Travaux pratiques – Application Mes Voyages

Étape 1 : Préparation des outils et construction des bases de l'application

Tâche 1 : Installations des outils

- Mise en place de **WampServer** pour héberger le projet localement.
- Installation de **Composer** pour la gestion des dépendances PHP.
- Installation de **Git** pour le versioning.
- Configuration de **NetBeans** comme IDE principal.

Tâche 2 : Création du projet Symfony

- Initialisation d'un projet avec Symfony via la commande composer create-project.
- Ajout des composants nécessaires (webapp, apache-pack) pour en faire un site web.

Tâche 3 : Configuration du dépôt Git

- Initialisation d'un dépôt Git local.
- Création d'un dépôt distant sur GitHub et liaison entre les deux.
- Réalisation de plusieurs commits correspondant aux étapes du projet.

Tâche 4 : Début du développement de l'application

- Création d'un **contrôleur** et d'une **route** d'accueil ("/") affichant une page "Hello world !".
- Intégration de **Twig**, le moteur de templates, pour générer les vues.
- Mise en place de plusieurs pages (accueil, voyages) avec navigation via des routes nommées.
- Utilisation de **Bootstrap** pour styliser l'interface et créer une barre de navigation responsive.

Tâche 5 : Structuration avec Symfony

- Utilisation du modèle MVC avec les fichiers dans src, templates, public, etc.
- Définition des environnements de travail (dev par défaut).
- Compréhension du fonctionnement d'une requête HTTP dans Symfony.

Étape 2 : ORM avec Doctrine

Tâche 1 : Connexion à la base de données

- Configuration de l'accès dans le fichier .env.
- Création de la base voyages via la commande Symfony.

Tâche 2 : Création de l’Entity Visite

- Utilisation de la commande make:entity pour générer la classe et ses propriétés (ville, pays, date, note, avis...).
- Génération et exécution des migrations pour créer la table correspondante.

Tâche 3 : Remplissage de la base

- Génération de données de test (100 enregistrements) via [generatedata.com].
- Insertion dans phpMyAdmin.

Tâche 4 : Affichage des données

- Utilisation de la méthode findAll() dans le contrôleur pour récupérer les enregistrements.
- Transmission à la vue voyages.html.twig avec affichage dynamique dans un tableau.

Tâche 5 : Tri et filtrage

- Ajout de méthodes personnalisées dans le repository pour trier (findAllOrderBy) ou filtrer (findByEqualValue) les résultats.
- Création de routes et formulaires Twig pour l’interaction utilisateur.

Tâche 6 : Affichage détaillé d’une visite

- Création d'une nouvelle vue avec un lien sur chaque ville pour afficher les détails complets d'une visite.

Tâche 7 : Interface d’administration

- Création d’un contrôleur et de vues spécifiques pour gérer l'administration (admin.voyages.html.twig).
- Implémentation de la suppression d’enregistrements avec confirmation.

Tâche 8 : Modification et ajout de visites

- Création de formulaires Symfony (VisiteType) pour modifier et ajouter des visites.
- Gestion des actions dans le contrôleur avec les méthodes add() et edit().

Tâche 9 : Gestion de relations entre tables

- Création de l’Entity Environnement.
- Mise en place d’une relation **ManyToMany** entre Visite et Environnement.

Étape 3 : Ajout de Bundles

Tâche 1 : Ajout d'images aux visites avec le bundle VichUploaderBundle

- Recherche du bundle via Packagist et installation avec composer require vich/uploader-bundle 2.3.2.
- Configuration du mapping dans vich_uploader.yaml pour stocker les images dans /public/images/visites.
- Modification de l'**Entity Visite** pour inclure les champs nécessaires (imageFile, imageName, etc.) et gestion via annotations.
- Intégration du champ de type FileType dans le **formulaire admin** (VisiteType) pour l'upload.
- Affichage conditionnel de l'image dans l'interface d'administration et dans la **vue publique** des visites.
- Mise en place de **validations personnalisées** (taille max, format image) avec @Assert et méthode validate().

Tâche 2 : Formulaire de contact et envoi de mail avec Symfony Mailer

- Création d'une **entité Contact** (non reliée à la BDD) avec les champs nom, email, message.
- Création du **formulaire ContactType** et intégration dans la page contact.html.twig.
- Installation et configuration de **MailDev** pour le test local des envois de mails (MAILER_DSN=smtp://localhost:1025).
- Développement d'une méthode sendMail() dans le contrôleur, utilisant MailerInterface pour envoyer un email avec les données du formulaire.
- Création d'un **template HTML de mail** _email.html.twig pour styliser le message.
- Mise en place de **messages flash** pour confirmer à l'utilisateur l'envoi du message.

Étape 4 : Sécurité

Tâche 1 : Identification des failles courantes

- **Injection SQL** : prévention via requêtes paramétrées (pas de concaténation).
- **XSS (Cross-site Scripting)** : protection contre l'injection de scripts dans les champs utilisateurs.
- **Upload de fichiers** : vigilance sur les extensions de fichiers et le nommage.
- **CSRF (Cross-Site Request Forgery)** : utilisation de tokens pour valider l'origine des requêtes.

Tâche 2 : Sécurisation des formulaires

- **Contrôle de saisie** : types de champs et contraintes définies (ex. : note entre 0 et 20).
- **Token CSRF automatique** : inclus par défaut dans les formulaires générés par Symfony.
- **Ajout manuel de token CSRF** : pour les formulaires simples, ajout d'un champ caché et vérification côté contrôleur.

Tâche 3 : Sécurisation des requêtes Doctrine

- Utilisation obligatoire de setParameter() pour intégrer les données utilisateurs dans les requêtes.
- Éviter toute concaténation directe dans les requêtes.

Étape 5 : Les Tests

Tâche 1 : Introduction aux tests

- Présentation des 4 niveaux : **unitaires**, **intégration**, **fonctionnels** et **acceptation**.
- Introduction aux concepts de **TDD** (Test Driven Development) et **tests de non-régression**.

Tâche 2 : Tests unitaires (TestCase)

- Création d'une classe VisiteTest.
- Écriture d'un test sur une méthode (ex : getDatecreationString()).
- Utilisation des assertions de PHPUnit (assertEquals, etc.).

Tâche 3 : Tests d'intégration

Sur les règles de validation (KernelTestCase) :

- Création de VisiteValidationsTest.
- Tests sur les contraintes (ex : note entre 0 et 20, cohérence tempmin/tempmax).
- Ajout d'une méthode de validation personnalisée assertErrors().

Sur les repositories :

- Création de VisiteRepositoryTest.
- Test des méthodes comme count, add, remove, findByEqualValue.
- Utilisation d'une **base de données de test** (voyages_test) + dama/doctrine-test-bundle pour rollback automatique.

Tâche 4 : Test fonctionnels

- Création de VoyagesControllerTest.
- Tests de la route /voyages, du contenu (ex : présence du titre Mes voyages), des balises HTML, et du comportement des **liens** et **formulaires de filtre**.

- Utilisation de client, crawler et assertions comme assertResponseStatusCodeSame, assertSelectorTextContains, etc.