

Gestionnaire de fichier

 Les class de cet utilitaire permettent de gérer le contenu de fichiers stockés et provenant de formulaire.


- `FileContentManager`
- `FileManager`
- `FormFileManager`

FileContentManager


 Cet utilitaire prend en constructeur un contenu textuel de fichier et apporte des fonctions de conversions par défaut.

- `getJsonContent` converti le contenu textuel en tableau php `JSON`

FileManager


 Cet utilitaire apporte des fonctions permettant la gestion d'un fichier existant ou non à partir du chemin absolu fourni. Elle implémente l'interface `Storable`

- `fileExist` alias à la fonction `file_exists` de PHP, fourni si le fichier lié au chemin fourni existe
- `getExtension` fourni l'extension du fichier fourni

 L'extraction de l'extension sur la position du dernier '.' rencontré dans le chemin

- `getToDownload` fourni une `DownloadResponse` permettant de rendre le fichier au téléchargement
- `getPath` fourni le chemin absolu lié
- `storeIn` méthode provenant de l'interface permet de stocker le fichier dans le dossier de stockage de l'application
- `delete` alias à la fonction `unlink` de PHP, permet de supprimer le fichier lié
- `getFromStorage` génère une instance de `FileContentManager` à partir du contenu du fichier

FromFileManager


 Cet utilitaire extends `FileManager` et est destiné aux fichiers provenant de formulaire.

Elle prend en constructeur les données du fichier


```
new FormFileManager($_FILES["f"]);
```

L'utilitaire certaines fonctionnalités propres aux fichiers formulaire :

- `isInTypes` vérifie si le type du fichier se trouve dans la liste fournie
- `getErrorState` fourni l'état d'erreur du fichier
- `getSize` fourni la taille

 Les fonctions suivantes sont désactivées : `getToDownload` `getFromStorage` `delete`

Liste

 L'utilitaire `SaboList` est fait pour fournir une liste au format tableau offrant des fonctionnalités de recherche.

La class implémente les interfaces `Countable` `Iterator` `Arrayable` qui permettent

- l'itération via boucle `foreach`

```
foreach(new SaboList(datas: ["sabo","framework"]) as $element)
    var_dump($element);
```

- récupération du nombre d'éléments via la méthode `count`

Les méthodes propres à la class permettent l'accès aux données accompagné de fonctions de recherche

- `getFirst` fourni le premier élément de la liste
- `getLast` fourni le dernier élément de la liste
- `setFinder` permet modifier la fonction de recherche par défaut de la class (qui se base sur l'égalité). *Référez-vous aux commentaires de la fonction pour plus de détails*
- `getDefaultFinder` fourni la fonction de recherche par défaut de la class
- `find` fonction majeure de recherche, elle prend en paramètres les éléments à faire correspondre dans la liste basée sur la fonction de recherche courante. Elle retourne une nouvelle instance de `SaboList` contenant les éléments correspondants.

La class SessionStorage

⚠ Cet utilitaire utilisé par le framework et principalement par la class `Request` permet d'interagir avec la session.

Séparation des données

Le framework sépare les données stockées en session via un tableau avec les clés suivantes :

- `FOR_USER` clé des données fournies par les fonctions utilisateur
- `FOR_FLASH` données flash
- `FOR_FRAMEWORK` données utiles au framework
- `FOR_CSRF_TOKEN` token csrf

⚠ L'utilisateur peut définir des clés au même niveau dans la session, mais il est important de ne pas modifier ces clés ainsi que les données qu'elles contiennent manuellement

Fonctions de l'utilitaire

- `store` permet de stocker une donnée utilisateur
- `storeFlash` stocke une donnée flash

i Les données flash sont des données ayant une durée de vie plus limitée que la durée de la session. Elles ont un temps d'expiration associé à leur création ainsi qu'un nombre de rafraichissements avant d'être supprimées.

- `storeFramework` permet de stocker une donnée framework



L'utilisation de cette fonction est réservée au framework

- `getValue` récupère une donnée stockée par la fonction `store`
- `getFrameworkValue` récupère une donnée stockée par la fonction `storeFramework`



L'utilisation de cette fonction est réservée au framework

- `getFlashValue` récupère une donnée flash
- `delete` supprime une donnée stockée par la fonction `store`
- `deleteInFramework` supprime une donnée stockée par la fonction `storeFramework`



L'utilisation de cette fonction est réservée au framework

- `deleteInFlash` supprime une donnée flash
- `manageFlashDatas` fonction utilisée pour gérer la durée de vie et de rafraichissement des données flash par le framework
- `storeCsrf` permet de stocker un token csrf
- `getCsrfFrom` permet de récupérer une instance de `CsrfManager` en faisant une correspondance entre le token fourni et un token stocké
- `deleteCsrf` supprime un token csrf
- `create` crée une instance `SessionStorage`

Server Send Event

⚠ Le principe d'échange sse (Server Send Event), permet, sans passer par l'utilisation de Web Sockets, d'envoyer des messages du serveur au client.

Ce principe peut être utilisé pour des systèmes de chat, notifications ...

L'utilitaire sse permet offre des fonctions de configuration de d'envoi de message correctement formatée.

Utilisation

L'utilitaire associé est la class `SaboCore\Utils\Sse`

Cet exemple est à but représentatif des possibilités

```
$resourceManager = new ResourceManager();
$resourceManager
    ->setResource(key: "userToNotifyId",resource: 1)
    ->setResource(key: "notifyWith",resource:
NotificationConfig::MESSAGE);
$sseManager = new SaboSee(
    resourceManager: $resourceManager,
    defaultSleepTimeSec: 10 # temps entre chaque tour de boucle
);
$sseManager->launch(
    executor: function(SaboSse $manager):void{
        if(condition)
            $manager->sendEvent(eventName: "notification",eventDatas:
$manager->getResourceManager()->getResource(key: "userToNotifyId"));
        }, # Callable de gestion à chaque tour de boucle
    stopVerifier: fn(SaboSse $manager):bool => condition_stop, #
fonction renvoyant si le sse doit s'arrêter
    stopEventName: "endOfNotifications" # nom de l'évènement envoyé en
cas d'arrêt
);
```


i Pour l'exemple des fonctions ont été utilisés, il est toutefois recommandé d'utiliser une class pour gérer un échange.

Chaine de caractères

Générateur de chaine aléatoire

La class abstraite `SaboCore\Utils\String\RandomStringGenerator` fourni via la méthode `generateString` une manière de récupérer une chaine aléatoire toutefois formatée

- `length` longueur de la chaine attendue
- `removeSimilarChars` supprime les caractères similaires lors de la génération (i l L l par exemple)

 Cela peut être utile pour des codes de confirmations ...

- `tolgnore` paramètres multiples définissant le type de chaine à ignorer lors de la génération via l'énumération `SaboCore\Utils\String\RandomStringType`

Utilisation

```
RandomStringGenerator::generateString(15,false,RandomStringType::UPPERCHARS)
```

Vérifications

⚠ L'utilitaire de vérification, utilisé notamment pour définir les conditions sur les routes, permet la définition d'un processus de vérification.

Processus de vérification

1. Le constructeur prend trois paramètres `verifier` la fonction de vérifiant retournant un booléen de succès. Deux paramètres optionnels `onFailure` callable de gestion d'échec et `onSuccess` callable de gestion de succès.
2. Appel de la méthode `verify` pour lancer la fonction de vérification ou lancer le processus complet via `execVerification` en fournissant les arguments à envoyer à chacun des callables fournis en constructeur.

Utilisation

```
$loginAccessVerifier = new Verifier(  
    verifier: fn(bool $toReturn):bool => $toReturn,  
    onSuccess: function(string $sentence):void{ echo $sentence; }  
);  
$loginAccessVerifier->execVerification(verifierArgs:  
[true],onSuccessArgs: ["sabo framework"]);
```

Commandes cli

⚠ Le framework vient avec un petit utilitaire en ligne de commandes visant à automatiser certaines actions. L'utilisateur peut ajouter ses propres commandes.

Utilisation

L'utilitaire se base sur un fichier PHP sans extension `sabo`.

Il peut donc être lancé via la commande PHP `php sabo nom_de_la_commande [options de la commande]`

Liste des commandes par défaut

- `help` gestionnaire d'aide
- `serve` lance le serveur de développement

i Il est intéressant de vérifier les options de cette commande. L'option utilisant `browser-sync` lance le serveur interactif utile pour le front end.

- `make:controller` création de controller
- `make:model` création de model

Créer une commande

1. Définir une class qui extends `SaboCore\Cli\Commands\SaboCommand`
2. Enregistrer la commande dans le fichier `sabo` en suivant le même model que les commandes précédentes