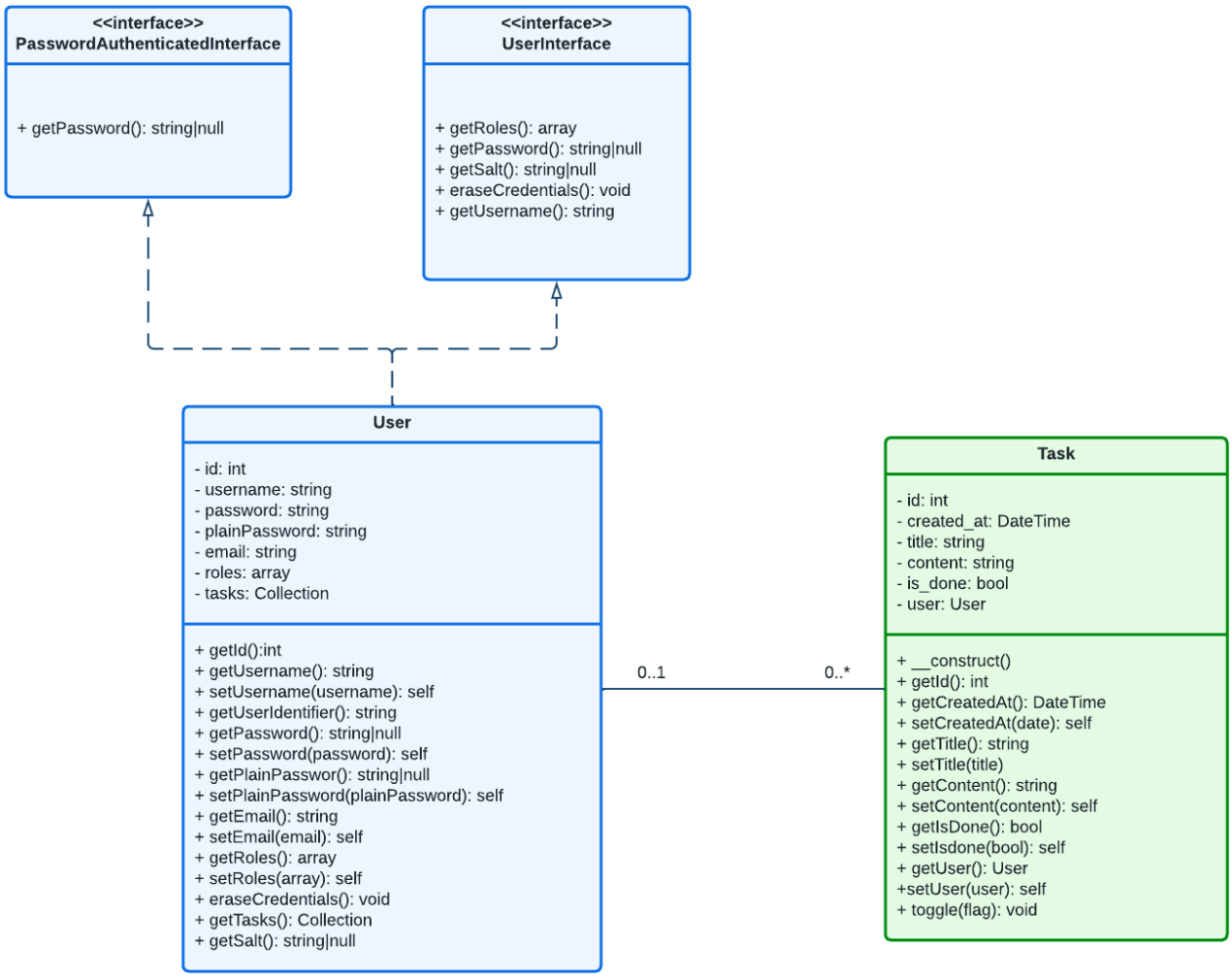
Documentation technique

Table des matières

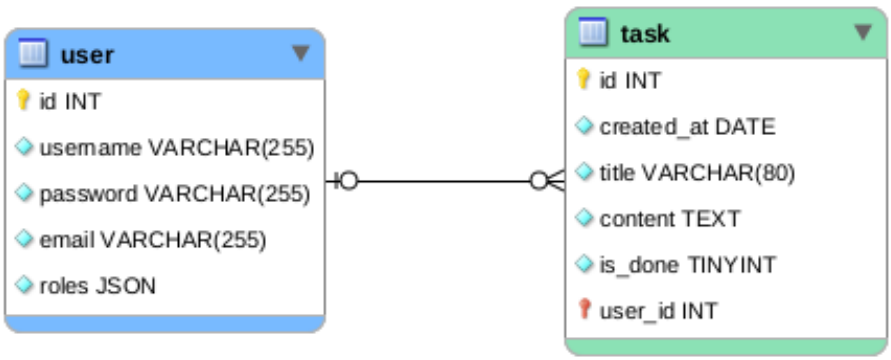
1. Objets métiers

Notre application contient deux objets métiers présents dans notre code sous forme d'entités :

- les utilisateurs de l'application: **User** (utilisé pour l'authentification)
- les tâches créées par les utilisateurs: **Task**



Ils sont également modélisés en base de donnée.



2. Sécurité

2.1 Le firewall main

Le firewall main nous permet de définir comment l'application est sécurisée.

```yaml
# config/packages/security.yaml
security:
    # ...
    firewalls:
        # ...
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\LoginFormAuthenticator
            logout:
                path: logout
```

- **lazy** empêche le démarrage d'une session si aucune autorisation n'est nécessaire (important pour garder les requêtes en cache).
- **provider** (cf 2.2.3 Fournisseur d'utilisateur).
- **custom_authenticator.**
- **logout** défini le nom de la route utilisée pour déconnecter un utilisateur.

2.2 Authentification

2.2.1 Activation de l'authentification:

```yaml
# config/packages/security.yaml
security:
    enable_authenticator_manager: true
```

2.2.2 L'utilisateur

Les autorisations sont liées à l'objet métier User représentant notre utilisateur, dont la classe implémente :

- **UserInterface** pour le chargement de l'utilisateur.
- **PasswordAuthenticationInterface** pour l'encodage du mot de passe.

2.2.3 Fournisseur d'utilisateur

Le Chargement de l'utilisateur se fait grâce à un fournisseur d'utilisateur: le provider **app_user_provider**, auquel on spécifie l'objet et la propriété choisie pour définir notre utilisateur.

```yaml
security:
    # ...
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
```

2.2.4 Encodage des mots de passe

Il est important que les mot de passe ne soient pas stockés en clair en base de donnée, nous indiquons donc l'utilisation d'un encodeur, dont l'algorithme défini sur auto permettra l'utilisation de bcrypt ou sodium(argon2i, argon2id), selon la configuration de PHP.

```yaml
# config/packages/security.yaml
security:
    # ...
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
        App\Entity\User:
            algorithm: auto
```

2.2.5 Connexion

L'utilisateur se connecte via le formulaire fournit par la méthode loginAction( ) du contrôleur SecurityController.

2.2.6 Authentification

Elle se fait grâce aux méthodes de la classe LoginFormAuthenticator.

- **start( )** redirige l'utilisateur non authentifié vers la page de login.
- **support( )** détermine si l'authentificateur doit être utilisé.
- **authenicate( )** authentifie et créé un passeport contenant l'utilisateur et son token de sécurité.
- **onAuthenticationSuccess( )** redirige après une authentification positive.
- **onAuthenticationFailure( )** redirige si l'authentification échoue.

2.3 Autorisation

2.3.1 Rôles de l'utilisateur

Deux rôles sont actuellement définis pour les utilisateurs connectés:

- utilisateur **ROLE_USER**: gestion des taches
- administrateur **ROLE_ADMIN**: gestion des utilisateurs, qui inclut un rôle utilisateur

```yaml
# config/packages/security.yaml
security:
    # ...
    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

2.3.2 Contrôle d'accès

Définition des limitations d'accès aux URL

```yaml
# config/packages/security.yaml
security:
    # ...
    access_control:
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/users, roles: ROLE_ADMIN }
        - { path: ^/, roles: ROLE_USER }
```

2.3.4 Voter

Les voters sont un puissant moyen pour gérer les autorisations. Ils permettent de centraliser l'ensemble de la logique d'autorisation.

Un voter **TaskVoter** a été implémenté afin d'obtenir plus de granularité au niveau du système d'autorisation d'accès aux tâches. Celui-ci étend la classe **Voter** qui elle même implémente **VoterInterface** et **CacheableVoterInterface**.

Nous y définissons des attributs qui seront utilisés dans les contrôleurs.

```php
// src/Security/Voter/TaskVoter.php
// ...
class User implements UserInterface, PaswwordAuthenticatedInterface
{
    public const EDIT   = 'TASK_EDIT';
    public const TOGGLE = 'TASK_TOGGLE';
    public const DELETE = 'TASK_DELETE';
```

- La méthode **support( )** détermine si le voter doit être utilisé.
- La méthode **voteOnAttribute( )** vérifie si l'utilisateur n'est pas anonyme, et le cas échéant vérifie les conditions d'accès liées aux attributs définis précédemment.

A l'intérieur du contrôleur TaskController, en début de chaque méthode est appelé la méthode **denyAccesUnlessGranted( )** qui appelle le système de voter pour vérifier les droits d'accès d'un utilisateur sur la route appelée par celui-ci.

```php
// src/Controller/TaskController.php
// ...
class TaskController extends AbstractController
{
    // ...
    public function editAction(Task $task, Request $request): Response
    {
        $this->denyAccessUnlessGranted('TASK_EDIT', $task);
```

2.4 Protection contre les failles CSRF

Il suffit pour cela de procéder à l'activation d'un token dédié, présent dans tous les formulaires, y compris celui de connexion.

```yaml
# config/packages/framework.yaml
framework:
# ...
    csrf_protection: true
```

2.5 Sessions

Une fois l'utilisateur connecté, nous utilisons le système de session pour stocker les informations de l'utlisiateur entre les requètes.

```yaml
# config/packages/framework.yaml
framework:
# ...
    session:
        handler_id: session.handler.native_file
        cookie_secure: auto
        cookie_samesite: strict
```

- **handler_id** correspond à l'id de service utilisé pour la gestion des sessions
- **cookie_secure** détermine si les cookies doivent être envoyés au travers de connexions sécurisées.
- **coockie_same_site** réglé sur strict permet de ne jamaius envoyer de cookie lorsque la requète HTTP provient d'un domaine différent de celui associé aux cookies. Utile dans la prévention d'attaques CSRF
- **storage_factory_id** correspond à l'id de service utilisé pour le stokage des sessions

3. Base de données

Les utilisateurs et les tâches sont enregistrés en base de donnée Mysql, dont le DSN (Data Source Name) est stocké dans une variable d'environement **DATABASE_URL.**

3.1 Requêtes

Les entités **User** et **Task** utilisent chaune une classe repository contenant la logique de requête.

3.2 Contraintes d'unicité

Une contrainte d'unicité est nécessaire sur les champs username et email de la table user:

- **username** permet d'identifier un utilisateur lors de la connexion
- **email** permet de s'assurer qu'un seul utilisateur est créé par adresse email

Cette contrainte est définie dans l'entité User.

```
// src/Entity/User.php
// ...
class User implements UserInterface, PaswwordAuthenticatedInterface
{
    // ...
    /**
     * @ORM\Column(type="string", length=25, unique=true)
     */
    private ?string $username;
    // ...
    /**
     * @ORM\Column(type="string", length=60, unique=true)
     */
    private ?string $email;
        // ...
```

3.3 Migrations

Le fichier **Version20220527164100.php** dans le dossier migration contient la requête SQL nécessaire pour créer la base de données. Le code généré prend en compote le système RGBD défini dans le DSN.

En cas de changement de structure de la base de données, il faudra générer une nouvelle migration puis la faire migrer via la commande doctrine, ce qui permettra de suivre plus facilement les changements apportés.