

Purr Palace

**Du château de la reine à la cabane dans les marais, vous trouverez tout chez nous !**

Olga

Petrenko

**Introduction / Présentation du projet**

L’objectif de ce projet est de créer un site web dédié à la **vente et à la location de biens immobiliers**.  
Ce site se veut **intuitif, moderne et fonctionnel**, offrant aux utilisateurs une plateforme claire et agréable pour consulter les annonces de vente ou de location.

Il permettra d’effectuer une **recherche simplifiée** grâce à des filtres par type de bien, emplacement, prix ou encore selon certains critères.  
Les utilisateurs pourront également créer leur **espace personnel** afin de gérer leurs informations et réservations.  
Les propriétaires, de leur côté, auront la possibilité **d’ajouter, modifier ou supprimer** leurs logements mis en vente ou en location.  
En cas de difficulté, tout visiteur pourra **contacter l’administration** via la page de contact.

#### ****Atouts du site****

Un **design attrayant, adaptatif et ergonomique**, adapté aussi bien aux ordinateurs qu’aux smartphones.

Une **navigation fluide** et une interface claire, permettant aux utilisateurs de trouver rapidement ce qu’ils recherchent.

Une **performance optimisée**, avec un chargement rapide et une bonne sécurité des données.

#### ****Originalité du projet****

Développeuse web ambitieuse et passionnée par les univers féeriques et fantastiques, j’ai choisi d’ajouter une touche d’imaginaire à ce projet.  
Ainsi, le site transporte les utilisateurs dans un **monde magique**, où ils peuvent réserver **des châteaux, des cabanes enchantées ou des tours mystérieuses** appartenant à des personnages de contes.

Ce document présente les **objectifs du projet**, sa **structure technique**, les **fonctionnalités réalisées** ainsi que les **maquettes** illustrant l’interface du site.

**Structure fonctionnelle et technique du projet**

**Les rôles et les utilisateurs**

**Le site comprend trois types d’utilisateurs principaux :**

****Administrateur** : gère l’ensemble du site, les utilisateurs, les biens, les réservations et les offres.**

****Propriétaire** : peut ajouter, modifier ou supprimer ses logements à vendre ou à louer.**

****Utilisateur (client)** : peut consulter les annonces, effectuer des réservations ou envoyer une demande d’achat.**

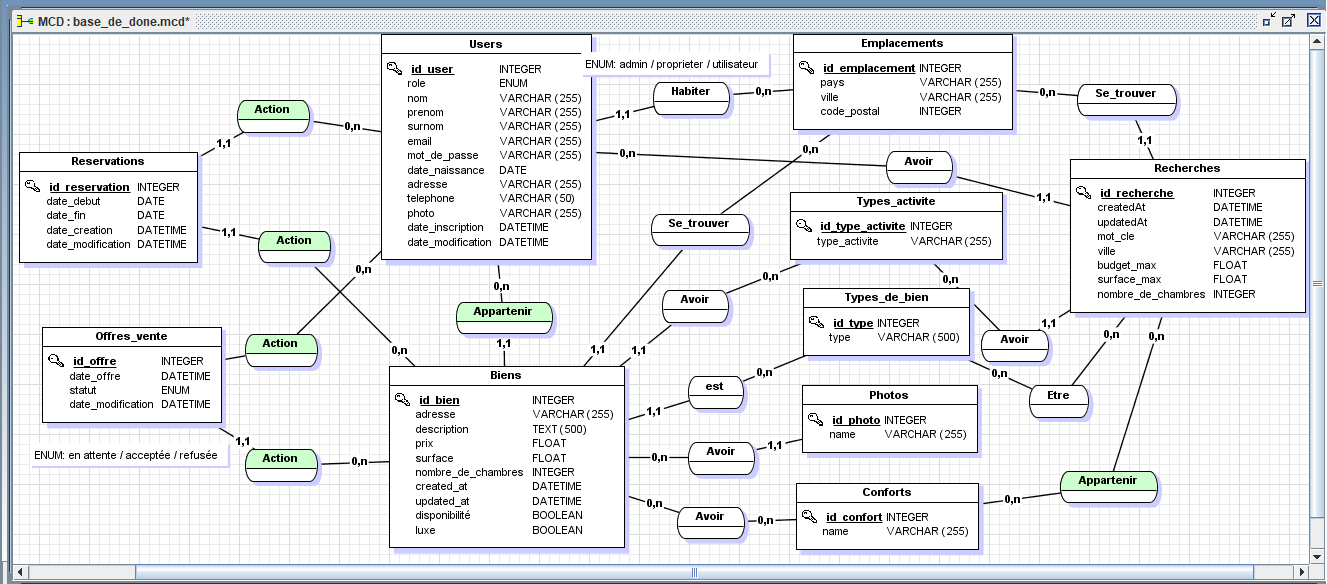
***Chaque utilisateur dispose d’un espace personnel accessible après authentification.***

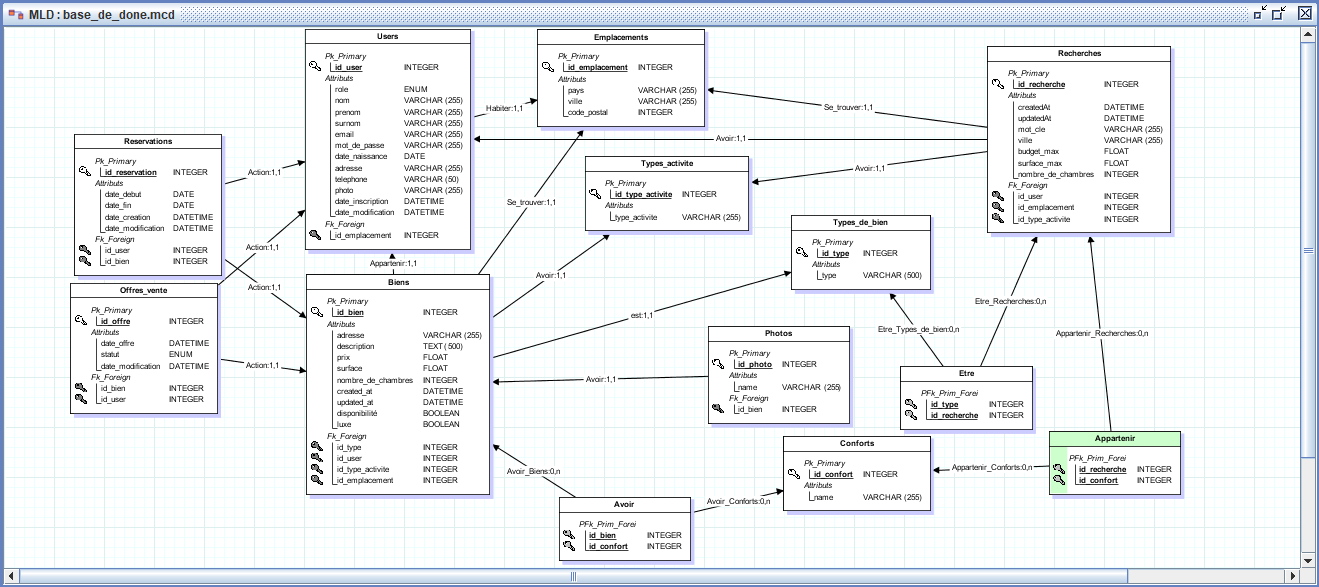
****Visiteur** : peut parcourir le site sans compte**

#### ****Modélisation des données****

La structure de la base de données repose sur plusieurs entités principales :

| **Entité** | **Description** | **Relations principales** |
| --- | --- | --- |
| **Users** | Contient les informations des utilisateurs (nom, prénom, rôle, etc.) | 1 emplacement, plusieurs logements, plusieurs réservations |
| **Biens (Logements)** | Représente les biens proposés à la vente ou à la location | 1 propriétaire, 1 type, plusieurs photos, plusieurs conforts, plusieurs réservations et offres |
| **Types\_de\_bien** | Définit le type du bien (château, maison, studio, tour...) | Plusieurs logements |
| **Type\_activités** | Vente ou location | Plusieurs logements |
| **Conforts** | Équipements et avantages (cheminée, jardin, vue...) | Relation N-N avec logements |
| **Emplacement** | Localisation du bien ou de l’utilisateur (pays, ville, code postal) | Plusieurs logements et utilisateurs |
| **Photos** | Images des biens | 1 logement |
| **Réservations** | Dates de location (courte ou longue durée) | 1 logement, 1 utilisateur |
| **Offres\_vente** | Demandes d’achat pour un logement | Plusieurs offres pour un bien |

****

****

****Cas d’utilisation principaux****

Cette section présente les principales interactions entre les différents types d’utilisateurs et le système.  
Le diagramme des cas d’utilisation illustre les fonctionnalités accessibles selon le rôle de chaque utilisateur : administrateur, propriétaire, client et visiteur.  
Chaque acteur dispose de droits spécifiques qui déterminent les actions qu’il peut effectuer sur la plateforme.

Les interactions entre les utilisateurs et le système sont les suivantes :

**Créer un compte et se connecter** : permettre aux utilisateurs d’accéder à leur espace personnel.

**Consulter la liste des logements** : afficher toutes les annonces disponibles.

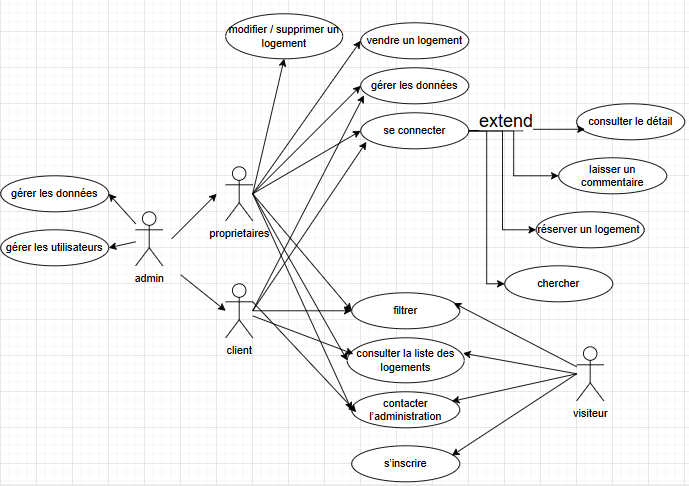
**Filtrer les annonces par critères** : rechercher un bien selon le type, la localisation, le prix ou le confort.

**Consulter le détail d’un logement** : visualiser la fiche complète d’un bien avec ses photos et caractéristiques.

**Ajouter, modifier ou supprimer un logement** (propriétaire) : gérer ses propres annonces.

**Réserver un logement ou faire une offre de vente** (client) : interagir avec les annonces en fonction du type de transaction.

**Gérer les utilisateurs et les données** (administrateur) : superviser la plateforme et administrer les contenus

.

### ****Diagramme de classes (UML)****

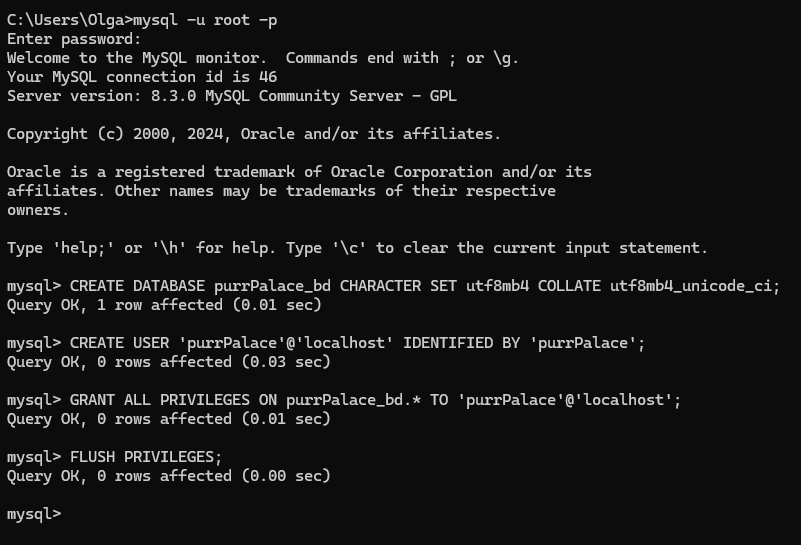
Le diagramme de classes représente la structure statique du système.  
Il met en évidence les principales entités du projet ainsi que les relations entre elles.  
Chaque classe correspond à une table dans la base de données et décrit les attributs, les clés primaires et les associations (un-à-plusieurs, plusieurs-à-plusieurs, etc.) entre les différentes entités.

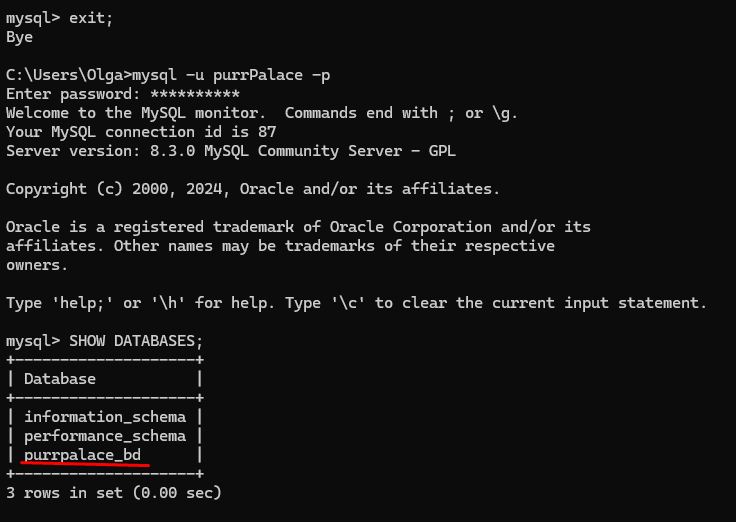
Ce diagramme permet de visualiser la logique de la base de données avant la phase de développement et d’assurer la cohérence entre les objets manipulés dans le code et les données stockées.



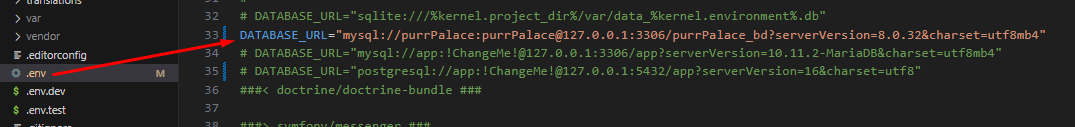
## Création et configuration de la base de données

Après la phase de modélisation (MCD, MLD et UML),  
une base de données MySQL a été créée sous le nom purrpalace\_bd afin de stocker l’ensemble des informations relatives aux utilisateurs, logements, emplacements, types de biens, activités, photos, réservations et recherches.



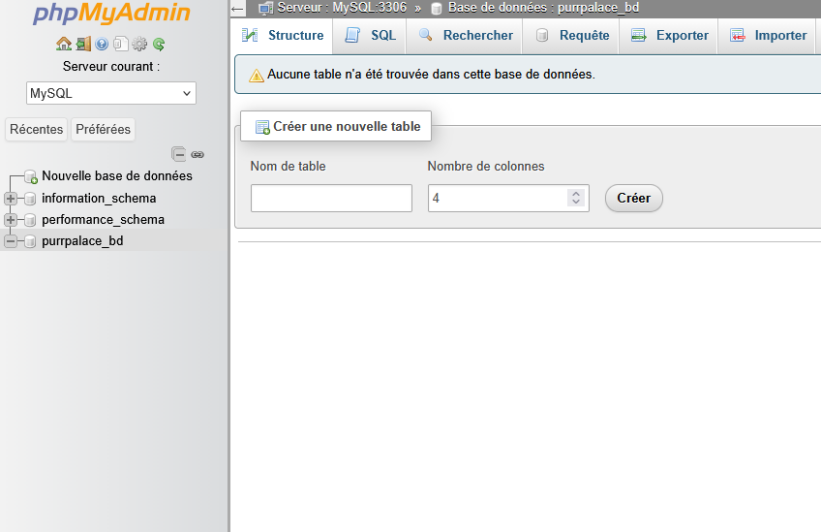
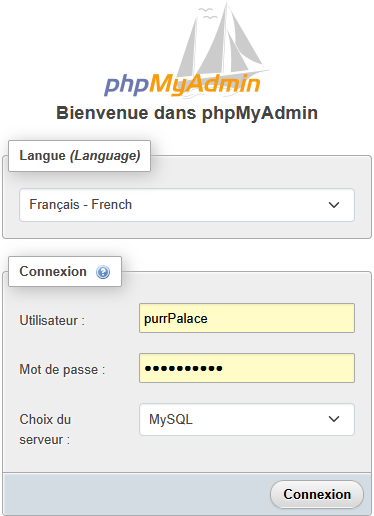


La connexion à la base de données a été configurée dans le projet Symfony à travers le fichier .env, en utilisant un utilisateur spécifique et sécurisé :



Les entités Doctrine ont ensuite été générées à partir des besoins fonctionnels identifiés dans le cahier de charges.  
Des migrations ont été exécutées pour créer automatiquement les tables et les relations définies dans le modèle logique de données.

Ainsi, la structure de la base de données respecte pleinement la modélisation initiale (relations 1:N, N:N, contraintes d’intégrité et clés étrangères) et sert de fondation au développement du site web **Symfony + Angular**.



****Utilisateur: purrPalace****

****Mot de passe: purrpalaceMaquettes et conception visuelle****

Cette section présente les maquettes fonctionnelles du site, réalisées avant le développement afin de définir la structure visuelle et la navigation entre les différentes pages.  
Les maquettes ont été conçues en noir et blanc, selon une logique de **zoning** et de **wireframe**, afin de se concentrer sur l’organisation des éléments sans se soucier encore du design graphique final.

Elles permettent de visualiser la disposition des zones principales, les parcours utilisateurs et l’ergonomie générale du site.  
Ces schémas servent également de référence pour le développement Angular et l’intégration du front-end.

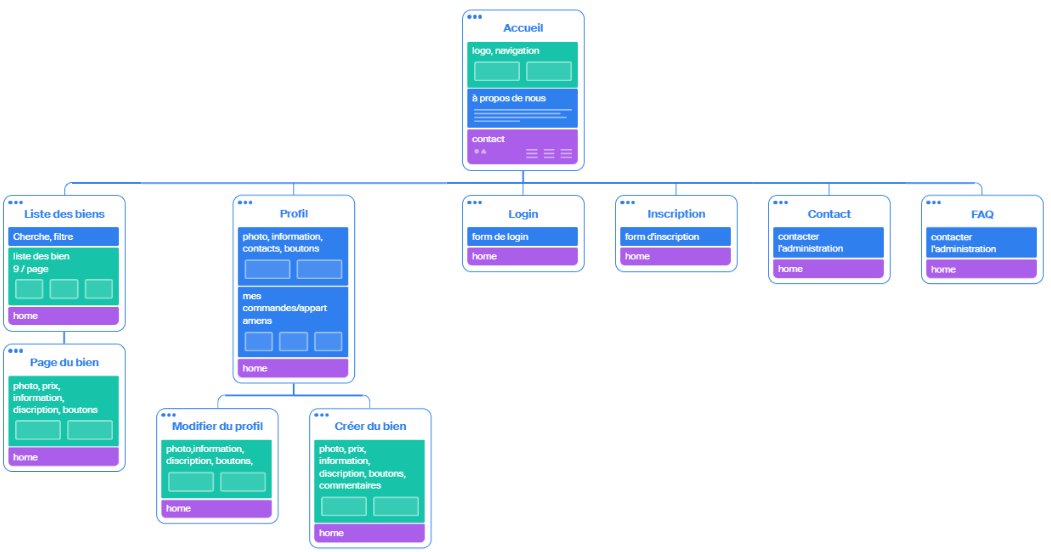
Les quatre maquettes présentées sont :

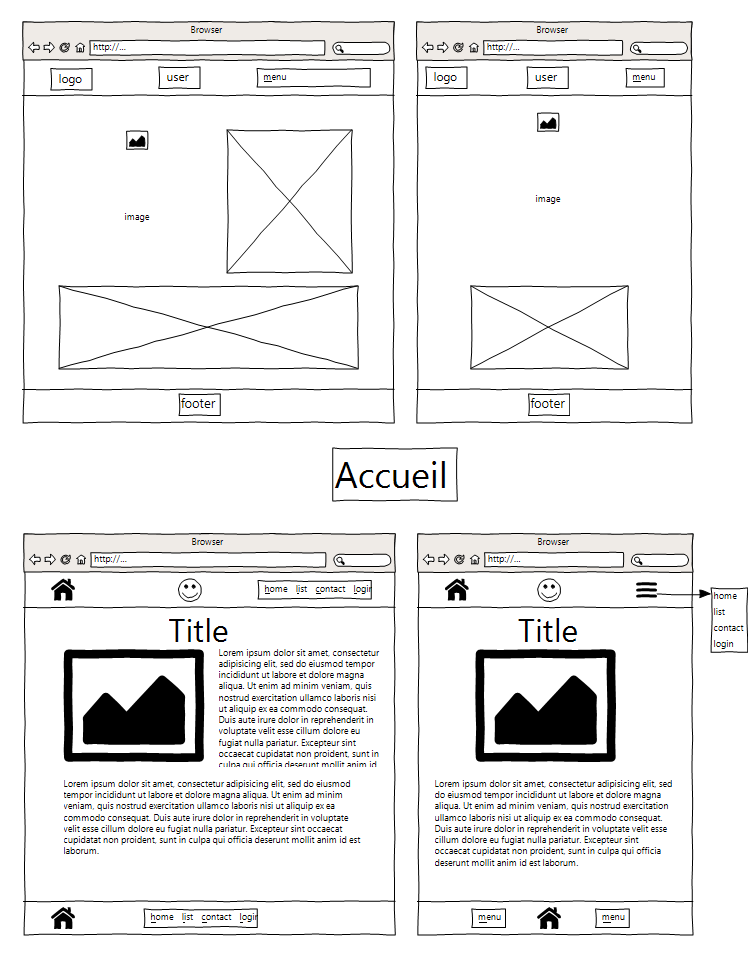
**Page d’accueil** : aperçu général du site et accès aux principales fonctionnalités.

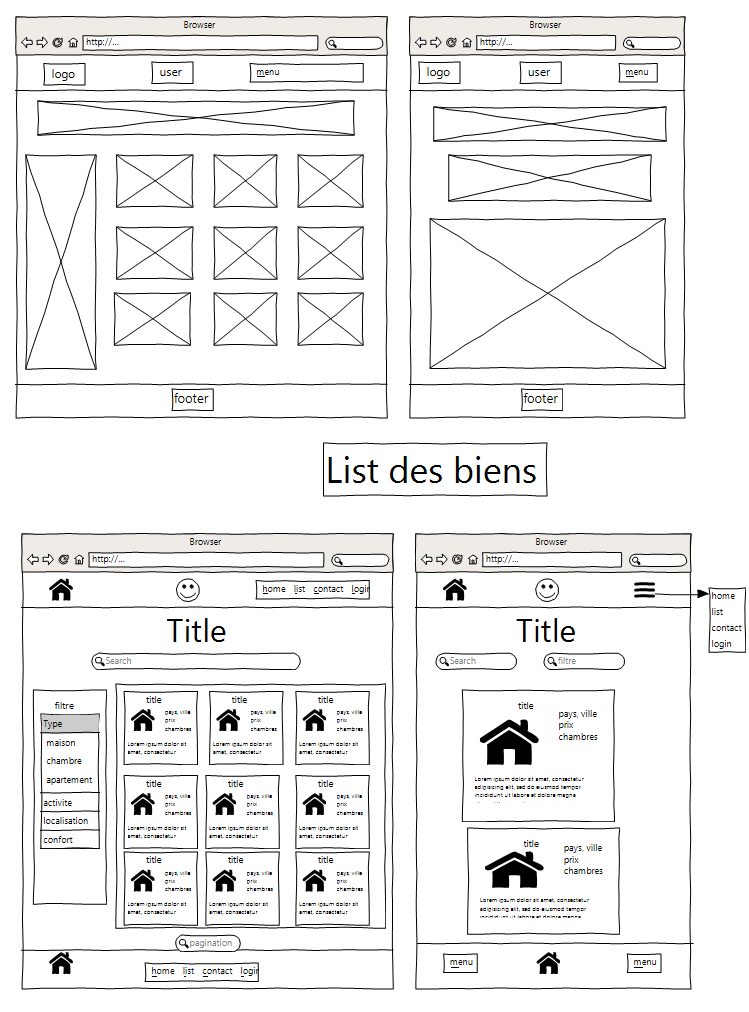
**Page de profil utilisateur** : gestion des informations personnelles et des annonces.

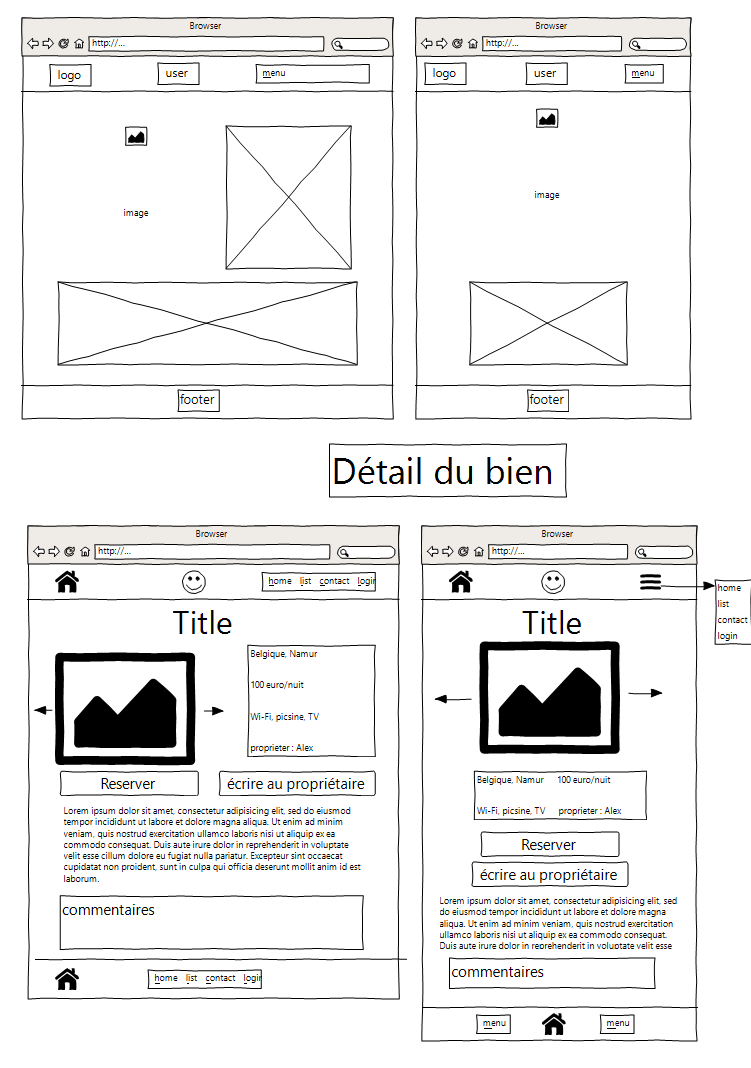
**Page de liste des logements** : affichage et filtrage des biens disponibles.

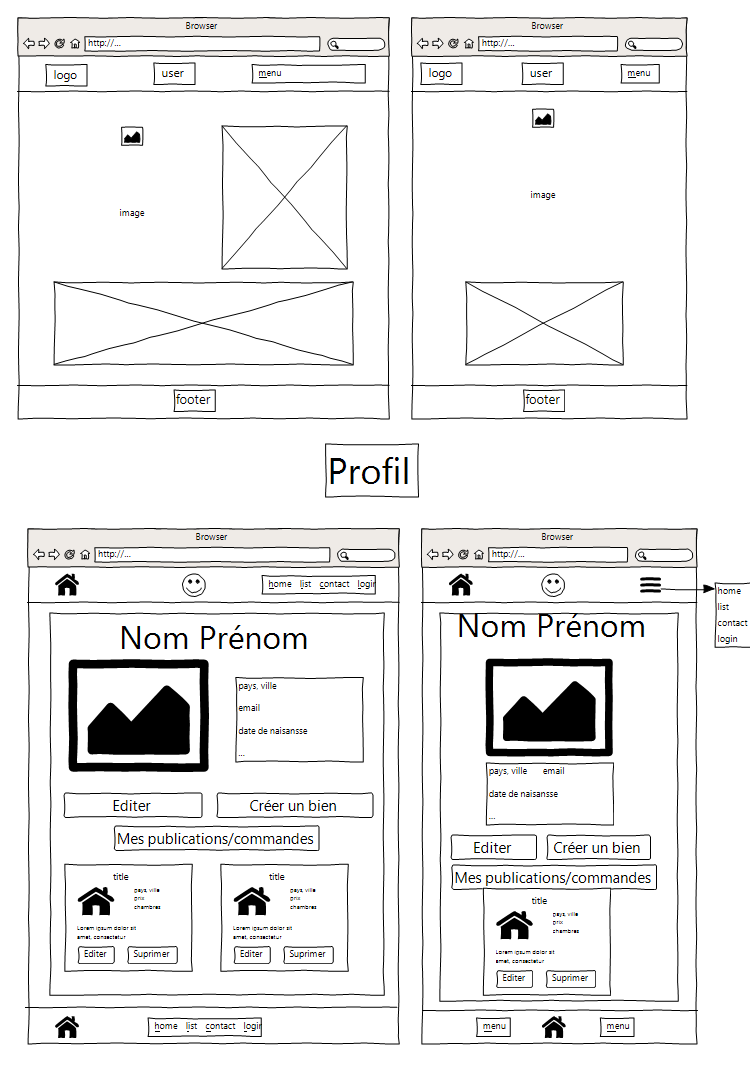
**Page de détail du logement** : présentation complète d’un bien avec ses caractéristiques et ses options d’action (réservation, offre, etc.).

****

****

****

****

****

**JSON Schema**

Schéma JSON du formulaire pour ajouter un logement (maison, appartement, etc.)

Le schéma JSON présenté ci-dessus décrit la structure complète du formulaire d’ajout d’un bien immobilier dans le cadre du projet « PurrPalace ».  
Il a pour objectif de normaliser et de valider les données envoyées lors de la création d’une nouvelle annonce de logement (vente ou location).  
Ce schéma permet d’assurer la cohérence entre les informations saisies par l’utilisateur et la structure de la base de données relationnelle utilisée par l’application.

{

"$schema": "https://json-schema.org/draft/2020-12/schema",

"title": "Formulaire d'ajout d'un bien immobilier",

"description": "Schéma JSON décrivant la structure du formulaire pour ajouter un logement (maison, studio, appartement, etc.)",

"type": "object",

"properties": {

"adresse": {

"type": "string",

"minLength": 5,

"maxLength": 255,

"description": "Adresse complète du bien"

},

"description": {

"type": "string",

"description": "Description détaillée du logement"

},

"prix": {

"type": "number",

"minimum": 0,

"description": "Prix du bien (vente ou location)"

},

"surface": {

"type": "number",

"minimum": 1,

"description": "Surface en mètres carrés"

},

"nombre\_de\_chambres": {

"type": "integer",

"minimum": 0,

"description": "Nombre de chambres"

},

"disponibilite": {

"type": "boolean",

"description": "Disponibilité du bien (true = disponible, false = non)"

},

"luxe": {

"type": "boolean",

"description": "Indique si le logement est de luxe"

},

"emplacement": {

"type": "object",

"description": "Localisation du bien",

"properties": {

"pays": { "type": "string" },

"ville": { "type": "string" },

"code\_postal": { "type": "integer" }

},

"required": ["pays", "ville", "code\_postal"]

},

"type\_de\_bien": {

"type": "integer",

"description": "Référence vers la table types\_de\_bien (ex: 1 = studio, 2 = maison, 3 = château)"

},

"type\_activite": {

"type": "integer",

"description": "Référence vers la table types\_activite (ex: 1 = location, 2 = vente)"

},

"conforts": {

"type": "array",

"description": "Liste des conforts associés au bien",

"items": {

"type": "integer",

"description": "ID d’un confort (ex: 1 = WiFi, 2 = Parking, 3 = Cheminée)"

}

},

"photos": {

"type": "array",

"description": "Photos associées au bien",

"items": {

"type": "string",

"format": "uri",

"description": "Lien ou nom de fichier de la photo"

}

},

"user\_id": {

"type": "integer",

"description": "ID du propriétaire (référence vers la table users)"

}

},

"required": [

"adresse",

"prix",

"surface",

"nombre\_de\_chambres",

"emplacement",

"type\_de\_bien",

"type\_activite"

]

}

Exemple de données correctement:

{

"adresse": "12 Rue du Château",

"description": "Magnifique château au bord du lac",

"prix": 2500000,

"surface": 800,

"nombre\_de\_chambres": 12,

"disponibilite": true,

"luxe": true,

"emplacement": {

"pays": "Belgique",

"ville": "Bruges",

"code\_postal": 8000

},

"type\_de\_bien": 3,

"type\_activite": 2,

"conforts": [1, 3, 5],

"photos": [

"https://monde-des-chateaux.com/photos/chateau1.jpg"

],

"user\_id": 1

}

**Structure générale**

Le type principal du document est un objet ("type": "object"), contenant un ensemble de propriétés représentant les champs du formulaire.

Chaque propriété correspond à une colonne de la table biens ou à une clé étrangère vers une autre table liée (par exemple emplacements, types\_de\_bien, types\_activite, conforts, photos, users).

**Détail des propriétés**

**adresse** : chaîne de caractères obligatoire contenant l’adresse complète du logement. Elle correspond à la colonne adresse dans la base de données.

Des contraintes de longueur (minLength, maxLength) permettent de garantir la qualité des données.

**description** : champ textuel facultatif permettant d’ajouter des informations complémentaires sur le bien (ex. : style, état, particularités).

**prix** : champ numérique obligatoire représentant le prix du logement (vente ou location). Une contrainte de valeur minimale (minimum: 0) empêche la saisie de montants négatifs.

**surface** : champ numérique obligatoire indiquant la surface habitable en mètres carrés. Ce champ correspond à la colonne surface de la table biens.

**nombre\_de\_chambres** : entier obligatoire indiquant le nombre total de chambres. Ce champ reflète directement la colonne nombre\_de\_chambres.

**disponibilite** : valeur booléenne (true ou false) précisant si le bien est actuellement disponible. Ce champ correspond au type TINYINT(1) dans la base.

**luxe** : champ booléen optionnel indiquant si le logement appartient à la catégorie des biens de luxe.

**Données liées**

Certaines propriétés représentent des relations avec d’autres entités du modèle de données :

**emplacement** : objet imbriqué décrivant la localisation du bien.

Il comprend les champs pays, ville et code\_postal, qui proviennent de la table emplacements.

Tous trois sont requis, garantissant que chaque logement possède une adresse géographique complète.

**type\_de\_bien** : identifiant numérique correspondant à une entrée de la table types\_de\_bien

(par exemple : 1 = studio, 2 = appartement, 3 = maison, 4 = château).

**type\_activite** : identifiant numérique faisant référence à la table types\_activite

(par exemple : 1 = location, 2 = vente, 3 = courte durée).

**conforts** : tableau d’identifiants entiers, représentant les options de confort associées au logement (connexion Wi-Fi, cheminée, parking, jardin, etc.).

Cette propriété illustre la relation many-to-many entre les tables biens et conforts.

**photos** : tableau de chaînes de caractères au format uri, contenant les liens ou noms de fichiers des images associées au bien.

Ce champ correspond à la table photos liée à biens par une relation one-to-many.

**user\_id** : identifiant numérique de l’utilisateur propriétaire du bien (clé étrangère vers la table users).

**Champs obligatoires**

La clé "required" définit les champs indispensables à la création d’un bien immobilier dans l’application :

"required": [

"adresse",

"prix",

"surface",

"nombre\_de\_chambres",

"emplacement",

"type\_de\_bien",

"type\_activite"

]

Cette contrainte garantit que les informations minimales nécessaires (adresse, caractéristiques principales et type du bien) sont toujours renseignées avant validation du formulaire.

**Rôle du schéma dans l’application**

Ce schéma JSON permet :

de valider automatiquement les données envoyées par le frontend Angular avant l’enregistrement dans la base de données via l’API Symfony ;

de documenter clairement la structure des formulaires pour les développeurs et les testeurs ;

d’assurer une cohérence stricte entre le modèle de données de l’application et les informations saisies par les utilisateurs ;

d’améliorer la sécurité et la qualité des données transmises à la base de données MySQL.

Ainsi, ce schéma constitue à la fois un outil technique de validation et un support de documentation du projet, garantissant une gestion cohérente des biens immobiliers dans l’ensemble du système **Symfony + Angular**.

## Fonctionnalités du site

Le site **PurrPalace** est une plateforme web développée avec **Symfony (backend)** et **Angular (frontend)**, permettant la gestion complète de biens immobiliers (vente et location) dans un univers thématique féerique.  
Les fonctionnalités principales sont réparties selon les rôles des utilisateurs.

### ****Authentification et rôles****

Système d’inscription et de connexion sécurisée avec gestion des rôles :

**Administrateur** : gestion complète du site, des utilisateurs et des logements.

**Propriétaire** : ajout, modification et suppression de ses propres biens.

**Utilisateur** : consultation, recherche, réservation ou dépôt d’offre d’achat.

Vérification d’adresse e-mail dans les 24 heures suivant l’inscription.

### ****Gestion des logements****

Création, modification et suppression d’un bien immobilier via un formulaire interactif.

Ajout de plusieurs photos, description, prix, surface, type, conforts et emplacement.

Association du logement à un type d’activité (vente, location, courte durée).

Possibilité de marquer un bien comme **“de luxe”**.

Gestion dynamique de la disponibilité (libre, réservé ou vendu).

### ****Réservations et offres****

Réservation de logements pour une **courte durée**.

Réservation de **longue durée** (avec date de début uniquement).

Dépôt d’offres d’achat sur un bien à vendre.

### ****Consultation et filtrage****

Liste complète des logements avec filtrage par :

pays, ville, type de bien, type d’activité, prix, surface.

Barre de recherche avec mot-clé (ex : “château”, “mer”, “forêt”).

Affichage des résultats sous forme de cartes ou de liste.

Pagination

### ****Interface multirôle****

**Administrateur** : accès à un tableau de bord via EasyAdmin, permettant la gestion complète de la base de données.

**Propriétaire** : accès à une page de profil avec la liste de ses logements et leurs réservations.

**Utilisateur** : accès à un espace personnel contenant son historique de recherches et de réservations.

### ****Design et expérience utilisateur****

Interface responsive et thématique, adaptée aux ordinateurs, tablettes et mobiles.

Identité visuelle inspirée d’un univers de contes (couleurs pastel, icônes personnalisées, éléments illustrés).

Navigation fluide et intuitive (Angular routing, animations légères).

Page d’accueil présentant le slogan :

« Chez nous, vous trouverez le château de la reine et la cabane dans les marais ! »

**Remarque :**  
Ce module de fonctionnalités met en évidence l’articulation entre la logique utilisateur (front) et la gestion des données (back), garantissant la cohérence entre l’expérience visuelle et le modèle de données.

## Implémentation technique

Le projet **PurrPalace** repose sur une architecture moderne combinant **Symfony (backend)** et **Angular (frontend)**, afin d’assurer une séparation claire entre la logique serveur, la gestion des données et l’interface utilisateur.

### ****Backend – Symfony 7 / PHP 8.3****

Le backend gère toutes les opérations liées aux données, à la sécurité et à la logique métier :

**Framework :** Symfony 7.3.4

**Langage :** PHP 8.3.6

**Base de données :** MySQL 8

**ORM :** Doctrine (migrations automatiques à partir des entités)

**Authentification :** système basé sur les rôles (Admin, Propriétaire, Utilisateur)

**API REST :** contrôleurs dédiés dans le dossier /src/Controller/Api

**Gestion des fichiers :** bundle VichUploader pour le téléchargement des photos des biens

**Interface d’administration :** EasyAdminBundle pour la gestion des utilisateurs, biens, types, emplacements et réservations

**CORS :** géré par le bundle NelmioCors pour permettre la communication entre le frontend Angular et le backend Symfony

Chaque entité Doctrine correspond à une table de la base de données MySQL, avec des relations OneToMany, ManyToOne et ManyToMany assurant l’intégrité des données (biens, utilisateurs, conforts, réservations, etc.).

### ****Frontend – Angular****

Le frontend offre une interface interactive, dynamique et responsive, permettant une navigation fluide entre les différentes pages :

**Framework :** Angular 19 (standalone components)

**Langage :** TypeScript

**Serveur local :** http://localhost:4200

**Communication avec l’API Symfony :** via le module HttpClient et des services personnalisés (UserService, BienService, FiltreService, etc.)

**Routing dynamique :** pour la navigation entre pages (accueil, liste, détail, profil, connexion, etc.)

**Composants réutilisables :** formulaires, cartes de biens, modales, barres de recherche et filtres

**Gestion d’état :** les données reçues de l’API sont stockées et affichées en temps réel

**Framework CSS :** Bootstrap 5.3 pour la mise en page responsive, complété par du CSS personnalisé

### ****Base de données – MySQL****

La base de données a été conçue à partir du **MCD** et du **MLD** générés avec **JMerise**, puis traduite en tables via **Doctrine Migrations**.  
Elle contient les entités principales suivantes :

users – informations des utilisateurs et leurs rôles

biens – logements (prix, surface, description, luxe, disponibilité)

emplacements – pays, ville, code postal

types\_de\_bien – studio, maison, villa, château, etc.

types\_activite – vente, location longue, location courte

reservations – gestion des séjours et disponibilités

offres\_vente – gestion des offres d’achat

conforts – équipements et services

photos – images associées aux biens

Toutes les relations entre les tables ont été mises en place selon le modèle conceptuel :  
1:N, N:N, 1:1, avec des clés étrangères (FOREIGN KEY) et des contraintes ON DELETE CASCADE.

### ****Interaction entre les couches****

**L’utilisateur (frontend)** interagit avec les formulaires Angular.

Les données sont envoyées via des requêtes **HTTP (GET, POST, PUT, DELETE)** à l’**API Symfony**.

Le backend traite la requête, interagit avec **Doctrine ORM** et renvoie une **réponse JSON**.

**Angular** affiche dynamiquement les résultats sans recharger la page.

Cette structure client–serveur garantit une application fluide, évolutive et facile à maintenir.

### ****Sécurité et performance****

Validation des entrées côté Angular et côté Symfony.

Séparation stricte des rôles et accès protégés par les firewalls dans security.yaml.

Gestion des images et des données sensibles via HTTPS local (127.0.0.1:8000).

Nettoyage des entrées utilisateur avant enregistrement en base.

Optimisation du rendu Angular (lazy loading, pipes personnalisés, pagination).

Cette architecture modulaire et découplée permet une évolution future du projet, par exemple l’ajout de nouvelles fonctionnalités (paiement en ligne, messagerie intégrée, carte interactive, etc.) sans remise en cause de la structure existante.

## Déroulement et organisation du travail

Le développement du projet **PurrPalace** s’est déroulé en plusieurs étapes, sur une durée d’environ deux semaines (12 à 14 jours).   
Chaque phase a permis de passer progressivement de la conception à la mise en ligne du site fonctionnel.

### ****Étape 1 — Conception du MCD, MLD et diagrammes UML****

Création du **Modèle Conceptuel de Données (MCD)** et du **Modèle Logique de Données (MLD)** à l’aide du logiciel **JMerise**.

Définition des entités principales : User, Bien, Emplacement, TypeDeBien, TypeActivite, Confort, Photo, Reservation, OffreVente.

Modélisation des relations entre les entités (1:N, N:N, 1:1).

Réalisation du **diagramme de classes UML** dans **StarUML**.

Objectif : concevoir une base de données claire, cohérente et extensible.

### ****Étape 2 — Création et configuration de la base de données****

Création de la base MySQL purrpalace\_bd via Doctrine Migrations.

Configuration de la connexion dans le fichier .env du projet Symfony.

Génération des entités avec la commande php bin/console make:entity.

Vérification de la cohérence entre la base et les schémas du MLD.

Objectif : disposer d’une base de données opérationnelle et reliée au backend.

### ****Étape 3 — Développement du backend (Symfony)****

Création du projet avec symfony new pp\_backend --webapp.

Mise en place des entités et des relations dans Doctrine.

Développement des contrôleurs API (/src/Controller/Api).

Configuration du bundle **NelmioCors** pour la communication avec Angular.

Installation d’**EasyAdminBundle** pour la gestion du contenu et des utilisateurs.

Tests des routes API via **Postman**.

Objectif : construire une API REST fiable et sécurisée pour alimenter le frontend.

### ****Étape 4 — Développement du frontend (Angular)****

Création du projet Angular avec ng new pp\_frontend --standalone.

Intégration de **Bootstrap 5.3** et configuration du style global.

Création des composants principaux :

HomeComponent (page d’accueil),

BienListComponent (liste des logements),

BienDetailComponent (fiche d’un logement),

ProfileComponent,

LoginComponent et RegisterComponent.

Développement des services (BienService, UserService, FiltreService) pour la communication avec l’API Symfony.

Ajout des routes Angular et test de l’affichage dynamique des données.

Objectif : offrir une interface moderne, responsive et intuitive.

### ****Étape 5 — Tests et finalisation****

Validation des formulaires et des appels API (création, suppression, mise à jour).

Test des filtres et de la pagination.

Vérification de la cohérence entre les rôles (Admin, Propriétaire, Utilisateur).

Correction des erreurs et optimisation des performances.

Préparation du **Cahier de charges** final, export en PDF et dépôt du projet sur GitHub.

Objectif : obtenir un site fonctionnel, stable et prêt à être présenté.

### ****Outils utilisés****

| **Catégorie** | **Outils** |
| --- | --- |
| **Backend** | Symfony 7.3.4, PHP 8.3.6, Doctrine, EasyAdmin, NelmioCors |
| **Frontend** | Angular 19, TypeScript, Bootstrap 5 |
| **Base de données** | MySQL 8, JMerise |
| **Modélisation** | StarUML, WireframeSketcher |

Ce découpage clair du travail permet de suivre la progression logique du projet et de démontrer la maîtrise de chaque étape du développement web complet (Full Stack).

****Tests unitaires****

Dans le cadre du développement du projet **PurrPalace**, une série de **tests unitaires** a été mise en place afin de garantir la fiabilité, la maintenabilité et la qualité du code.  
Ces tests ont été réalisés à l’aide du framework **Jasmine/Karma** intégré à Angular, permettant de vérifier le comportement isolé de chaque composant, service et pipe sans dépendre du reste de l’application.

Les objectifs principaux de cette phase de test étaient :

de **valider le bon fonctionnement logique** de chaque unité du code (composants, services, pipes) ;

de **détecter rapidement d’éventuelles erreurs** avant l’intégration finale ;

de **mesurer la couverture de code** pour évaluer la robustesse et la cohérence du projet.

Les fichiers testés incluent notamment :

le **service** BienService (gestion des appels API liés aux biens immobiliers),

le **composant** BienFormComponent (création et édition de logements),

et le **pipe** TruncatePipe (affichage tronqué et sécurisé des textes longs).

### ****Tests unitaires – Service**** BienService

Les tests unitaires du service **BienService** ont été réalisés afin de vérifier le bon fonctionnement des appels HTTP et la logique interne de gestion des biens (logements).  
L’objectif de ces tests était d’assurer que le service communique correctement avec l’API Symfony et traite les données de manière fiable, tout en respectant la structure prévue dans l’application.

Les principaux points testés :

✅ Vérification de la création correcte du service (should be created).

✅ Test du chargement de la liste paginée des biens (getPaginatedBiens).

✅ Test du chargement d’un bien par son identifiant (getBienById).

✅ Test de la création d’un nouveau bien (createBien).

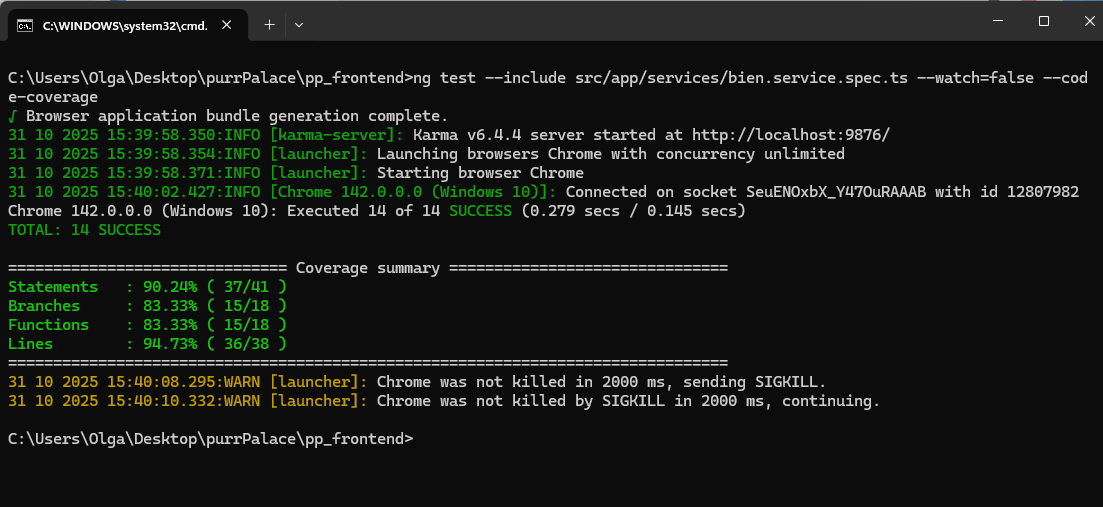
✅ Test de la mise à jour et de la suppression d’un bien (updateBien, deleteBien).

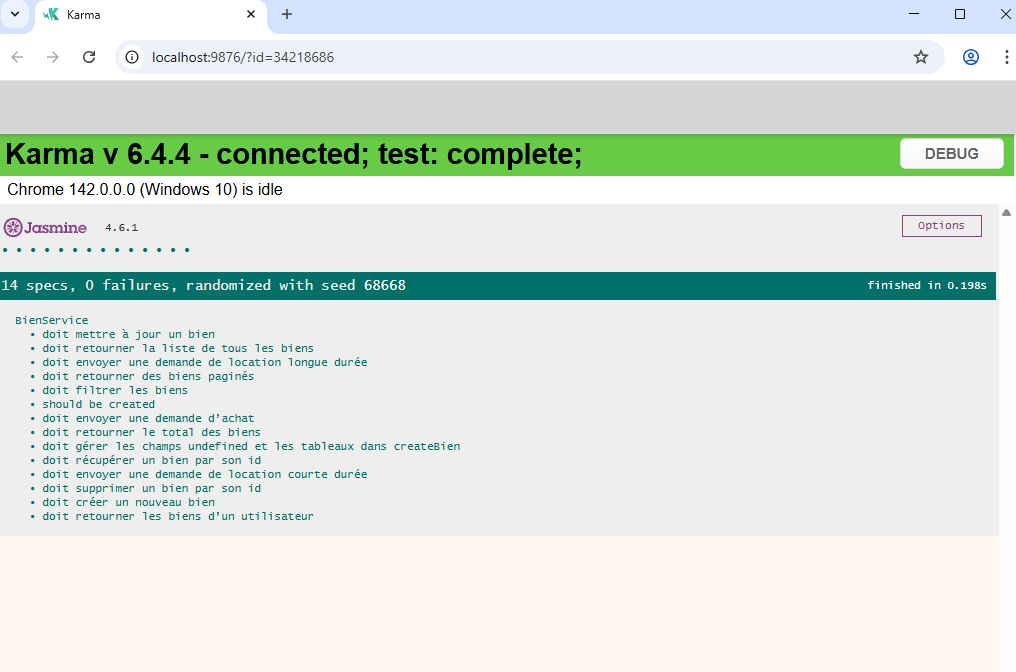
✅ Vérification du comportement du service en cas d’erreur HTTP (gestion des erreurs avec HttpTestingController).

Les tests ont permis de garantir la fiabilité du service et la cohérence des échanges avec le backend.

### ****Résultats des tests –**** BienService

Après exécution des tests unitaires avec la commande  
*ng test --include src/app/services/bien.service.spec.ts --browsers ChromeHeadless --watch=false --code-coverage*,  
tous les scénarios ont été validés avec succès.

Les résultats de couverture sont les suivants :



Ces résultats témoignent d’une excellente couverture du code.  
Le service **BienService** fonctionne de manière stable et conforme aux attentes.  
Les tests confirment que la communication avec l’API, le traitement des réponses et la gestion des erreurs sont correctement implémentés.

### ****Tests unitaires – Composant**** BienFiltreComponent

Les tests unitaires du composant **BienFiltreComponent** ont été réalisés pour vérifier la bonne gestion du système de filtres dans la recherche des biens immobiliers.  
Ce composant permet à l’utilisateur de filtrer les logements selon différents critères (type, pays, ville, etc.) et d’émettre les paramètres correspondants vers le composant parent.

Les principaux points testés :

✅ Vérification de la création correcte du composant (should create).

✅ Test de l’initialisation des listes de filtres (type, pays, ville).

✅ Vérification de la structure et du contenu de l’objet filters après interaction.

✅ Simulation du déclenchement de l’événement search.emit() avec des critères valides.

✅ Test du comportement du composant lorsque certains champs sont laissés vides.

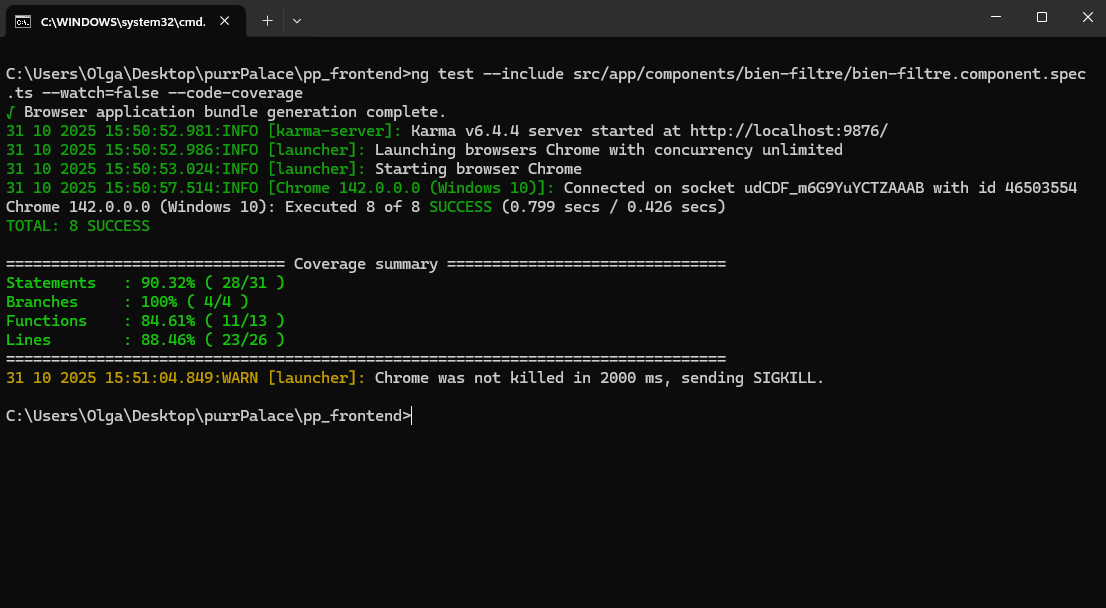
✅ Vérification que les valeurs saisies dans le formulaire sont bien transmises à la recherche.

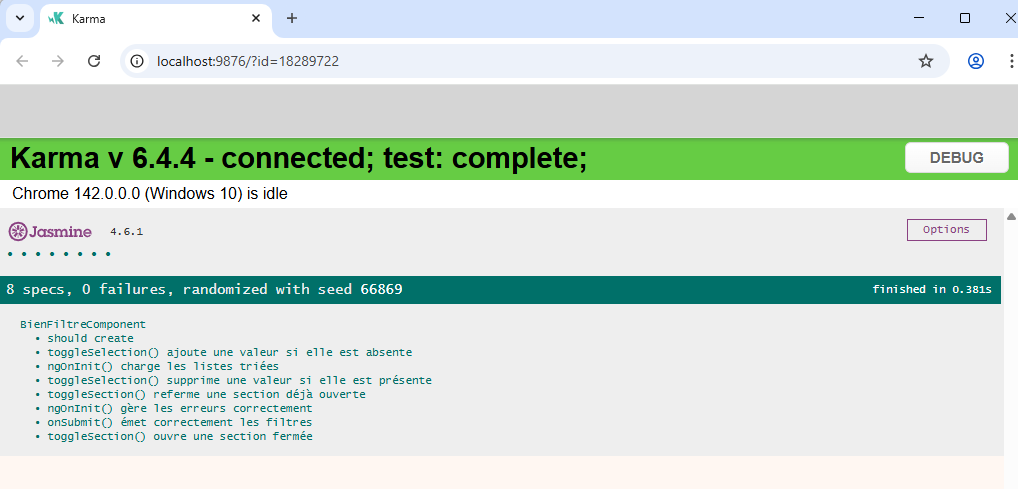
Ces tests permettent d’assurer le bon fonctionnement de la logique interne du composant et sa communication avec les autres éléments de l’application.

### ****Résultats des tests –**** BienFiltreComponent

Les tests ont été exécutés à l’aide de la commande  
*ng test --include src/app/components/bien-filtre/bien-filtre.component.spec.ts --browsers ChromeHeadless --watch=false --code-coverage*

Tous les tests se sont déroulés avec succès.  
Les résultats de couverture obtenus sont les suivants





Ces résultats démontrent une couverture de code très satisfaisante.  
Le composant **BienFiltreComponent** fonctionne de manière fiable et les interactions utilisateur (sélection, validation, émission des filtres) sont testées avec succès, garantissant une expérience fluide et cohérente dans l’application.

### Résultat des tests unitaires — Pipe Truncate

Le **pipe TruncatePipe** a pour rôle de limiter la longueur d’un texte affiché dans l’interface et d’ajouter des points de suspension (...) lorsque le texte dépasse la limite spécifiée.  
Il utilise le service Angular DomSanitizer afin de garantir la sécurité du contenu HTML affiché (prévention des injections de code).

#### Objectifs du test

Les tests unitaires ont été réalisés pour vérifier :

la bonne création et initialisation du pipe ;

le bon comportement selon différentes longueurs de texte ;

la gestion des cas limites (null, undefined, valeurs numériques, ou limite nulle) ;

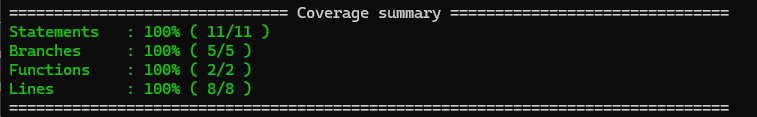
la sécurité du rendu via DomSanitizer.bypassSecurityTrustHtml.

#### Méthodologie

Les tests ont été exécutés à l’aide du framework **Jasmine/Karma** avec la commande :

*ng test --include src/app/pipes/truncate.pipe.spec.ts --watch=false --code-coverage*

#### Résultats obtenus



Tous les scénarios possibles (texte court, long, vide, numérique, ou sans limite) ont été validés avec succès.  
Le pipe gère correctement les entrées non textuelles en les convertissant en chaînes de caractères avant traitement et ne provoque plus d’erreurs de type (substr).  
La couverture complète atteste d’un **comportement stable, sécurisé et robuste**, respectant les bonnes pratiques Angular.

### Conclusion générale

Les tests unitaires réalisés sur les différentes parties de l’application confirment que le projet **PurrPalace** présente un **haut niveau de fiabilité technique et fonctionnelle**.  
Les composants et services testés se comportent conformément aux attentes, les erreurs potentielles sont correctement gérées, et les taux de couverture obtenus témoignent d’un code bien structuré et stable.

Cette phase de test valide la **qualité du développement Angular** et assure que l’application est prête à être intégrée, déployée et maintenue dans des conditions réelles, tout en respectant les standards professionnels exigés dans un projet complet **Symfony + Angular**.

## ****Conclusion et bilan du projet****

Ce projet, intitulé **PurrPalace**, a permis de mettre en œuvre l’ensemble des compétences acquises en développement web full stack à travers la création d’un site complet de gestion immobilière.  
Il combine à la fois **Symfony (backend)** pour la logique métier et la gestion des données, et **Angular (frontend)** pour l’affichage dynamique et l’interaction utilisateur.

La réalisation du projet a nécessité plusieurs étapes : conception, modélisation, développement, intégration et tests.  
Chaque phase a contribué à construire une application cohérente, fonctionnelle et fidèle au cahier de charges initial.

Sur le plan technique, la mise en place d’une API REST et la communication entre les deux environnements (HTTP/HTTPS) ont permis de comprendre les enjeux réels d’un projet full stack.  
Les fonctionnalités principales — authentification, gestion des logements, réservations, offres de vente et interface multirôle — ont été développées avec une attention particulière à la sécurité et à l’expérience utilisateur.

Sur le plan personnel, ce projet a renforcé la capacité d’organisation, d’autonomie et de résolution de problèmes techniques, notamment liés à la communication entre le front-end et le back-end.

En conclusion, **PurrPalace** est un projet abouti, alliant technique, créativité et logique de gestion, qui démontre la maîtrise des technologies modernes et la compréhension complète du cycle de développement d’une application web.