

# Prise en main de Symfony 4.4

## Les formulaires

### Partie 1

#### Objectifs :

- ✓ Développer une petite application qui va gérer un les navires.
- ✓ Versionner le projet
- ✓ Créer des contrôleurs
- ✓ Créer et valider des formulaires

#### Environnement :

- ✓ Serveur apache : celui de wamp
- ✓ Base de données : mysql en local (wamp)
- ✓ IDE : netbeans
- ✓ virtualHost : navire.sio
- ✓ PHP 7.4.x
- ✓ Symfony 4.4
- ✓ Mysql 5.7.xx

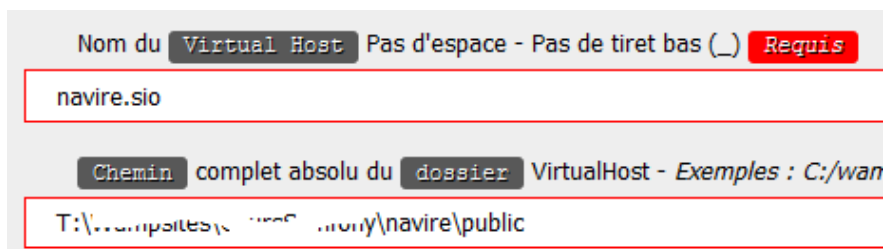
#### Partie 1 : MISE EN PLACE DU PROJET

Vous allez créer un nouveau projet appelé navire à l'aide de composer.

- ✓ Ouvrez une fenêtre de terminal
- ✓ Positionnez-vous sur le dossier parent du dossier contenant l'application
- ✓ Exécutez la commande :

```
PS T:\wamp\www\symfony> composer create-project symfony/skeleton:~4.4 navire
```

- ✓ Créez le virtualhost navire.sio qui va pointer sur le dossier navire/public



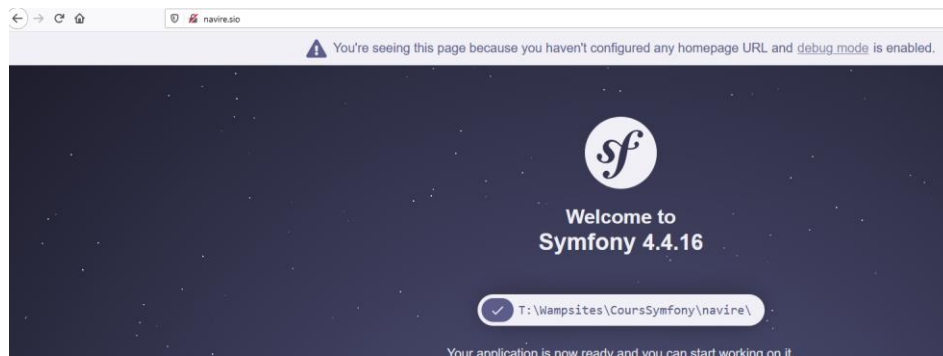
Nom du Virtual Host Pas d'espace - Pas de tiret bas (\_) Requis

navire.sio

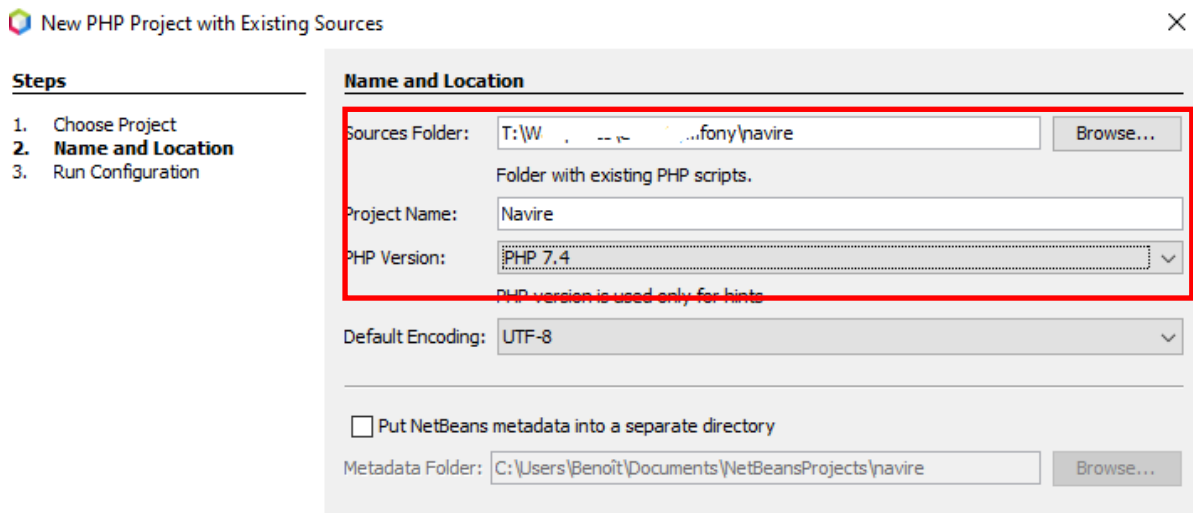
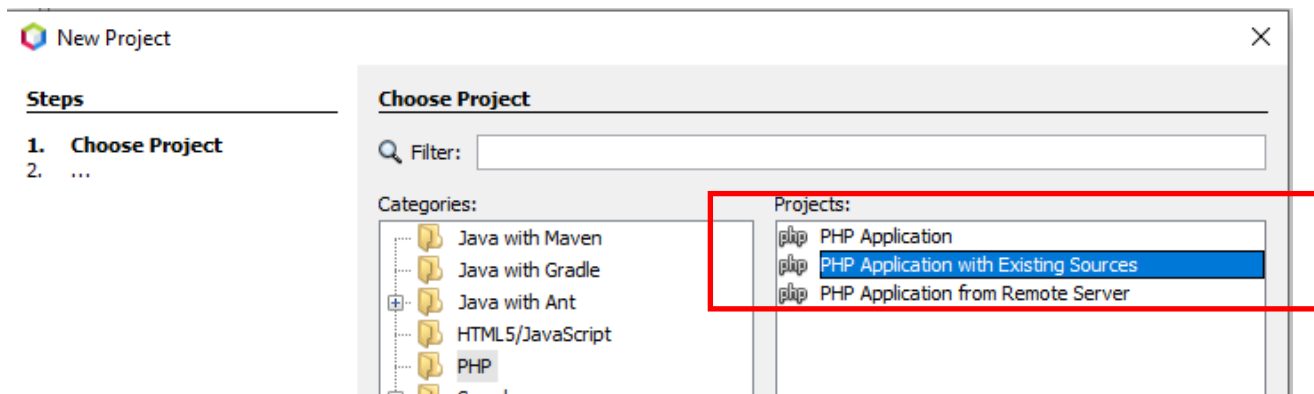
Chemin complet absolu du dossier VirtualHost - Exemples : C:/wan

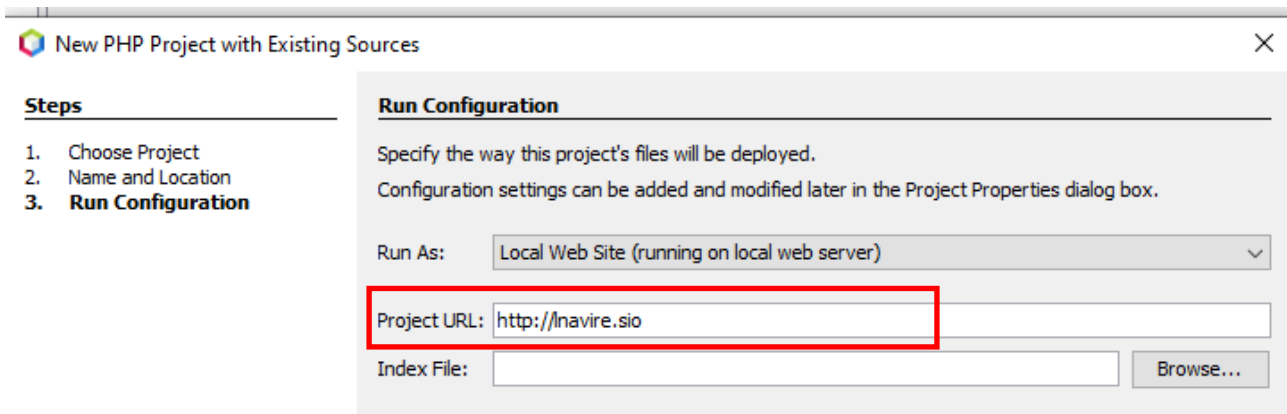
T:\wamp\www\symfony\navire\public

- ✓ Redémarrez votre DNS local
- ✓ Testez l'url <http://navire.sio/>



Vous allez maintenant ouvrir le projet dans netbeans :

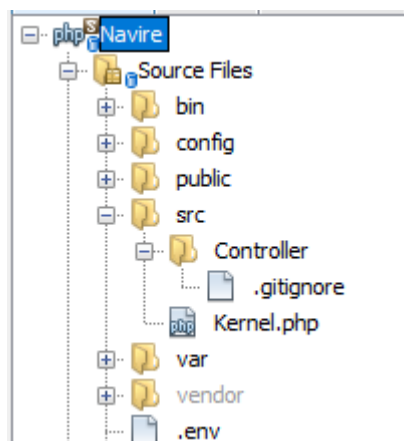




Vous pouvez tester l'url : <http://navire.sio/>



Vous devriez arriver sur le projet vierge :



Vous installerez les packages suivants :



Rappel : on installe un package par la commande

**Composer require nomDuPackage**

- ✓ symfony/maker-bundle
- ✓ sensio/framework-extra-bundle
- ✓ symfony/profiler-pack
- ✓ symfony/twig-bundle
- ✓ symfony/asset
- ✓ symfony/form
- ✓ symfony/validator
- ✓ symfony/security-bundle
- ✓ symfony/orm-pack
- ✓ symfony/apache-pack

Le fichier composer.json devrait ressembler à ceci :

```
"require": {
    "php": ">=7.1.3",
    "ext-ctype": "*",
    "ext-iconv": "*",
    "composer/package-versions-deprecated": "1.11.99.1",
    "doctrine/doctrine-bundle": "^2.2",
    "doctrine/doctrine-migrations-bundle": "^3.0",
    "doctrine/orm": "^2.7",
    "sensio/framework-extra-bundle": "^5.6",
    "symfony/apache-pack": "^1.0",
    "symfony/asset": "4.4.*",
    "symfony/console": "4.4.*",
    "symfony/dotenv": "4.4.*",
    "symfony/flex": "^1.3.1",
    "symfony/form": "4.4.*",
    "symfony/framework-bundle": "4.4.*",
    "symfony/maker-bundle": "^1.23",
    "symfony/stopwatch": "4.4.*",
    "symfony/twig-bundle": "4.4.*",
    "symfony/validator": "4.4.*",
    "symfony/web-profiler-bundle": "4.4.*",
    "symfony/yaml": "4.4.*"
},
```

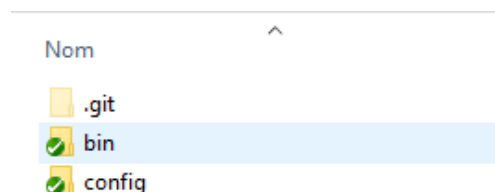
Vous allez maintenant réaliser votre premier commit et votre premier push sur un repository distant.



Vous utiliserez votre outil préféré pour réaliser cette opération.

Vous utiliserez votre plateforme distante préférée pour pousser votre code

Votre explorateur de fichier windows devrait ressembler à ceci maintenant :



Et vous devriez avoir un repository sur github ou gitlab :





Mon repository est privé, le vôtre sera **public**.

## Partie 2 : L'INTERFACE GRAPHIQUE

Vous prévoyez d'utiliser bootstrap 4.

L'interface devra présenter :

- ✓ Une barre de navigation dans la partie supérieure de l'interface
- ✓ Un corps où seront affichées les informations demandées
- ✓ Un pied de page où vous afficherez les informations légales, ...



Le reste de l'interface est à votre convenance. Vous pourrez utiliser un thème donné ou votre propre thème.



Le problème n'est pas de créer une interface graphique sophistiquée, il est de rendre l'application fonctionnelle. Vous ne consacrerez donc l'essentiel de votre temps sur l'aspect fonctionnel de l'application

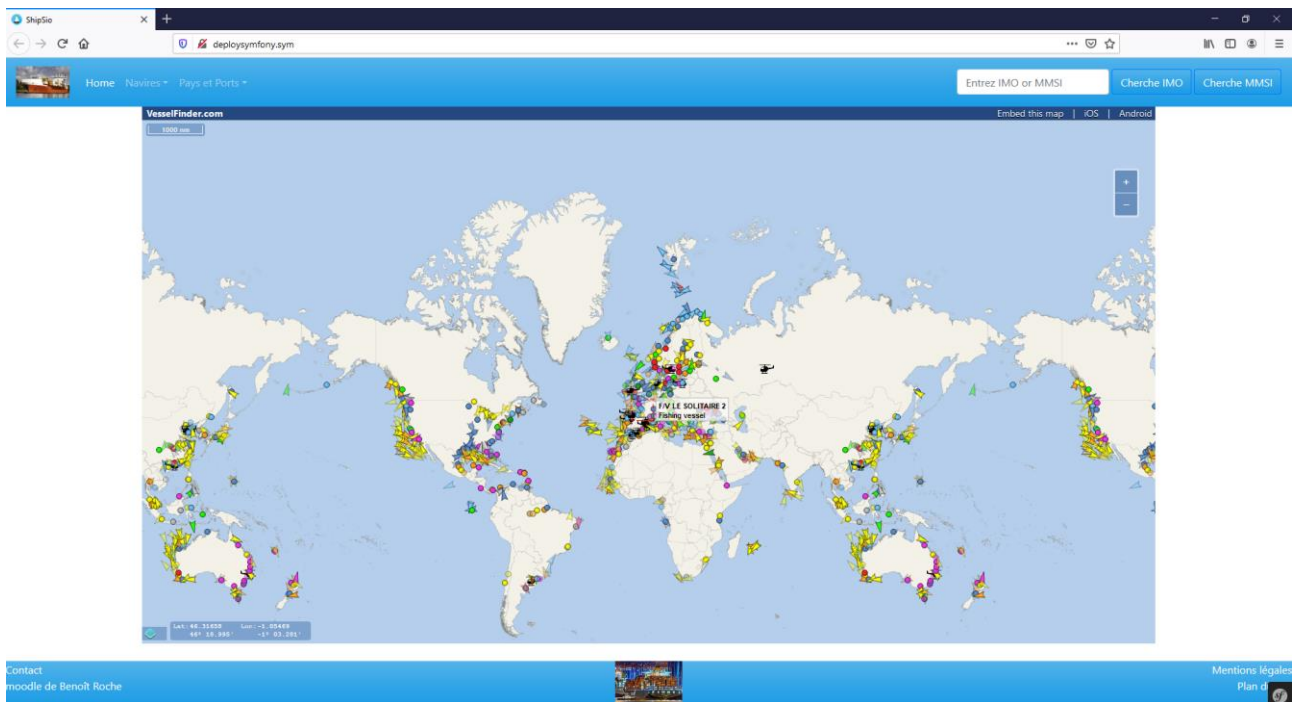
- ✓ La barre de navigation devra comporter une photo en haut à droite,
- ✓ Un formulaire de recherche
- ✓ Vous prévoyez un block *footer* dans le fichier base.html.twig pour le pied de page
- ✓ Un icône dans l'onglet du navigateur



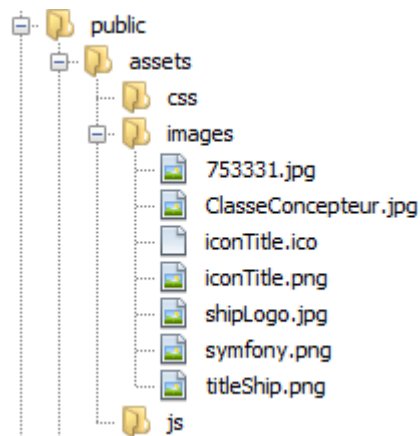
Vous travaillerez en mode CDN, il vous appartiendra donc

- ✓ de définir vous-même les ressources nécessaires à l'application,
- ✓ de mettre ces liens dans le fichier base.html.twig pour tout ce qui est commun à l'application

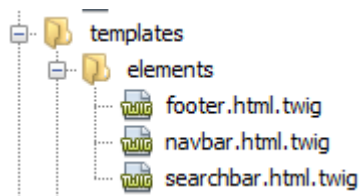
Ainsi, votre page pourrait ressembler à ceci :



Vous placerez toutes vos ressources dans le dossier public/assets :



Dans le dossier templates vous allez créer un dossier éléments qui contiendra les fichiers suivants :



✓ Le fichier navbar.html.twig :

Vous vous inspirerez du modèle fourni ici : <https://bootswatch.com/cerulean/>

## Navbars



Vous n'intègrerez pas encore le formulaire de recherche !

✓ Le fichier footer.html.twig :

```
<footer class="footer fixed-bottom bg-primary">
    {% block footer %}
        <div class="row">
            <div class="col-4 text-left txt-footer">
                <a href="#" class="text-white">Contact</a><br>
                <a href="https://moodle.benoitroche.fr" target="_blank" class="text-white">moodle de Benoît Roche</a>
            </div>
            <div class="col-4 text-center">
                
            </div>
            <div class="col-4 text-right txt-footer">
                <a href="#" class="text-white">Mentions légales</a><br>
                <a href="#" class="text-white">Plan du site</a>
            </div>
        </div>
    {% endblock %}
</footer>
```

✓ Le fichier search.html.twig



On verra plus tard

### Partie 3 : LE PROJET

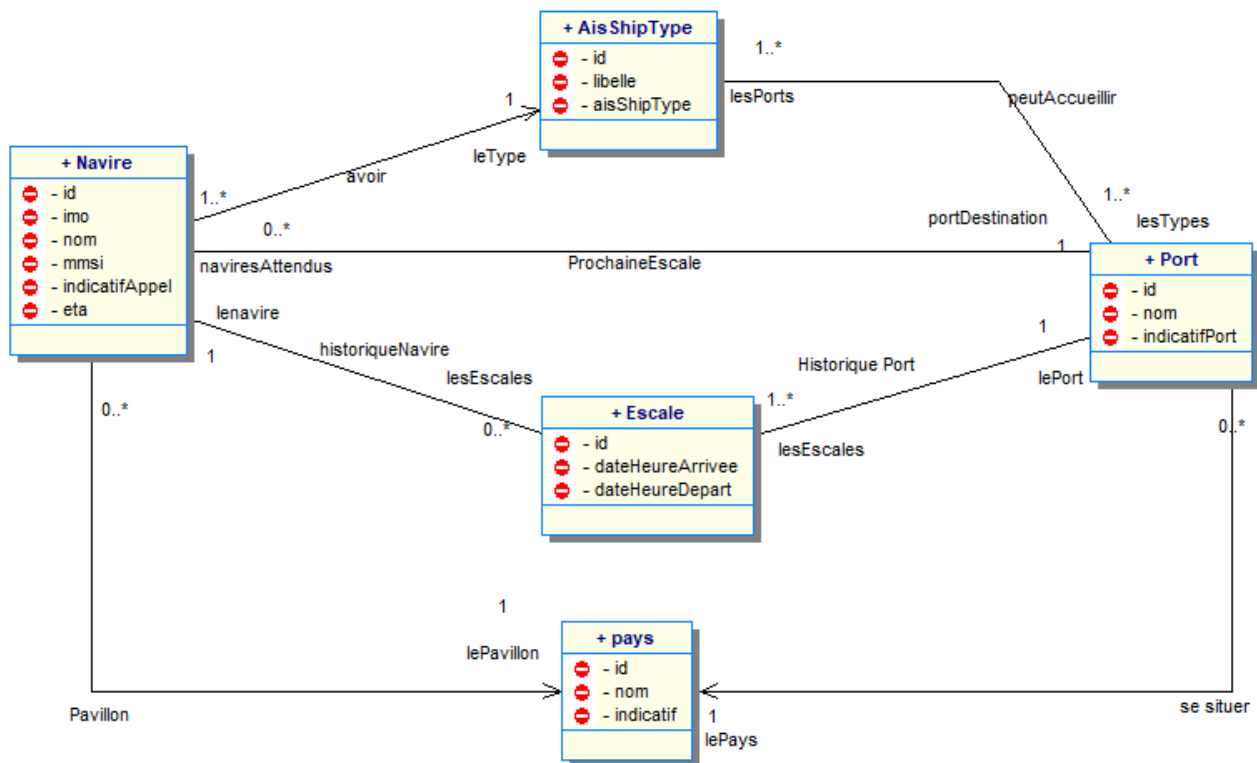
Vous développez le projet pour un armateur propriétaire de plusieurs navires et qui souhaite suivre sa flotte sur les océans du monde entier.

Il s'agira donc de gérer les navires, notamment leurs dates et heures d'escales dans les ports.

Le diagramme de classes a été partiellement validé lors de l'étape précédente du projet. :

Vous allez vous intéresser à la première User Story :

En tant que gestionnaire,  
je voudrais pouvoir lister les types de Navire,  
afin de pouvoir visualiser les ports où les navires de ce type sont susceptibles d'être accueillis



Vous créez l'entity AisShipType avec la commande php bin/console make:entity

```

private int $id;

/**...3 lines */
private string $libelle;

/**...3 lines */
private int $aisShipType;

```

La variable aisShipType représente le premier niveau de classification des navires au niveau international. ([voir ici](#)). Les valeurs permises vont de 1 à 9.

Vous gèrerez les règle d'intégrité au niveau de l'entity.

Le type de navire doit obligatoirement être compris entre 1 et 9.

Code de l'entity aisShipType hors accesseurs et mutateurs générés automatiquement :



```

class AisShipType
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="integer")
     * @Assert\Length(min=1,
     *                 max=9,
     *                 minMessage = "Le type d'un navire est compris entre 1 et 9",
     *                 maxMessage = "Le type d'un navire est compris entre 1 et 9",
     *                 allowEmptyString = false
     *                 )
     */
    private $aisShipType;

    /**
     * @ORM\Column(type="string", length=60)
     */
    private $libelle;
}

```

Vous créez de la même manière l'entity Navire.

Il faudra prendre en compte les contraintes suivantes :

- ✓ Numéro IMO : 7 chiffres
- ✓ Nom du navire : 3 caractères alphanumériques au minimum
- ✓ Numéro MMSI : 9 chiffres.

```
class Navire
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=7)
     * @Assert\Regex(
     *     pattern="/[1-9]{7}/",
     *     message="Le numéro IMO doit comporter 7 chiffres"
     * )
     */
    private $imo;

    /**
     * @ORM\Column(type="string", length=100)
     * @Assert\Length(
     *     min=3,
     *     max=100
     * )
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=9)
     */
    private $mmsi;

    /**
     * @ORM\Column(type="string", length=10)
     */
    private $indicatifAppel;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $eta;
```



Il vous restera à compléter la contrainte sur le numéro MMSI

## Partie 4 : LES CONTROLEURS ET LES FORMULAIRES

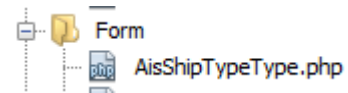
### 1. Un premier petit formulaire

Vous commencerez par créer le contrôleur AisShipTypeController à l'aide de la commande php bin/console make:controller



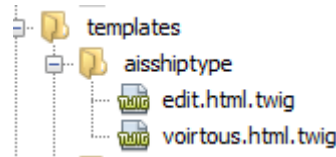
- ✓ Toutes les routes de cette classe contrôleur doivent commencer par /aishiptype
- ✓ Tous les noms de route de cette classe contrôleur doivent commencer par aishiptype\_

Vous renommerez éventuellement la classe AisShipTypeType



Mais aussi la classe elle-même : `class AisShipTypeType extends AbstractType`

Et le dossier templates :



## 2. La classe AisShipTypeController

Vous créerez ensuite le contrôleur voitous dans la classe AisShipTypeController :

```
/**
 * @Route("/voirtous", name="voirtous")
 */
public function voirTous(): Response {
    $types = [
        1 => 'Reserved',
        2 => 'Wing In Ground',
        3 => 'Special Category',
        4 => 'High-Speed Craft',
        5 => 'Special Category',
        6 => 'Passenger',
        7 => 'Cargo',
        8 => 'Tanker',
        9 => 'Other',
    ];
    return $this->render('aishiptype/voirtous.html.twig', [
        'types' => $types,
    ]);
}
```



Les types de navire sont ici en dur, il est évident que plus tard dans l'application, les types seront extraits de la base de données

## Partie 5 : FORMULAIRE DE DEMANDE D'INFORMATIONS

Vous allez maintenant gérer un formulaire de contact.

Ce formulaire s'ouvrira lorsque l'on cliquera sur le lien contact dans le footer :

Contact

Pour cela, vous allez :

- ✓ Créer une entity Message
- ✓ Créer un formulaire
- ✓ Créer le contrôleur MessageController
- ✓ Créer le template twig
- ✓ Créer un service
- ✓ Afficher ce formulaire à l'aide d'une méthode du contrôleur MessageController
- ✓ Mettre à jour le lien contact dans le footer du template de base (footer)
- ✓ Envoyer le mail

## 1. L'entity Message :



Vous ferez attention

- ✓ au type du message
- ✓ aux contraintes

```
class Message
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=60)
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=60)
     */
    private $prenom;

    /**
     * @ORM\Column(type="string", length=100)
     * @Assert\Email(
     *     message = "{{ value }}" n'est pas une adresse mail valide"
     * )
     */
    private $mail;

    /**
     * @ORM\Column(type="text")
     * @Assert\Length(
     *     min=30,
     *     minMessage = "Le message à envoyer doit faire au moins {{ limit }} caractères",
     * )
     */
    private $message;
}
```

## 2. Le formulaire

Vous créerez ensuite le formulaire de saisie du message :

php bin/console make:form

Le formulaire s'appellera *MessageType*

```
class MessageType extends AbstractType {

    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
            ->add('nom', TextType::class)
            ->add('prenom', TextType::class)
            ->add('mail', EmailType::class)
            ->add('message', TextareaType::class)
    }
}
```

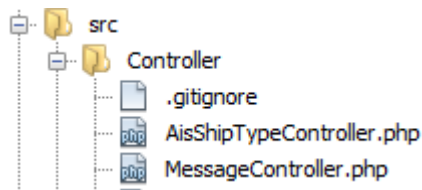


Remarquez les types des attributs mail et message

Vous retrouverez ces attributs dans le template  
templates/message/contact.html.twig

### 3. Le contrôleur MessageController

Création du contrôleur : `php bin/console make:controller`



✓ Toutes les routes de cette classe contrôleur doivent commencer par /message

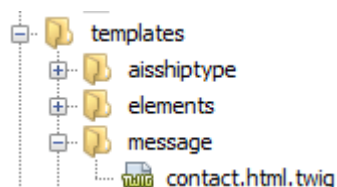
✓ Tous les noms de route de cette classe contrôleur doivent commencer par message\_



Vous reviendrez plus tard sur le code du contrôleur

### 4. Le template contact.html.twig

Vous créerez le template contact.html.twig :



```
{% extends 'base.html.twig' %}

{% block title %}Contact{% endblock %}

{% block body %}
<div class="container">
    {{ form_start(form) }}
    <div class="col-8" id="some-custom-id">
        {{ form_label(form.nom, 'Nom') }}
        {{ form_widget(form.nom, {'attr': {'placeholder': 'Nom du contact'}}) }}
    </div>
    <div class="col-8" id="some-custom-id">
        {{ form_label(form.prenom, 'Prénom') }}
        {{ form_widget(form.prenom, {'attr': {'placeholder': 'Prénom du contact'}}) }}
    </div>
    <div class="col-8" id="some-custom-id">
        {{ form_label(form.mail, 'Mail') }}
        {{ form_widget(form.mail, {'attr': {'placeholder': 'Adresse email'}}) }}
    </div>
    <div class="form-group">
        {{ form_label(form.message, 'Message') }}
        {{ form_widget(form.message, {'attr': {'placeholder': 'Que voulez-vous nous dire ?'}}) }}
    </div>

    <button type="submit" class="btn btn-success">Envoyer</button>
    {{ form_end(form) }}
</div>
{% endblock %}
```



Vous remarquerez que le bouton submit a été rajouté dans le template et non dans le formulaire MessageForm  
Pourquoi ?

Votre formulaire ressemblera à ceci :

Nom

Prénom

Mail

Message

Envoyer

## 5. Charger swiftMailer

Avant de créer le service, il vous faut charger le recipe swiftMailer :

composer require symfony/swiftmailer-bundle

MAILER\_URL=smtp://in-v3.mailjet.com:587?encryption=tls&amp;auth\_mode=logins&amp;username=b7d0cf6dd4aduzie8769693f8-00cah2&amp;password=b88...

Service  
GestionContact.php

```

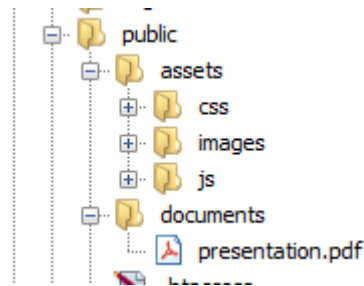
class GestionContact {
//documentation : https://swiftmailer.symfony.com/docs/sending.html
    private MailerInterface $mailer;
    private Environment $environnementTwig;
    private ManagerRegistry $doctrine;
    private MessageRepository $repo;

    function __construct(MailerInterface $mailer, Environment $environnementTwig, ManagerRegistry $doctrine, MessageRepository $repo) {
        $this->mailer = $mailer;
        $this->environnementTwig = $environnementTwig;
        $this->doctrine = $doctrine;
        $this->repo = $repo;
    }

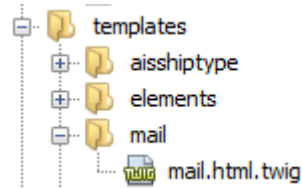
    public function envoiMailContact(Message $message) {
        $email = (new TemplatedEmail())
            ->from(new Address('symfony@benoitroche.fr', 'Contact Symfony'))
            ->to($message->getMail())
            ->subject('Nouvelle demande')
            ->text('Bonjour')
            ->attachFromPath('assets/documents/presentation.pdf', 'Présentation')
            ->htmlTemplate('mail/mail.html.twig')
            ->context([
                'message' => $message,
            ]);
        $this->mailer->send($email);
    }
}

```

Page 15 sur 24 TD\_BR\_symfony44\_Formulaires\_21V01.docx



Vous aurez aussi à concevoir le corps du message à envoyer : mail.html.twig qui se trouve dans le dossier templates/mail :



Comme on lui passe l'objet \$message ,

```

    'mail/mail.html.twig',
    ['message' => $message]
    ).

```

Vous pourrez accéder à tous les attributs de la classe Message, c'est-à-dire à toutes les informations saisies dans le formulaire.

Une idée pour le fichier mail.html.twig :

```

<h3>Demande de contact</h3>

<p>
    Message reçu de {{ message.getNom()|title }} {{ message.getPrenom()|title }}!
</p>
<p>
    mail : {{ message.getMail() }}
</p>
<br>
Message :
<p>
    {{ message.getMessage() }}
</p>

```



Vous remarquez l'utilisation des attributs de l'objet message de la classe Message !



## 7. Le contrôleur MessageController

Il reste maintenant à créer la route et la méthode qui va afficher et exploiter le formulaire.

```
/**
 * @Route("/message", name="messagee_")
 */
class MessageController extends AbstractController {

    /**
     * @Route("/contact", name="contact")
     */
    public function contact(Request $request, GestionContact $gestionContact): Response {
        $message = new Message();

        $form = $this->createForm(MessageType::class, $message);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $message = $form->getData();

            $gestionContact->envoiMailContact($message);

            return $this->redirectToRoute("home");
        }

        return $this->render('message/contact.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```



Vous remarquez le prototype de la méthode contact :

```
public function contact(Request $request, GestionContact $gestionContact): Response {
```

On fait une injection de dépendance

- ✓ de l'objet Request pour pouvoir l'exploiter

ET

- ✓ Du service GestionContact ! Il ne sera donc pas nécessaire d'instancier la classe pour accéder à ses membres non statique, symfony va s'en charger.

Il sera nécessaire d'instancier un objet \$message de la classe Message qui va être transmis au formulaire et qui va permettre de véhiculer le message du serveur au navigateur et retour via le protocole http.

Cette portion de code va s'exécuter lorsque le formulaire aura été soumis :

```

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $message = $form->getData();

        $gestionContact->envoiMailContact($message);

        return $this->redirectToRoute("home");
    }
}

```

Dans cette portion de code qui vous voyez que :

- ✓ On récupère l'objet \$message
- ✓ On appelle la méthode envoiMailContact de la classe (service) GestionContact.
- ✓ On redirige l'application sur la homepage de l'application si tout se passe

Il ne reste plus qu'à mettre à jour le lien dans le footer.

## 8. Mise à jour du lien dans le footer

Rendez-vous dans le template footer.html.twig :

```

<a href="{{ path('message_contact') }}" class="text-white" >Contact</a><br>

```

## 9. Test et envoi de mail

Rendez-vous sur la page qui affiche le formulaire et remplissez les champs :

Nom

Prénom

Mail

Message

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec consectetur dolor sapien, sed aliquet turpis consequat vitae. Quisque fermentum magna a eleifend cursus. Vestibulum mi turpis, interdum a porta at, porta a ipsum. Integer porta nisl ut neque sodales posuere.

Envoyer

Et surveillez votre messagerie :


>> Nouvelle demande

Demande de renseignement

Et la mail qui va bien:

Nouvelle demande [Se désabonner](#)  
À moi ▼

**Demande de contact**  
Message reçu de Roche Benoît!  
mail : [contact@benoitroche.fr](mailto:contact@benoitroche.fr)  
  
Message :  
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec consectetur dolor sapien, sed aliquet turpis consequat vitae. Quisque fermentum magna a eleifend



Répondre Transférer

Vous devriez revenir sur la homepage !

Essayez maintenant de saisir une adresse mail erronée :

Nom

Roche

Prénom

Benoît

Mail

jkdjlsjfdlj|

Veuillez saisir une adresse électronique valide.

Envoyer

Le contour du champ doit s'afficher en rouge lorsqu'il perd le focus et si on envoie le formulaire, il ne part pas et un message apparaît !



Amusez-vous à regarder le code source de la page !!!

## 10. Amélioration

Vous allez rajouter un flash message pour indiquer à l'utilisateur que le mail a bien été envoyé !

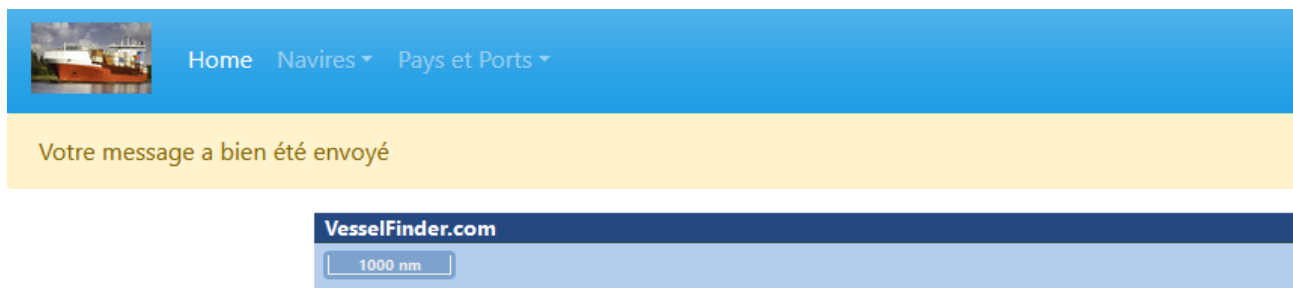
Dans le fichier base.html.twig, on va prévoir l'affichage. C'est-à-dire que quelque soit l'endroit de l'application où on créera un message 'notification', il va s'afficher sur la prochaine page appelée :

```
{% for message in app.flashes('notification') %}
    <div class="alert alert-warning">
        {{ message }}
    </div>
{% endfor %}
```

Il ne reste plus qu'à modifier le contrôleur qui a envoyé le message :

```
$this->addFlash('notification', "Votre message a bien été envoyé");
return $this->redirectToRoute("home");
}
```

Il ne vous reste plus qu'à tester !



## Partie 6 : LA BARRE DE RECHERCHE

La problématique :

Etant donné qu'on va saisir du texte dans la zone de recherche, chacun des boutons doit être de type submit. Il faut donc créer un formulaire spécifique à la recherche.

On vient de le voir, on aura très souvent un formulaire dans le block body de la page.

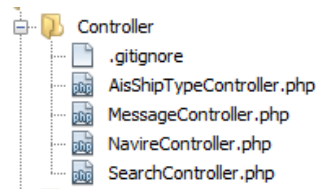
On va donc devoir gérer 2 formulaires distincts dont un sera présent sur toutes les pages.

L'idée est donc de créer de créer un formulaire dans une méthode de formulaire et de l'intégrer dans la barre de navigation (navbar).

Vous allez donc :

- ✓ Créer un contrôleur SearchController
- ✓ Créer la méthode searchBar qui va construire et appeler le formulaire
- ✓ Créer le template searchbar.html.twig qui va s'insérer dans la navbar.
- ✓ Créer le contrôleur handleSearch qui va gérer le retour du formulaire de recherche.

Vous allez donc commencer par créer le contrôleur SearchController avec la commande `make:controller`



Dans ce contrôleur vous écrirez la méthode searchBar qui va générer le formulaire :

```
public function searchBar() {
    $form = $this->createFormBuilder()
        ->setAction($this->generateUrl("search_handlesearch"))
        ->add('cherche', TextType::class)
        ->add('envoiimo', SubmitType::class)
        ->add('envoimmsi', SubmitType::class)
        ->getForm()
    ;
    return $this->render('elements/searchbar.html.twig', [
        'formSearch' => $form->createView()
    ]);
}
```



Vous remarquerez :

Que l'action que devra effectuer le formulaire sur sa soumission et indiquée dynamiquement par la méthode

```
->setAction($this->generateUrl("search_handlesearch"))
```

Qu'il y a 2 boutons submit : au retour de formulaire, on testera donc lequel des deux a envoyé le formulaire.

```
->add('envoiimo', SubmitType::class)
->add('envoimmsi', SubmitType::class)
```

Le template search.html.twig est très simple :

```
<div class="form-inline my-2">
    {{ form_start(formSearch) }}

    {{ form_widget(formSearch.cherche, {'attr': {'placeholder': 'Entrez IMO or MMSI'}}) }}
    {{ form_widget(formSearch.envoiimo, {'label': 'Cherche IMO' }) }}
    {{ form_widget(formSearch.envoimmsi, {'label': 'Cherche MMSI' }) }}

    {{ form_end(formSearch) }}
</div>
```



On pourrait le sophistiquer un peu plus !...

Vous l'avez remarqué, dans la construction du formulaire on a écrit l'action :

```
->setAction($this->generateUrl("search_handlesearch"))
```

A la soumission, le contrôleur va donc devoir générer une nouvelle URL à l'aide du nom de la route : `search_handlesearch`

Vous allez donc écrire dans le contrôleur SearchController le contrôleur handleSearch qui correspondra à cette route :

Il ne reste plus qu'à intégrer le formulaire dans la navbar c'est-à-dire rajouter le code suivant dans le template navbar.html.twig :

```

        </li>
    </ul>
    {{ render(controller(
        'App\\Controller\\SearchController::searchBar'
    )) }}
</div>
</nav>

```



Twig s'est chargé ici de réaliser dynamiquement ce que le contrôleur fait généralement. Il a généré le formulaire avec l'instruction : `render(controller)`

Vous pouvez tester :



C'est cool la barre de recherche apparait !

Il suffit maintenant de gérer le retour c'est-à-dire écrire le contrôleur qui correspond à la route `search_handlesearch` dans la classe SearchController :

```

/**
 *
 * @Route("/search/handlesearch", name="search_handlesearch")
 */
public function handleSearch(Request $request, NavireRepository $repo): Response {
    $valeur = $request->request->get('form')['cherche'];
    if (isset($request->request->get('form')['envoiimo'])) {

        $critere = "imo Recherché : " . $valeur;
    } else {

        $critere = "mmsi recherché " . $valeur;
    }

    return new Response("<h1> $critere </h1>");
}

```



Vous remarquerez  
la récupération en get de la valeur saisie dans la zone de texte,  
le test pour savoir quel bouton a envoyé le formulaire

```
$valeur = $request->request->get('form')['cherche'];  
if (isset($request->request->get('form')['envoiimo'])) {
```



Ce que fait ce formulaire n'est pas génial..... je l'avoue mais vous complèterez  
l'action dans la partie Doctrine

Vous pouvez tester :

Résultat :

imo Recherché : 9241061

Ou

**mmsi recherché 563090400**

Amélioration possible

Vous pouvez dans la formulaire de recherche :

- ✓ Rajouter un couple de boutons radio IMO/MMSI
- ✓ Ne mettre qu'un seul bouton submit

*Bravo pour votre travail, il vous reste maintenant à complexifier les formulaires grâce aux données stockées en base .....*

