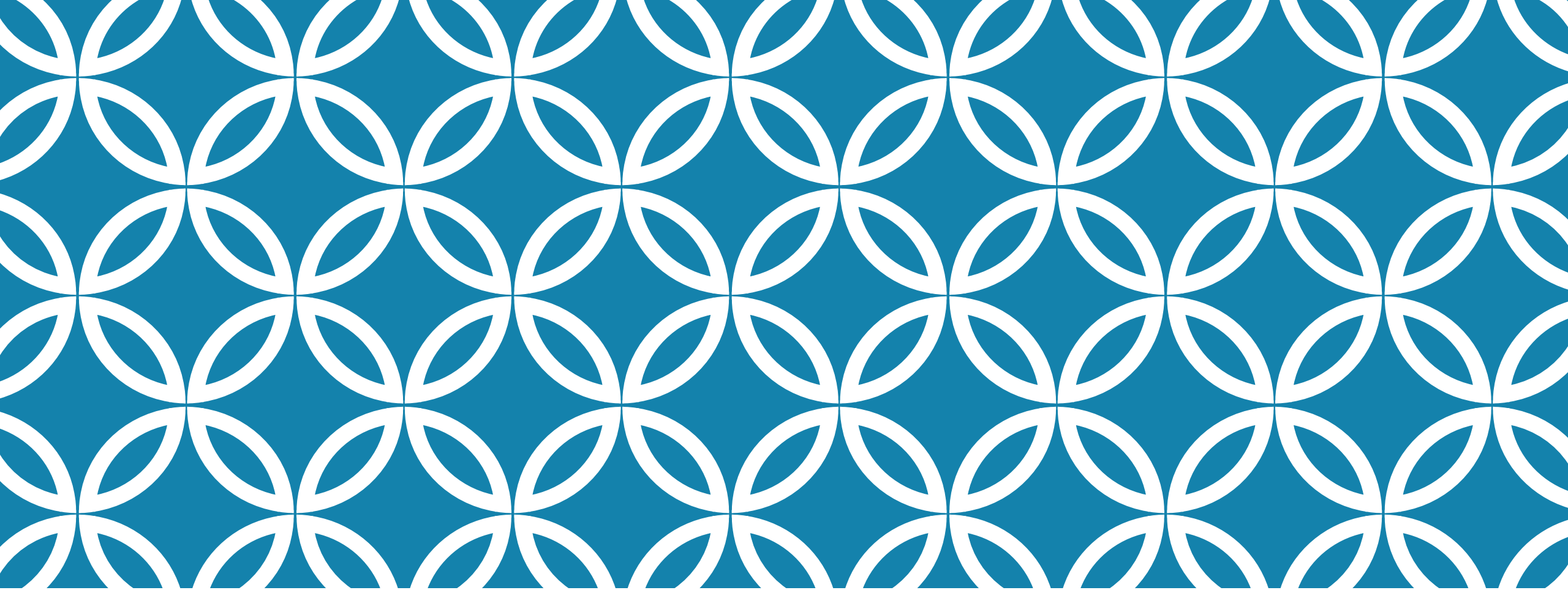
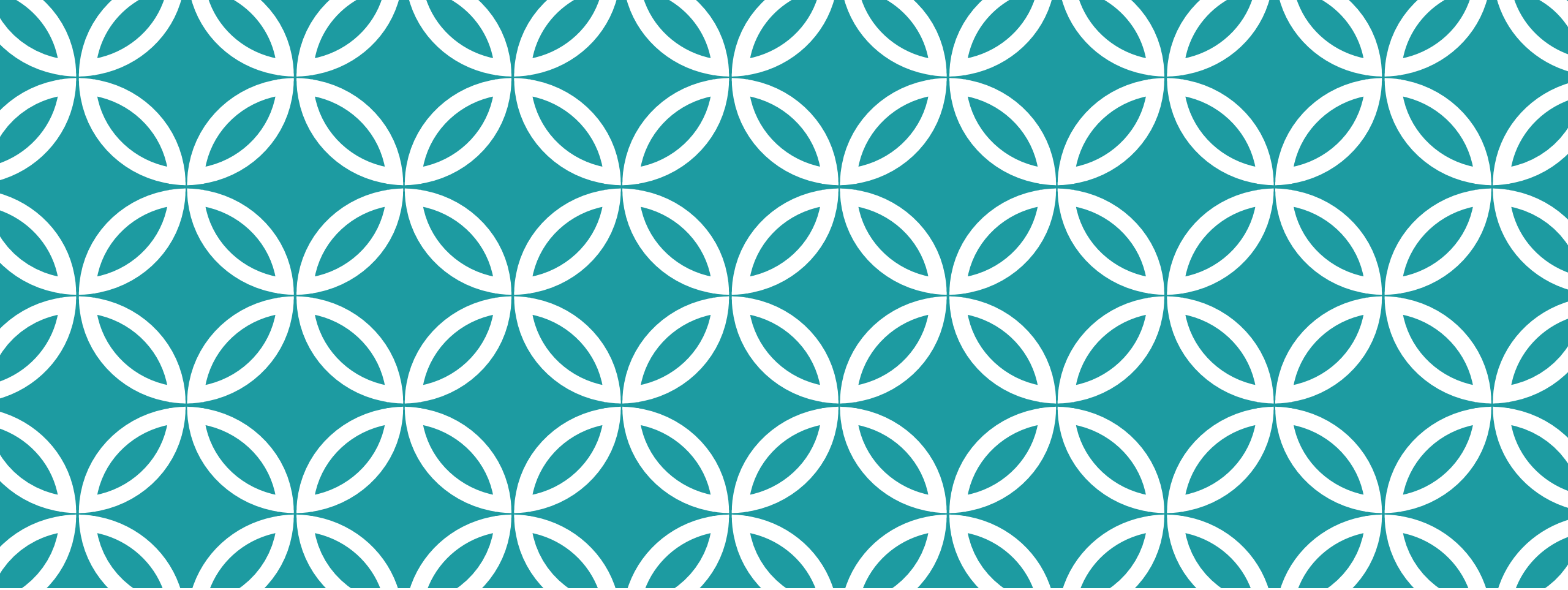


ANGULAR 4

Jérémy PERROUAULT



LES PIPES

Transformation des données avec Angular

PIPES

Un pipe (ou filtre) est une fonctionnalité front

- Intégré à un module
- Réutilisable

Permet d'afficher une donnée transformée, formatée

Certains sont prévus par Angular

- Transformer une date en date « Jour n mois année »
- Transformer un chiffre en monnaie (euros, dollars, …)
- Mettre tout en majuscule / minuscule

PIPES

Les pipes sont à utiliser dans les templates

- Il faut utiliser le caractère pipe « | » pour appliquer un filtre

```
<p>{{ produit.nom | uppercase }}</p>
```

- On peut ajouter des paramètres en utilisant les deux points « : »

```
<p>{{ produit.prix | currency:'EUR' }}</p>
```

- Il est possible d'en enchaîner plusieurs à la suite : chaining pipes
  - La transformation s'applique dans l'ordre de lecture (gauche vers la droite)

```
<p>{{ produit.dateAchat | date:"EEEE dd/MM/yyyy" | uppercase }}</p>
```

PIPES

On peut aussi créer son propre Pipe ; dans le fichier *nom-filtre.pipe.ts*

- On a besoin de l'annotation @**Pipe**, donc on l'importe

```
import { Pipe } from '@angular/core';
```

- On déclare le filtre, avec une classe, en l'annotant de @**Pipe**

```
@Pipe({
  name: 'nomFiltre'
})
export class NomFiltrePipe { }
```

Nom du pipe à utiliser dans les templates

```
<p>{{ produit.prix | nomFiltre }}</p>
```

PIPES

Dans la classe du pipe

- Il faut une méthode transform, qui attend au moins un argument
  - La donnée qui sera utilisée pour le formatage
- Selon les besoins, on peut y ajouter plusieurs autres arguments
  - Qui seront les paramètres du pipe

```
export class NomFiltrePipe {
  transform(prix: number, arg0: string): string {
    return "valeur";
  }
}
```

```
<p>{{ produit.prix | nomFiltre:"valeur de argument 1" }}</p>
```

PIPES

Chaque pipe doit être dans un module, il faut donc

- Importer ce pipe dans le module
- Le déclarer dans la liste des déclarations

EXERCICE

Créer un pipe **PrixCategoryPipe** qui permet

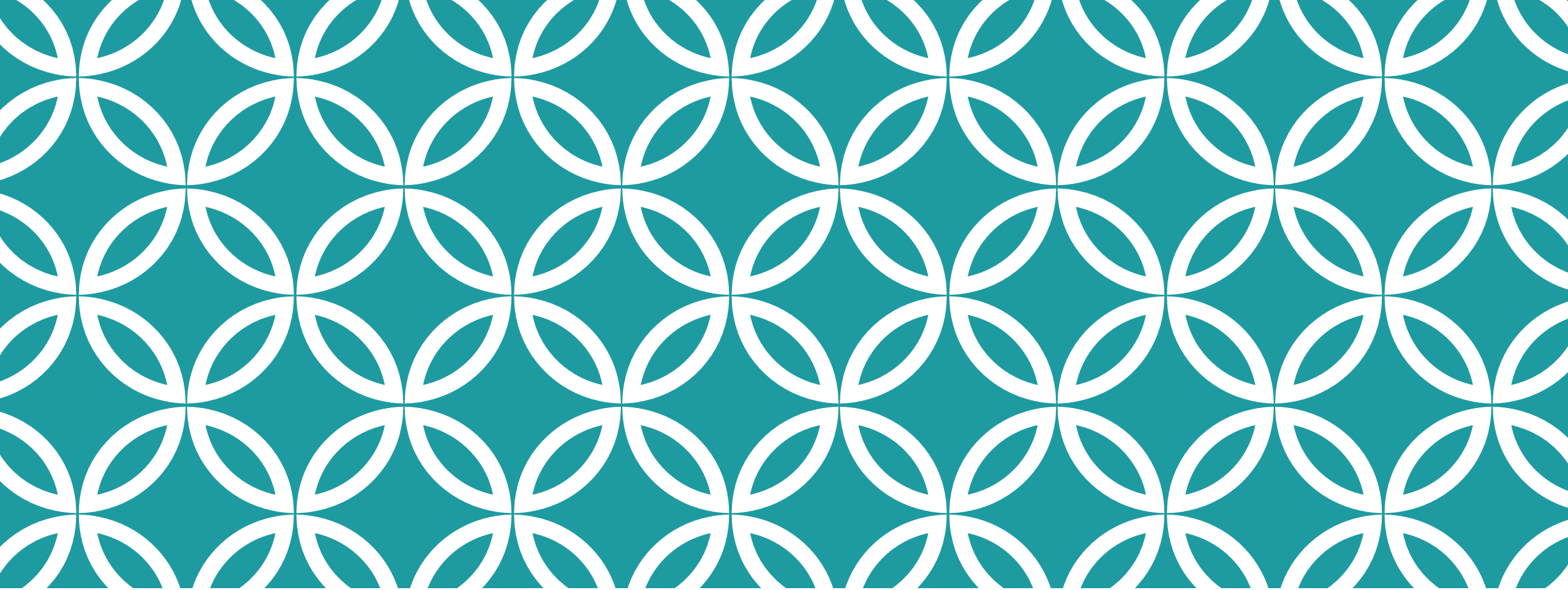
- D'afficher une couleur en correspondance avec cette catégorie

Prix < 0	Prix négatif	rouge
Prix > 0	Prix positif	vert
Prix == 0	Prix neutre	bleu

- Si le paramètre reçu est "string", alors c'est le nom de la catégorie qui est retourné
- Sinon, c'est la couleur

- Dans le template, le nom de la catégorie doit s'afficher de la bonne couleur

```
<p [style.color]="produit.prix | prixCategory">{{ produit.prix | prixCategory:"string" }}</p>
```



LES SERVICES

Distribuer des services avec Angular

SERVICES

Un service fourni (c'est un fournisseur) une fonctionnalité back-end

- Intégré à un module ou à un composant
- Réutilisable

Un service permet de créer un objet TypeScript ordinaire

- Qui vont fournir un ensemble de tâches (fonctionnalités) back-end
- La création est encapsulée et isolée du reste du code
- Eviter de définir un objet n'importe où dans TypeScript …
- Eviter la technique « un truc général qui englobe tout et accessible par tous »

Se base sur l'injection de dépendance par le type

- Le service est réutilisable en transférant son type en argument d'un constructeur
  - C'est le cas de **ElementRef** et **ActivatedRoute** par exemple

SERVICES

Dans le fichier *nom-service.service.ts*

- On a besoin de l'annotation @**Injectable**, donc on l'importe

```
import { Injectable } from '@angular/core';
```

- On déclare le service, avec une classe, en l'annotant de @**Injectable**

```
@Injectable()
export class NomServiceService { }
```

- On y ajoute tous les attributs et méthodes dont le service a besoin pour faire son travail

SERVICES

Pour injecter ce service (dans un composant par exemple)

- On importe le service
- On modifie le constructeur de ce composant en y incluant le service en argument

```
export class UnComposantComponent {
  constructor(private service: NomServiceService) { }
}
```

SERVICES

Chaque service doit être dans un module (ou un composant), il faut donc

- Importer ce service dans le module (ou le composant)
- Le déclarer dans la liste des providers

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { NomServiceService } from './nom-service.service';


@NgModule({
  imports: [ BrowserModule ],
  providers: [ NomServiceService ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

```
import { Component } from '@angular/core';
import { NomServiceService } from './nom-service.service';

@Component({
  selector: 'eshop-app',
  templateUrl: 'app/app.component.html',
  providers: [ NomServiceService ]
})
export class AppComponent {
  constructor(private service: NomServiceService) { }
}
```

EXERCICE

Créer un service **AppConfigService** qui incluera la configuration suivante

- URL de l'API de eshop

- *Pour le moment, ce service ne sera pas utilisé*

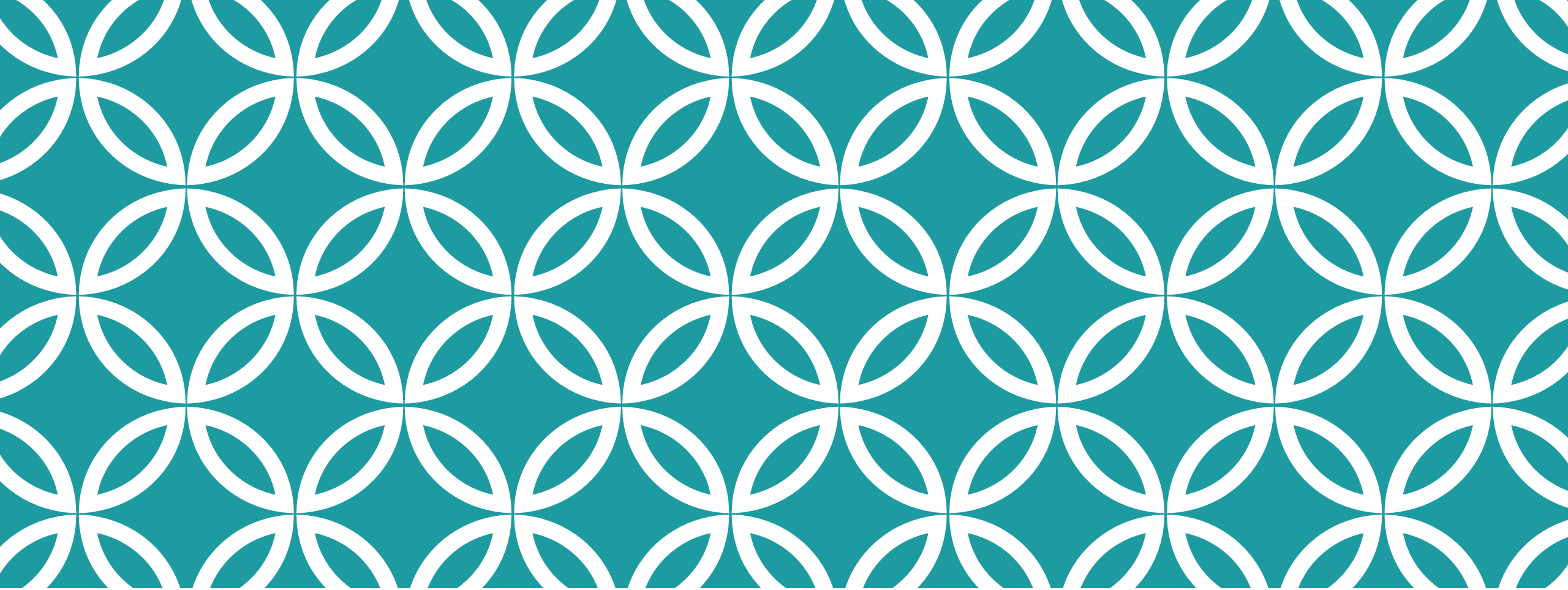
EXERCICE

Créer un service **ProduitService** qui permet

- De gérer une liste de produits
- De retourner tous les produits (méthode findAll())
- De retourner des produits par leur nom (méthode findAllByNom())
- De retourner un produit avec son id (méthode findById(id))
- D'ajouter un produit à sa liste (méthode save(produit))
- De modifier un produit de sa liste (méthode save(produit))
- De supprimer un produit de sa liste (méthode delete(produit))

Modifier les composants pour utiliser ce service (récupérer l'instance du service)

- **ProduitComponent**
- **ProduitDetailComponent**



LE MODULE HTTP

Communiquer avec un WebService depuis Angular

MODULE HTTPCLIENT

A l'instar de $.ajax (en *jQuery*), ou de *fetch*, **HttpClient** nous permet d'interroger une ressource

- En précisant la commande HTTP
- En ajoutant des données dans le corps de la requête (*body*)

Pour l'utiliser

- Il faut déclarer l'utilisation du module **HttpClientModule** (dans le module principal)
- Importer et injecter **HttpClient** là où on a besoin de l'utiliser (dans le service par exemple)

```
import { HttpClientModule } from '@angular/common/http';
import { HttpClient } from '@angular/common/http';
```

MODULE HTTPCLIENT

Disponible depuis la version 4.3 de Angular

- C’est une évolution de **Http**
- Il convertit automatiquement en JSON si nécessaire
- Il retourne un **Observable**
- Pour le reste, il s’utilise de la même façon que **Http** (get, post, put, delete, patch)

MODULE HTTPCLIENT

HttpClient met à disposition ces méthodes

Nom de la méthode	Paramètres
get	url, options?
post	url, body, options?
put	url, body, options?
patch	url, body, options?
delete	url, options?

```
http.get("http://localhost:8080/api/produit");
http.post("http://localhost:8080/api/produit", produit);
http.put("http://localhost:8080/api/produit/1", {
  nom: "GoPRO HERO 8",
  prix: 420
});
http.delete("http://localhost:8080/api/produit/1");
```

- Le nom des méthodes correspond aux commandes HTTP

MODULE HTTPCLIENT

Chacune des méthodes retournent un objet de type **Observable**

- Puisque chaque appel au service web est asynchrone !
- On va écouter la réception d'une réponse en s'inscrivant à l'**Observable**
  - Avec la méthode *subscribe* (Si cette méthode n'est pas appelée, l'appel **HTTP** ne se fait pas !)

```
http.get('http://localhost:8080/api/produit')
    .subscribe(actionAuSuccès, actionSiErreur);
```

- Dans le service, on a deux possibilités
  - Gérer une liste de produits en interne (on passera toujours par le service pour l'affichage des produits)
  - Laisser ceux qui consomment le service gérer leur propre liste de produits en interne

MODULE HTTPCLIENT

La réponse reçue est traitée en JSON

- En revanche pour être tout à fait précis, il faut préciser la nature du flux reçu

```
@Injectable()
export class ProduitService {
  private produits: Array<Produit>;

  constructor(private http: HttpClient) { }

  public findAll() {
    this.http
      .get<Array<Produit>>('http://localhost:8080/api/produit')
      .subscribe(resp =>
        this.produits = resp
      );
  }
}
```

MODULE HTTPCLIENT

La réponse envoyée l'est en JSON

- En revanche pour être tout à fait précis, il faut préciser la nature du flux reçu

```
@Injectable()
export class ProduitService {
  private produits: Array<Produit>;

  constructor(private http: HttpClient) { }

  public save(produit: Produit) {
    this.http
      .post<Produit>('http://localhost:8080/api/produit', produit)
      .subscribe();
  }
}
```

MODULE HTTPCLIENT

Le template affiche le liste des produits

```
<ul>
  <li *ngFor="let p of produits">{{ p.nom }}</li>
</ul>
```

- Le pipe *async*
  - Si on manipule un Promise, on utilisera ce pipe

```
<ul>
  <li *ngFor="let p of produits | async">{{ p.nom }}</li>
</ul>
```

EXERCICE

Utiliser l'API pour chercher la liste des produits

- Ajouter l'annotation @**CrossOrigin**("*") sur le contrôleur **ProduitRestController**
- Désactiver la sécurité (autoriser tout le monde sur */api/***)
  - La sécurité sera de nouveau implémentée lorsque le service sera prêt

Modifier le service **ProduitService**, en allant au plus simple dans un premier temps

- Utiliser **HttpClient**
- Ecrire l'adresse de l'API dans le code-source de **ProduitService**
- Charger la liste pendant la construction de **ProduitService**

EXERCICE

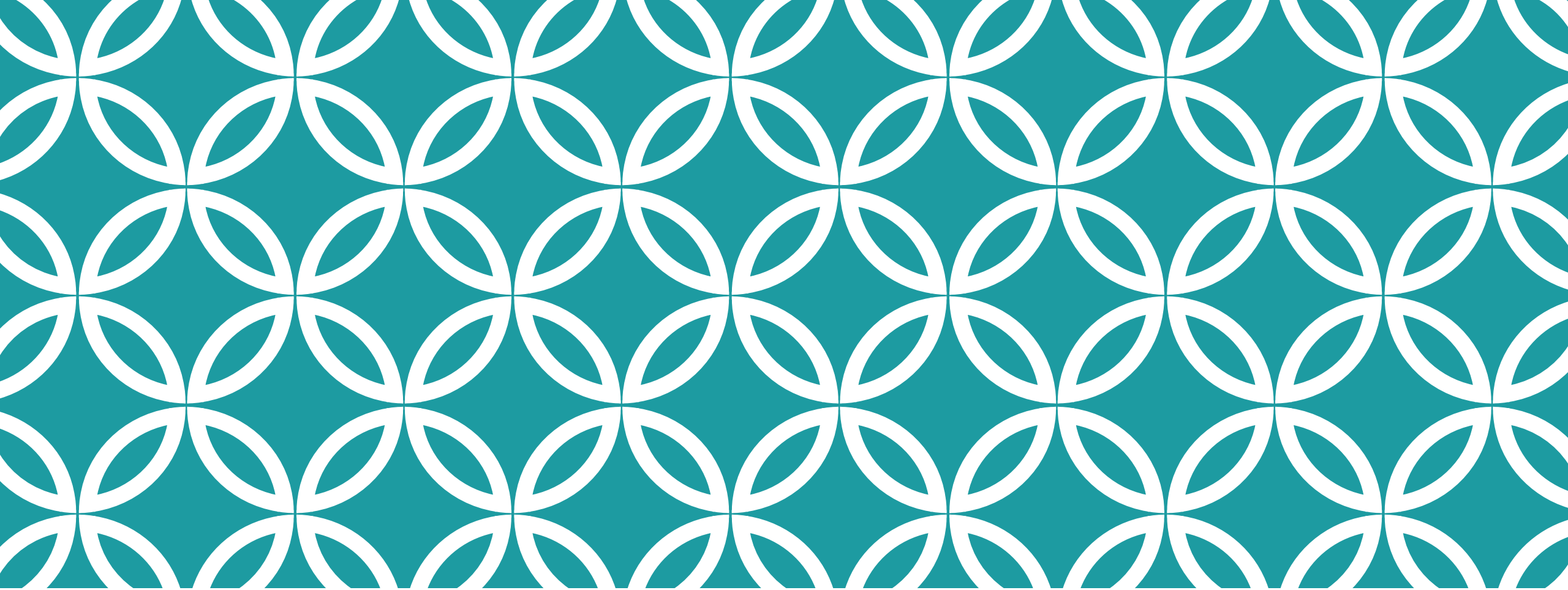
Modifier le service **ProduitService**

- Il utilise maintenant **HttpClient** et le service **AppConfigService** (pour récupérer l'adresse de base de l'API)
- La méthode *findAll* appelle le service, et retourne tous les produits
  - La méthode appelle le service seulement si la liste des produits est ***null*** !

EXERCICE

Modifier le service **ProduitService**

- Implémenter les méthodes restantes pour le CRUD
  - La méthode *findAll* appelle le service qui retourne tous les produits
  - La méthode *findAllByNom* filtre les produits par leur nom
  - La méthode *findById* appelle le service qui retourne un produit par son identifiant
    - Utiliser un Promise pour ce cas
  - La méthode *save* appelle le service qui ajoute ou sauvegarde un produit selon si l'identifiant est présent ou non
  - La méthode *delete* appelle le service qui supprime un produit



LE MODULE HTTP – SECURITÉ

Spring Sécurité

MODULE HTTP – SÉCURITÉ

Remettre en place la sécurité Spring Security

- D’autres solutions existent, comme *JWT*

MODULE HTTP – SÉCURITÉ

Côté Spring, quelques modifications de la configuration Spring Security

- Autoriser toutes les requêtes *OPTIONS*
- Activer l'authentification via *HttpBasic*
- Puisqu'il y aura une authentification par formulaire et une authentification par *HttpBasic*
  - Il faudra une classe **SecurityConfig** qui englobe deux classes de configuration de sécurité internes
    - L'une gèrera les requêtes provenant de /api/**
    - L'autre gèrera les autres requêtes
    - (Penser à ajouter une priorisation avec l'annotation @**Order**)

MODULE HTTP – SÉCURITÉ

Côté Angular

- On transmet les informations d'identification via les options de **HttpClient**
  - On a besoin de la classe **HttpHeaders**, du package @angular/common/http
  - Dans l'en-tête HTTP, il faut ajouter la clé *Authorization*, qui sera égale à "Basic " + "username:password" encodé en base64

```
let myHeaders: HttpHeaders = new HttpHeaders();
myHeaders = myHeaders.append('Authorization', 'Basic ' + btoa('jeremy:123456'));
let myOptions: Object = { headers: myHeaders };

http.get("http://localhost:8080/api/produits", myOptions);
```

  - Dans l'exemple ci-dessus, la valeur de Authorization sera Basic amVyZW15OjEyMzQ1Ng==

EXERCICE

Réimplémenter la sécurité avec Spring Security