TP3 : Full Stack

Objectifs du TP

- corriger le docker-compose pour que l'application démarre correctement
- modifier la gestion des prix en centimes dans la base
- s'assurer que les horaires des boutiques n'entrent pas en conflit

- améliorer de manière générale la qualité du code
- rendre l'interface responsive
- indexer la base de données pour permettre des accès plus rapides

Docker Compose

Le docker compose fourni avec ce projet ne semble pas fonctionner correctement, comme décrit dans le `README.md`, il semble que le backend n'attende pas que la base de données ait démarrée pour se lancer, ce qui provoque des erreurs de connexion.

Pour corriger ce problème, nous ajoutons un 'healthcheck' aux containers concernés :

shop-db

```
db:
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 10
```

shop-server

```
elasticsearch:
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:9200/_cluster/health || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 30
```

Pour elasticsearch, j'ai également dû ajouter une propriété d'environnement :

```
environnement:
      ES_JAVA_OPTS: "-Xms512m -Xmx512m"
```

car elasticsearch demande beaucoup de mémoire et sans cela, le processus est tué.

Un problème peut toujours être apercu dans les logs concernant le sniffer d'elasticsearch :

```
shop-server         | java.net.ConnectException: Connection refused
shop-server         |   at java.base/sun.nio.ch.SocketChannelImpl.checkConnect(Native Met
shop-server         |   at java.base/sun.nio.ch.SocketChannelImpl.finishConnect(SocketCha
shop-server         |   at org.apache.http.impl.nio.reactor.DefaultConnectingIOReactor.pr
shop-server         |   at org.apache.http.impl.nio.reactor.DefaultConnectingIOReactor.pr
shop-server         |   at org.apache.http.impl.nio.reactor.AbstractMultiworkerIOReactor.
shop-server         |   at org.apache.http.impl.nio.conn.PoolingNHttpClientConnectionMana
shop-server         |   at org.apache.http.impl.nio.client.CloseableHttpAsyncClientBase$1
shop-server         |   at java.base/java.lang.Thread.run(Thread.java:829) ~[na:na]
shop-server         |
shop-server         | 2025-12-02 07:27:52.417 DEBUG 1 --- [pool-1-thread-1] o.a.h.i.n.c.I
shop-server         | 2025-12-02 07:27:52.421 ERROR 1 --- [nt_sniffer[T#1]] org.elasticse
shop-server         |
```

Néanmoins, pour cette partie du projet, j'ai préféré ne modifier que le `docker-compose.yml` en partant du principe que le code fonctionne, donc je n'ai pas corrigé ce problème pour l'instant, et cela n'empêche l'application de fonctionner.

Montés de versions

Technologie	Version de départ	Version ciblée
Spring Boot	2.7.5	4.0.0
JDK	11	24
React	18.2	18.3

Spring Boot

On remarque ici qu'on passe d'une version inférieure à la 3.0.0 à une version supérieur. Cette migration provoque des breaking changes qu'on va devoir fixer :

- Migration de javax → jakarta : Toutes les dépendances utilisant javax doivent être mises à jour pour utiliser jakarta à la place.
  - spring-boot-starter-validation : 4.0.0
  - postgresql : 42.7.8
  - remplacer partout dans le code les imports de javax par des imports de jakarta.
  - remplacer springfox car cette dépendance n'a pas été mise à jour.

Pour conserver la documentation, nous allons utiliser cette dépendance :

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.7.0</version>
</dependency>
```

La documentation est désormais accessible à l'URL : http://localhost:8080/swagger-ui/index.html

React

Pour monter de version sur React, on pourrait utiliser l'outil officiel `codemod` qui vient mettre à jour des aspects particulier du code source d'un projet pour le rendre compatible avec la nouvelle version après avoir installé la nouvelle version de react, idem pour typescript :

```
$ npm install --save-exact react@^19.0.0 react-dom@^19.0.0
$ yarn add --exact @types/react@^19.0.0 @types/react-dom@^19.0.0
$ npx codemod@latest react/19/migration-recipe
```

Néanmoins, la version 18 de React est plus stable, la version 19 étant très récente, j'ai donc décidé de la conserver.

Améliorations du backend

Les modifications suivantes ont été apportées au projet :

- **Validation de model** : il est désormais impossible d'ajouter des horaires incompatible pour une boutique.
- **Indexes** : des indexs ont été ajoutés aux champs représentants des clés étrangères dans chaque entité pour améliorer les performances.
- **ElasticSearch** : l'endpoint `GET /api/v1/shops` a été modifié pour inclure un paramètre optionel `label` correspondant à une recherche plein texte.
- **Indexation ElasticSearch** : un service a été ajouté pour permettre l'indexation de la base de données complète au démarrage de l'application.

Indexes

J'ai ajouté des indexes sur les clés étrangères du modèle de la manière suivante :

```
@Table(name = "products", indexes = {
        @Index(name = "idx_shop", columnList = "shop_id")
})
```

Validation de model

Pour permettre une validation pratique et réutilisable du model, j'ai ajouté une annotation `@NotOverlapping` :

```java
@Documented
@Constraint(validatedBy = OpeningHoursShopValidator.class)
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface NotOverlapping {

    String message() default "Opening hours of a shop must not overlap";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Cette annotation permet d'assurer qu'une liste d'OpeningHoursShop ne contient pas deux horaires qui se chevauchent :

```
public class OpeningHoursShopValidator implements ConstraintValidator<NotOverlapping, Lis

    @Override
    public boolean isValid(List<OpeningHoursShop> value, ConstraintValidatorContext contex
        if (value.isEmpty()) {
            return true;
        }
        List<OpeningHoursShop> openingHours = value;

        for (int i = 0; i < openingHours.size(); i++) {
            for (int j = i + 1; j < openingHours.size(); j++) {
                if (isOverlapping(openingHours.get(i), openingHours.get(j))) {
                    return false;
                }
            }
        }
        return true;
    }

    // TOOLS

    private boolean isOverlapping(OpeningHoursShop i, OpeningHoursShop j) {
        return (i.getDay() == j.getDay())
                && !(i.getCloseAt().isBefore(j.getOpenAt()) || i.getOpenAt().isAfter(j.ge
    }

}
```

Nous n'avons plus qu'à annoter le champ concerné dans l'entité `Shop` :

```
    @OneToMany(cascade = { CascadeType.ALL })
    @NotOverlapping
    private List<@Valid OpeningHoursShop> openingHours = new ArrayList<OpeningHoursShop>(
```

Il est maintenant impossible d'ajouter des horaires incompatibles pour une boutique.

Indexes

ElasticSearch

Cette partie du projet s'appuit sur les dépendances de `hibernate-search` qui permet entre autre d'implémenter de la recherche plein texte.

Pour se faire, j'ai commencé par créer un service `elasticSearchService` qui sera en charge de réaliser les opérations liées à elasticsearch. Bien sûr, ce choix est discutable car si l'on avait de nombreuses opérations à réaliser, elles seraient toutes regroupées dans ce même service ce qui ne respecterait pas les principes SOLID. Néanmoins, puisqu'il n'y a qu'une seule fonctionalité qui utilise elasticsearch pour le moment, j'ai préférée l'implémenter dans un service dédié.

Le service contient donc une méthode qui réalise la recherche à partir d'un string :

```
    public Page<Shop> searchShops(String searchQuery, Pageable pageable) {
            Session session = entityManager.unwrap(Session.class);
              SearchSession searchSession = Search.session(session);

              String sanitizedSearchQuery = searchQuery.trim();

              List<Shop> results = searchSession.search(Shop.class)
                              .where(f -> f.bool()
                                              .must(f.match()
                                                              .field("name")
                                                              .matching(sanitizedSearch
                                                              .fuzzy(2)))
                              .fetch((int) pageable.getOffset(), pageable.getPageSize()
                              .hits();

              return new PageImpl<>(results, pageable, results.size());
      }
```

Cette méthode se base sur le champ "name" des boutiques et tente de les matcher avec la recherche de l'utilisateur et renvoie un résultat paginé.

On doit maintenant mettre à jour la méthode du service :

```
    public Page<Shop> getShopList(
            Optional<String> sortBy,
            Optional<Boolean> inVacations,
            Optional<String> createdBefore,
            Optional<String> createdAfter,
            Optional<String> label,
            Pageable pageable) {
        // SORT
        if (sortBy.isPresent()) {
            return switch (sortBy.get()) {
                case "name" -> shopRepository.findByOrderByNameAsc(pageable);
                case "createdAt" -> shopRepository.findByOrderByCreatedAtAsc(pageable);
                default -> shopRepository.findByOrderByNbProductsAsc(pageable);
            };
        }

        // FILTERS
        Page<Shop> shopList = getShopListWithFilter(inVacations, createdBefore, createdAf
        if (shopList != null) {
            return shopList;
        }

        // NONE
        return shopRepository.findByOrderByIdAsc(pageable);
    }
```

Enfin, on met à jour notre controller :

```
    @GetMapping
    public ResponseEntity<Page<Shop>> getAllShops(
            Pageable pageable,
            @RequestParam(required = false) Optional<String> sortBy,
            @RequestParam(required = false) Optional<Boolean> inVacations,
            @RequestParam(required = false) Optional<String> createdAfter,
            @RequestParam(required = false) Optional<String> createdBefore,
            @RequestParam(required = false) Optional<String> label

    ) {
        return ResponseEntity.ok(
                service.getShopList(sortBy, inVacations, createdAfter, createdBefore, labe
    }
```

À ce stade, notre backend expose bien un endpoint qui permet la recherche plein texte, il faut maintenant mettre à jour le front pour ajouter cette fonctionalité.

On commence par ajouter un état :

```
const [searchLabel, setSearchLabel] = useState < string > "";
```

Une fonction qui permet de mettre l'état à jour :

```
const handleChangeSearchLabel = (event: ChangeEvent<HTMLInputElement>) => {
        setSearchLabel(event.target.value);
};
```

Un champ d'input :

```
<FormControl>
        <TextField
                id="filled-basic"
                label="Rechercher"
                variant="outlined"
                value={searchLabel}
                onChange={handleChangeSearchLabel}
        />
</FormControl>
```

Enfin, on met à jour la fonction du service :

```
// HOME
const completedFilters = filters + searchLabel ? "&label=" + searchLabel : "";
promisedShops = ShopService.getShopsFiltered(pageSelected, 9, completedFilters);
// SERVICE
export function getShopsFiltered(
        page: number,
        size: number,
        urlFilters: string
): Promise<ResponseArray<Shop>> {
        return axios.get(
                `${process.env.REACT_APP_API}/shops?page=${page}&size=${size}${urlFilters}
        );
}
```

À partir de maintenant, il est possible de réaliser de la recherche plein-texte qui est compatible avec les autres filtres car le label recherché par l'utilisateur est lui-même traité comme un filtre. Il n'est cependant pas si simple de combiner le tri et la recherche plein texte car l'endpoint de tri ne prend pas en compte les filtres.

Pour corriger ca, on va modifier la méthode du service :

```
public Page<Shop> getShopList(
        Optional<String> sortBy,
        Optional<Boolean> inVacations,
        Optional<String> createdBefore,
        Optional<String> createdAfter,
        Optional<String> label,
        Pageable pageable) {

    if (label.isPresent()) {
        Pageable searchPageable = pageable;
        if (sortBy.isPresent()) {
            String[] sortParts = sortBy.get().split(",");
            String field = sortParts[0];
            Sort.Direction direction = sortParts.length > 1 && sortParts[1].equalsIgn
                    ? Sort.Direction.DESC
                    : Sort.Direction.ASC;
            searchPageable = PageRequest.of(
                    pageable.getPageNumber(),
                    pageable.getPageSize(),
                    Sort.by(direction, field));
        }
        Page<Shop> elasticResults = elasticSearchService.searchShops(label.get(), sea
        System.out.println("Elastic Results : " + elasticResults.getContent().size() +

        List<Shop> filteredResults = elasticResults.getContent().stream()
                .filter(shop -> inVacations.isEmpty() || shop.getInVacations() == (inV
                .filter(shop -> createdAfter.isEmpty() ||
                        !shop.getCreatedAt().isBefore(LocalDate.parse(createdAfter.get
                .filter(shop -> createdBefore.isEmpty() ||
                        !shop.getCreatedAt().isAfter(LocalDate.parse(createdBefore.get
                .toList();

        return new PageImpl<>(filteredResults, pageable, elasticResults.getTotalElemen
    }

    Specification<Shop> spec = (root, query, cb) -> {
        List<Predicate> predicates = new ArrayList<>();

        if (inVacations.isPresent()) {
```

```
                predicates.add(inVacations.get()
                        ? cb.isTrue(root.get("inVacations"))
                        : cb.isFalse(root.get("inVacations")));
            }

            if (createdBefore.isPresent()) {
                LocalDate ld = LocalDate.parse(createdBefore.get());
                predicates.add(cb.lessThanOrEqualTo(root.get("createdAt"), ld));
            }

            if (createdAfter.isPresent()) {
                LocalDate ld = LocalDate.parse(createdAfter.get());
                predicates.add(cb.greaterThanOrEqualTo(root.get("createdAt"), ld));
            }

            return cb.and(predicates.toArray(new Predicate[0]));
        };

        Pageable finalPageable = pageable;
        if (sortBy.isPresent()) {
            String[] sortParts = sortBy.get().split(",");
            String field = sortParts[0];
            Sort.Direction direction = sortParts.length > 1 && sortParts[1].equalsIgnoreCa
                    ? Sort.Direction.DESC
                    : Sort.Direction.ASC;
            finalPageable = PageRequest.of(
                    pageable.getPageNumber(),
                    pageable.getPageSize(),
                    Sort.by(direction, field));
        }

        return shopRepository.findAll(spec, finalPageable);
    }
```

Puis, on prend ce changement en compte côté front en modifiant la fonction getShops() :

```
const getShops = () => {
        setLoading(true);

        let combinedParams = "";

        if (sort) {
                combinedParams += `&sortBy=${sort}`;
        }

        if (searchLabel) {
                combinedParams += `&label=${searchLabel}`;
        }

        if (filters) {
                combinedParams += filters;
        }

        if (combinedParams.startsWith("&")) {
                combinedParams = combinedParams.substring(1);
        }

        const promisedShops = combinedParams
                ? ShopService.getShopsFiltered(pageSelected, 9, "&" + combinedParams)
                : ShopService.getShops(pageSelected, 9);

        promisedShops
                .then((res) => {
                        setShops(res.data.content);
                        setCount(res.data.totalPages);
                        setPage(res.data.pageable.pageNumber + 1);
                })
                .finally(() => setLoading(false));
};
```

On peut maintenant combiner tri, filtres et recherche plein texte.

Indexation ElasticSearch

Pour pouvoir indexer les éléments déjà présents en base, j'ai ajouté une méthode dans le service

elasticsearch :

```
        @Transactional
        public void reindexAll() throws InterruptedException {
                Session session = entityManager.unwrap(Session.class);
                SearchSession searchSession = Search.session(session);

                MassIndexer indexer = searchSession.massIndexer(Shop.class)
                                .threadsToLoadObjects(5)
                                .batchSizeToLoadObjects(25)
                                .idFetchSize(150);

                indexer.startAndWait();
        }
```

pour appeler cette méthode, on a un command line runner :

```java
@Component
public class StartupIndexer implements ApplicationRunner {

    // ATTRIBUTES

    private final ElasticSearchService indexService;
    @Value("${elasticsearch.reindex-on-startup}")
    private boolean shouldReindex;

    // CONSTRUCTOR

    public StartupIndexer(ElasticSearchService indexService) {
        this.indexService = indexService;
    }

    // METHODS

    @Override
    public void run(ApplicationArguments args) throws Exception {
        if (shouldReindex) {
            indexService.reindexAll();
        }
    }

}
```

Enfin, comme on peut le voir dans ce code, il y a une propriété dans `application.properties` qui permet d'indiquer si l'on souhaite ré-indexer les entités présentes dans la base de données ou non.

Montée de version du Frontend

L'application est actuellement dans la version 18.2 de React, qui est toujours supportée. La version 18 de React est la plus stable donc on va la garder, mais on va monter vers la version de React 18 la plus récente :

```
npm install react@^18.3.0 react-dom@^18.3.0
```

Les composants eux utilisent MUIv5.10.14, on va donc les mettres à jour :

```
npm install @mui/material@^6.0.0 @mui/icons-material@^6.0.0 @mui/x-date-pickers@^7.0.0
```

Il y a quelques changements cassants dans cette version de mui, notamment concernant les date pickers que nous allons devoir mettre à jour. Voici les correctifs à appliquer aux components DatePicker et TimePicker :

```
value={dayjs(`2014-08-18T${openingHour.openAt}`)} // utiliser dayjs pour les valeurs
slotProps={{
    textField: {
        fullWidth: true, // utiliser slot props à la place de renderInput
    },
}}
format="DD/MM/YYYY" // utiliser format à la place de inputFormat
```

Gestion des erreurs

À l'heure actuelle, on remarque que lorsqu'on lance le frontend mais que le backend n'est pas démarré, des erreurs typescript apparaissent, ce qui indique une mauvaise gestion des erreurs. Étant donné que cette gestion des erreurs était demandée au TP précédent et que cela sera également plus pratique pour la suite de ce TP, je vais ajouter une gestion des erreurs adaptée à axios.

Pour commencer, on définit notre propre instance de axios :

```
import axios, { AxiosError, AxiosResponse } from "axios";

const axiosInstance = axios.create({
        baseURL: process.env.REACT_APP_API,
        timeout: 10000,
});

axiosInstance.interceptors.response.use(
        (response: AxiosResponse) => response,
        (error: AxiosError) => {
                const customError = {
                        message: "An error occurred",
                        status: error.response?.status,
                        originalError: error,
                };

                if (error.code === "ECONNABORTED") {
                        customError.message = "Request timeout - server is not responding"
                } else if (error.code === "ERR_NETWORK" || !error.response) {
                        customError.message = "Unable to connect to the server";
                } else {
                        switch (error.response?.status) {
                                case 400:
                                        customError.message = "Invalid request";
                                        break;
                                case 404:
                                        customError.message = "Resource not found";
                                        break;
                                case 500:
                                        customError.message = "Server error";
                                        break;
                                default:
                                        customError.message = `Error: ${
                                                error.response?.statusText || "Unknown er
                                        }`;
                        }
                }

                return Promise.reject(customError);
```

```
        }
);

export default axiosInstance;
```

On va maintenant utiliser cette instance d'axios dans tous le services. Ensuite, on met à jour les pages pour qu'elles incluent un state "error", qu'on mettra à jour au début de chaque requête, puis lorsque la requête renverra une erreur, on conservera le message d'erreur et on affichera une notification :

```
            .catch((err) => {
                setError(err.message || 'Unable to find shops');
                setShops(null);
            })
            ...
                        {error && (
                <Alert severity="error" sx={{ width: '100%' }}>
                    {error}
                </Alert>
            )}
```

Rendre le frontend responsive

Comme on peut l'observer, l'interface ne se comporte pas bien sur les tailles d'écrans plus petites.

Critères

On considerera que l'application est responsive si :

- elle se comporte correctement sur des appareils de tailles 375x812 (Iphone 11 Pro) et plus, les appareils plus petits ne sont pas pris en charge.
- on dira que l'application se comporte correctement dans la mesure où toutes les fonctionalités sont accessibles et qu'aucun bug visuel n'est remarqué.

Enfin, pour rester au plus proche du design original et en modifiant le moins possible le code, nous rendrons l'interface responsive uniquement en modifiant les paramètres des components

MUI utilisées.

Modifications apportées

La première des choses à mettre à jour est le layout de chaque page comprenant la barre de navigation. Malheuresement, il n'est pas possible de rendre cette navbar responsive en ajustant simplement les tailles, car il faut que chaque onglet reste accessible, mais il est impossible de tous les afficher les uns à côté des autres tout en gardant les labels lisibles, on va donc implémenter un menu drawer qui permettra d'afficher le menu sur le côté de la page lorsqu'on clique sur le menu burger situé en haut à gauche de la navbar.

```
const Layout = ({ children }: Props) => {
        const navigate = useNavigate();
        const theme = useTheme();
        const isMobile = useMediaQuery(theme.breakpoints.down("md"));
        const [mobileOpen, setMobileOpen] = useState(false);

        const handleDrawerToggle = () => {
                setMobileOpen(!mobileOpen);
        };

        const handleNavigate = (path: string) => {
                navigate(path);
                setMobileOpen(false);
        };

        const drawer = (
                <Box onClick={handleDrawerToggle} sx={{ textAlign: "center" }}>
                        <Typography variant="h6" sx={{ my: 2 }}>
                                Gestion de boutiques
                        </Typography>
                        <List>
                                {navItems.map((item) => (
                                        <ListItem key={item.label} disablePadding>
                                                <ListItemButton
                                                        sx={{ textAlign: "center" }}
                                                        onClick={() => handleNavigate(ite
                                                >
                                                        <ListItemText primary={item.label
                                                </ListItemButton>
                                        </ListItem>
                                ))}
                        </List>
                </Box>
        );

        return (
                <Box
                        sx={{
                                display: "flex",
```

```
                        flexDirection: "column",
                        minHeight: "100vh",
                }}
        >
                <AppBar component="nav" position="fixed">
                        <Toolbar>
                                {isMobile && (
                                        <IconButton
                                                color="inherit"
                                                aria-label="open drawer"
                                                edge="start"
                                                onClick={handleDrawerToggle}
                                                sx={{ mr: 2 }}
                                        >
                                                <MenuIcon />
                                        </IconButton>
                                )}

                                <Typography
                                        variant="h6"
                                        onClick={() => navigate("/")}
                                        sx={{
                                                cursor: "pointer",
                                                flexGrow: isMobile ? 1 : 0,
                                                mr: isMobile ? 0 : 4,
                                        }}
                                >
                                        Gestion de boutiques
                                </Typography>

                                {!isMobile && (
                                        <>
                                                <Box sx={{ flexGrow: 1, display: '
                                                        {navItems.map((item) => (
                                                                <Button
                                                                        key={item
                                                                        sx={{ colo
                                                                        onClick={
                                                                >
                                                                        {item.labe
```

```
                                                                </Button>
                                                        ))}
                                                </Box>
                                                <Box>
                                                        <SwitchLanguage />
                                                </Box>
                                        </>
                                )}

                                {isMobile && (
                                        <Box>
                                                <SwitchLanguage />
                                        </Box>
                                )}
                        </Toolbar>
                </AppBar>

                <Drawer
                        variant="temporary"
                        open={mobileOpen}
                        onClose={handleDrawerToggle}
                        sx={{
                                display: { xs: "block", md: "none" },
                                "& .MuiDrawer-paper": {
                                        boxSizing: "border-box",
                                        width: 240,
                                },
                        }}
                >
                        {drawer}
                </Drawer>

                <Loader />

                <Container
                        component="main"
                        maxWidth="xl"
                        sx={{
                                mt: { xs: 10, sm: 12 },
                                mb: 4,
```

```
                                        px: { xs: 2, sm: 3, md: 4 },
                                        flexGrow: 1,
                                    }}
                            >
                                    {children}
                            </Container>
                    </Box>
            );
};
```

La taille des titres des pages semblent trop grande, pour remédier à celà on ajoute :

```
                sx={{
                    typography: { xs: 'h4', sm: 'h3', md: 'h2' },
                }}
```

dans les paramètres des components `Typography`.

Enfin, les cartes pour les boutiques, produits et catégories ainsi que leurs containers ont besoin d'ajustements :

En règle générale, nous allons ajouter une gestion de la taille des polices responsive :

```
                    fontSize: { xs: '1.75rem', sm: '2.5rem', md: '3rem' },
```

et nous allons ajouter des paramètres à nos components cartes (boutique, produit et catégorie) :

```
            sx={{
                display: 'flex',
                flexDirection: 'column',
                height: '100%',
                transition: 'transform 0.2s, box-shadow 0.2s',
                '&:hover': {
                    transform: 'translateY(-4px)',
                    boxShadow: 4,
                },
                cursor: 'pointer',
            }}
```

et pour les containers :

```
<Grid container alignItems="center" spacing={4} columns={{ xs: 4, sm: 8, md: 1
```

On doit ajuster le padding et le margin de la plupart des components, on modifie aussi certains components qui utilisent des Dividers afin que le texte s'affiche sur plusieurs lignes si nécéssaires :

```
sx={{
    '& .MuiDivider-wrapper': {
        whiteSpace: 'normal',
        textAlign: 'center',
        px: 1
    }
}}
```

Les noms des catégories peuvent parfois être très long et il est impossible de les afficher en entier sur de petits écrans sans changer drastiquement le layout, j'ai donc opté pour des ellipses masquant la fin d'un nom qui dépasserait :

```
sx={{
    textAlign: 'center',
    overflow: 'hidden',
    textOverflow: 'ellipsis',
    whiteSpace: 'nowrap',
    width: '100%',
    fontSize: { xs: '1.2rem', sm: '1.5rem' },
}}
```

Avec toutes ces modifications, l'application devrait être fonctionnelle sur les appareils mentionnés précédemment.

Améliorations possibles

Backend

Les plus grandes améliorations que l'on pourrait apporter concernent essentiellement la recherche plein texte avec elasticsearch. En effet, la méthode de recherche donne des résultats parfois peu pertinents, il suffirait de modifier la méthode findShops dans ElasticsearchService pour avoir de meilleurs résultats. La manière dont j'ai choisi d'indexer les entités au démarrage fonctionne, mais pourrait être trop coûteuse si le nombre d'entité à indéxer est important, néanmoins cela n'a d'impact que lors du démarrage, donc si nous n'avons pas à faire cette migration trop souvent le coût reste acceptable.

Frontend

Le frontend a subit quelques modifications visant à le rendre responsive, la plus grande étant la barre de navigation qui devient très différente. J'ai estimé que cette différence était nécéssaire pour la rendre responsive, mais ce choix s'éloigne de ma philosophie de migration consistant à garder l'application la plus semblable à l'originale tout en remplissant le cahier des charges.