

Détection et remplacement variables



J'ai un travaille à faire dans un projet angular 19.

Je gère les modèles de mail dans un projet web addin.

Lors de l'insertion du modèle de mail au moment où l'utilisateur clique sur insérer. Je veux pouvoir détecter des variables présentes dans le modèle.

Pour tout ce qui est \$client, appeler lapi client.

Pour banque, lapi banque et pour établissement, lapi établissement. Le résultat de ces appels sera utilisé pour valoriser les variables

Propose moi une solution

Voici une solution complète et structurée pour gérer la détection et le remplacement dynamique des variables dans un modèle d'email dans ton projet Angular 19 (Web Add-in Outlook), avec appels API conditionnels selon les variables détectées :

1. Structure du modèle

Supposons que les modèles contiennent des variables sous forme de placeholders comme :

txt

Bonjour \$client,
Votre demande a été envoyée à la \$banque pour traitement par \$etablissement.

2. Étapes de traitement

Étape 1 : Détection des variables

Utilise une expression régulière pour extraire tous les placeholders du modèle (`\$client`, `\$banque`, `\$etablissement`).

ts

```
function extractPlaceholders(template: string): string[] {
  const matches = template.match(/\$[a-zA-Z]+/g);
  return matches ? [...new Set(matches)] : [];
}
```

Étape 2 : Résolution des variables via API

Tu peux créer un service qui gère la résolution :

```
ts

@Injectable({ providedIn: 'root' })
export class PlaceholderResolverService {
  constructor(
    private http: HttpClient
  ) {}

  resolvePlaceholders(placeholders: string[]): Observable<Record<string, string>> {
    const apiCalls: Observable<[string, string]>[] = [];

    placeholders.forEach(ph => {
      const key = ph.substring(1); // remove the $
      let api$: Observable<string>;

      switch (key) {
        case 'client':
          api$ = this.http.get<{ nom: string }>('/api/client').pipe(map(res => res.nom));
          break;
        case 'banque':
          api$ = this.http.get<{ nom: string }>('/api/banque').pipe(map(res => res.nom));
          break;
        case 'etablissement':
          api$ = this.http.get<{ nom: string }>('/api/etablissement').pipe(map(res => res.nom));
          break;
        default:
          api$ = of(`[${key} inconnu]`);
      }

      apiCalls.push(api$.pipe(map(value => [ph, value])));
    });

    return forkJoin(apiCalls).pipe(
      map(results => Object.fromEntries(results))
    );
  }
}
```

Étape 3 : Remplacement des variables dans le modèle

```
ts

function replacePlaceholders(template: string, values: Record<string, string>): string {
  return Object.entries(values).reduce((acc, [key, value]) => {
    const regex = new RegExp(`\\${key}\\b`, 'g'); // match full placeholder
    return acc.replace(regex, value);
  }, template);
}
```

Étape 4 : Intégration dans ton composant

```
ts

insertTemplate(template: string): void {
  const placeholders = extractPlaceholders(template);

  this.placeholderResolver.resolvePlaceholders(placeholders).subscribe(resolved => {
    const filledTemplate = replacePlaceholders(template, resolved);

    // Insérer dans Outlook ici via Office.js
    Office.context.mailbox.item.body.setSelectedDataAsync(filledTemplate, { coercionType: Office.CoercionType.Text });
  });
}
```

Bonus : Typage (facultatif mais recommandé)

Crée une interface ou enum pour les placeholders connus afin d'avoir un meilleur contrôle à la compilation.

Souhaites-tu que je t'aide à intégrer cette logique dans un composant standalone Angular avec Fluent UI ?



Propose moi l'option d'interface.

Les placeholders sont `{{CLIENT}}`, `{{ETABLISSEMENTS}}`, `{{BANQUE}}`.

Dans mon modèle de mail, j'ai par exemple `{{CLIENT;NOM}}`, tout ce qui comment par CLIENT correspond à un seul appel ainsi de suite

Parfait. Voici une solution optimisée en Angular 19 avec :

- Des **placeholders en** ``{{...}}`` (ex: ``{{CLIENT;NOM}}``).
- Un appel API **par type principal** (``CLIENT``, ``BANQUE``, ``ETABLISSEMENTS``).
- Une interface pour typer les données des variables.
- Un remplacement de tous les sous-champs (ex: ``{{CLIENT;NOM}}`` → ``Jean Dupont``) à partir de la réponse API unique.

1. Exemple de modèle de mail

txt

Bonjour `{{CLIENT;NOM}}`,
Votre demande a été transmise à la banque `{{BANQUE;NOM}}` et à l'établissement `{{ETABLISSEMENTS;VILLI`

2. Interfaces de données

ts

```
export interface ClientData {  
  NOM: string;  
  EMAIL: string;  
  [key: string]: string;  
}  
  
export interface BanqueData {  
  NOM: string;  
  AGENCE: string;  
  [key: string]: string;  
}  
  
export interface EtablissementData {  
  NOM: string;  
  VILLE: string;  
  [key: string]: string;  
}
```

3. Détection des placeholders avec groupes

```
ts

function extractPlaceholderGroups(template: string): Map<string, Set<string>> {
  const regex = /\{\{(\w+);(\w+)\}\}/g;
  const groups = new Map<string, Set<string>>();
  let match;

  while ((match = regex.exec(template)) !== null) {
    const [, type, field] = match;
    if (!groups.has(type)) {
      groups.set(type, new Set());
    }
    groups.get(type)?.add(field);
  }

  return groups;
}
```

4. Service de résolution

```
ts

@Injectable({ providedIn: 'root' })
export class PlaceholderResolverService {
  constructor(private http: HttpClient) {}

  resolveData(groups: Map<string, Set<string>>): Observable<Record<string, Record<string, string>>> {
    const result: Record<string, Observable<any>> = {};

    groups.forEach((_, type) => {
      switch (type) {
        case 'CLIENT':
          result['CLIENT'] = this.http.get<ClientData>('/api/client');
          break;
        case 'BANQUE':
          result['BANQUE'] = this.http.get<BanqueData>('/api/banque');
          break;
        case 'ETABLISSEMENTS':
          result['ETABLISSEMENTS'] = this.http.get<EtablissementData>('/api/etablislements');
          break;
        default:
          result[type] = of({});
      }
    });

    return forkJoin(result);
  }
}
```

5. Remplacement dans le modèle

```
ts

function replacePlaceholders(template: string, data: Record<string, Record<string, string>>): string {
  return template.replace(/\{\{(\w+);(\w+)\}\}/g, (_, type, field) => {
    const value = data[type]?.[field];
    return value !== undefined ? value : `{{${type};${field}}}`;
  });
}
```

```
});  
}
```

6. Utilisation dans ton composant Angular

```
ts  
  
insertTemplate(rawTemplate: string): void {  
  const placeholderGroups = extractPlaceholderGroups(rawTemplate);  
  
  this.placeholderResolver.resolveData(placeholderGroups).subscribe(data => {  
    const filledTemplate = replacePlaceholders(rawTemplate, data);  
  
    // Insertion dans Outlook via Office.js  
    Office.context.mailbox.item.body.setSelectedDataAsync(filledTemplate, {  
      coercionType: Office.CoercionType.Html,  
    });  
  });  
}
```

Souhaites-tu que je t'envoie un fichier Angular complet avec composant standalone pour intégrer tout ça proprement ?



Et est-ce que c'est optimisé cette manière de faire?

Oui, la solution proposée est **optimisée et scalable** pour ton besoin, notamment dans un projet Angular moderne. Voici pourquoi :

1. Optimisations en place

a. Un seul appel API par entité

Tu détectes les types (`CLIENT`, `BANQUE`, `ETABLISSEMENTS`) et fais **un seul appel** API par type, peu importe le nombre de champs utilisés (`NOM`, `EMAIL`, etc.).

Cela évite les appels redondants et réduit la latence.

b. `Map<string, Set<string>>` pour éviter les doublons

Tu ne traites que les placeholders uniques et par groupe.

C'est plus rapide que de parcourir chaque occurrence dans le template.

c. `forkJoin()` pour faire les appels API en parallèle

Les appels sont parallélisés, donc **le temps total = le plus lent des 3 appels**.

Idéal pour la réactivité dans un add-in Outlook.

d. Remplacement par regex compilée une seule fois

La regex ``template.replace(/\{\{(\w+);(\w+)\}\}/g, ...)`` est traitée en **un seul passage**, avec substitution immédiate.

2. Facile à maintenir et étendre

- Ajouter un nouveau type (``{{PROJET;NOM}}``) = juste une ligne dans le ``switch``.
- Compatible avec typage fort via TypeScript.
- La logique est séparée : détection, API, injection — **propre et testable**.

3. Alternative plus avancée (optionnelle)

Si un jour tu veux **éviter tout appel API inutile** même si ``{{CLIENT;NOM}}`` est dans le template mais que ``CLIENT`` est déjà dans le cache/mémoire, tu pourrais :

- Ajouter un cache temporaire (``Map<string, any>``)
- Ne faire appel à l'API que si l'entité n'est pas déjà chargée.

Conclusion : Oui, cette solution est optimisée **pour performance, lisibilité, et maintenabilité** dans un contexte Angular 19 / Web Add-in.

Souhaites-tu que je t'aide à encapsuler cette logique dans un composant Angular standalone prêt à l'emploi ?



Je n'ai pas compris la partie `extractionPlaceholder`

Très bonne remarque — je vais t'expliquer **clairement et étape par étape** ce que fait la fonction ``extractPlaceholderGroups`` (appelée ici ``extractionPlaceholder`` dans ton message).

Objectif

Analyser un modèle de mail comme :

```
txt

Bonjour {{CLIENT;NOM}}, votre agence est {{ETABLISSEMENTS;VILLE}}. Votre conseiller est {{CLIENT;PRI
```

Et en sortir une structure comme :

```
ts

Map {
  "CLIENT" => Set { "NOM", "PRENOM" },
  "ETABLISSEMENTS" => Set { "VILLE" }
}
```

Pourquoi cette structure ?

Tu veux :

- Appeler une seule fois l'API `/api/client`, même si plusieurs champs de CLIENT sont utilisés.
- Savoir quels champs sont attendus par entité pour les injecter ensuite.

La fonction ligne par ligne

```
ts

function extractPlaceholderGroups(template: string): Map<string, Set<string>> {
  const regex = /\{\{(\w+);(\w+)\}\}/g; // correspond à {{TYPE;CHAMP}}, ex: {{CLIENT;NOM}}
  const groups = new Map<string, Set<string>>();
  let match;

  while ((match = regex.exec(template)) !== null) {
    const [, type, field] = match; // ex: match = ['{{CLIENT;NOM}}', 'CLIENT', 'NOM']

    if (!groups.has(type)) {
      groups.set(type, new Set()); // initialise le Set pour ce type
    }

    groups.get(type)?.add(field); // ajoute le champ au Set correspondant
  }

  return groups;
}
```

Exemple d'utilisation

```
ts

const template = `
  Bonjour {{CLIENT;NOM}}, vous êtes suivi par {{CLIENT;PRENOM}}.
  Votre établissement est {{ETABLISSEMENTS;VILLE}}.
`;

const result = extractPlaceholderGroups(template);
console.log(result);
// Affiche :
// Map {
//   'CLIENT' => Set { 'NOM', 'PRENOM' },
//   'ETABLISSEMENTS' => Set { 'VILLE' }
// }
```

Tu veux une version plus simple ?

Voici une version commentée, pas à pas :

```
ts

function extractPlaceholderGroups(template: string): Map<string, Set<string>> {
  const regex = /\{\{(\w+);(\w+)\}\}/g;
  const groups = new Map();

  let match;
```

```
while ((match = regex.exec(template))) {
  const type = match[1]; // ex: "CLIENT"
  const field = match[2]; // ex: "NOM"

  if (!groups.has(type)) {
    groups.set(type, new Set());
  }

  groups.get(type).add(field);
}

return groups;
}
```

Souhaites-tu que je te fournisse un petit fichier `.ts` prêt à tester avec un exemple ?



Reprend la même fonction en utilisant les potentialités de typescript

Bien sûr ! Voici une **version TypeScript typée, lisible et robuste** de la fonction `extractPlaceholderGroups`, en exploitant les fonctionnalités de TypeScript comme les types génériques, `Map`, `Set` et des types d'entités déclarés :

1. Déclaration des types d'entités possibles (optionnel mais recommandé)

```
ts

export type PlaceholderType = 'CLIENT' | 'BANQUE' | 'ETABLISSEMENTS';
export type PlaceholderGroups = Map<PlaceholderType, Set<string>>;
```

Cela permet de garantir que seules les entités connues sont traitées.

2. Version typée et optimisée de `extractPlaceholderGroups`

```
ts

export function extractPlaceholderGroups(template: string): PlaceholderGroups {
  const regex = /\{\{(\w+);(\w+)\}\}/g;
  const result: PlaceholderGroups = new Map();

  let match: RegExpExecArray | null;

  while ((match = regex.exec(template)) !== null) {
    const [, rawType, field] = match;
    const type = rawType.toUpperCase() as PlaceholderType;

    if (!result.has(type)) {
      result.set(type, new Set());
    }

    result.get(type)!.add(field.toUpperCase()); // standardiser le champ aussi
  }

  return result;
}
```


Explication des points TypeScript importants

- ``PlaceholderType`` limite les types reconnus : ``'CLIENT' | 'BANQUE' | 'ETABLISSEMENTS'``
- ``Map<PlaceholderType, Set<string>>`` garantit une structure forte, claire et itérable.
- ``as PlaceholderType`` force une correspondance stricte : si un type non reconnu est passé, une erreur TypeScript peut t'aider à le corriger.
- ``result.get(type)!:`` : le `!`` est utilisé ici parce qu'on sait que la clé existe après l'avoir ajoutée juste au-dessus.

Exemple d'utilisation avec typage

```
ts

const template = `
  Bonjour {{CLIENT;NOM}}, votre banque est {{BANQUE;NOM}}.
  Le représentant de {{ETABLISSEMENTS;VILLE}} vous contactera.
`;

const placeholders = extractPlaceholderGroups(template);

// Tu peux ensuite faire :
placeholders.forEach((fields, type) => {
  console.log(`Type: ${type} | Champs: ${Array.from(fields).join(', ')}`);
});
```

Souhaites-tu que je t'intègre cette fonction dans un service Angular, ou que je te montre une version testable avec un composant standalone Angular 19 ?



Je souhaite que lorsqu'un appel API a échoué que l'utilisateur puisse saisir les variables correspondantes présentés dans un formulaire réactif

Parfait — voici une solution **complète et propre** pour que l'utilisateur **puisse saisir manuellement** les données si un appel API échoue, avec affichage dynamique d'un **formulaire réactif basé sur les variables attendues**.

Fonctionnalité cible

1. Tu détectes les placeholders (ex: ``{{CLIENT;NOM}}``, ``{{BANQUE;NOM}}``).
2. Tu appelles les API correspondantes (``/api/client``, etc.).
3. Si une API échoue, tu génères un **formulaire dynamique** basé sur les champs manquants (ex: ``CLIENT -> NOM, PRENOM``).
4. L'utilisateur les saisit.
5. Tu remplaces les variables dans le modèle avec les valeurs saisies.

Architecture proposée

- **Service `PlaceholderResolverService`** : tente les appels API et indique les échecs.
- **Composant Angular** : affiche un formulaire basé sur les champs manquants.
- **FormBuilder + ReactiveFormsModule**.

1. Résultat du Resolver : succès/échec

```
ts

export interface PlaceholderResolutionResult {
  resolved: Record<string, Record<string, string>>; // données résolues
  missing: Map<string, Set<string>>; // données à remplir
}
```

2. `PlaceholderResolverService` mis à jour

```
ts

@Injectable({ providedIn: 'root' })
export class PlaceholderResolverService {
  constructor(private http: HttpClient) {}

  resolveData(groups: Map<string, Set<string>>): Observable<PlaceholderResolutionResult> {
    const resolved: Record<string, Record<string, string>> = {};
    const missing: Map<string, Set<string>> = new Map();

    const calls = Array.from(groups.entries()).map(([type, fields]) => {
      let api$: Observable<any>;

      switch (type) {
        case 'CLIENT':
          api$ = this.http.get('/api/client');
          break;
        case 'BANQUE':
          api$ = this.http.get('/api/banque');
          break;
        case 'ETABLISSEMENTS':
          api$ = this.http.get('/api/etablissements');
          break;
        default:
          api$ = of(null);
      }

      return api$.pipe(
        map(data => ({ type, data })),
        catchError(() => {
          missing.set(type, fields);
          return of({ type, data: null });
        })
      );
    });

    return forkJoin(calls).pipe(
      map(results => {
        results.forEach(({ type, data }) => {
          if (data) {
            resolved[type] = data;
          }
        });

        return { resolved, missing };
      })
    );
  }
}
```

```
});
}
}
```

3. Composant Angular (résumé)

```
ts

@Component({
  selector: 'app-template-inserter',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule, FormsModule],
  templateUrl: './template-inserter.component.html'
})
export class TemplateInserterComponent implements OnInit {
  form = this.fb.group({});
  missingData: Map<string, Set<string>> = new Map();
  template = ''; // à remplir dynamiquement

  constructor(
    private fb: FormBuilder,
    private resolver: PlaceholderResolverService
  ) {}

  ngOnInit() {
    const groups = extractPlaceholderGroups(this.template);

    this.resolver.resolveData(groups).subscribe(({ resolved, missing }) => {
      this.missingData = missing;

      // Construction du formulaire si données manquantes
      if (missing.size > 0) {
        missing.forEach((fields, type) => {
          fields.forEach(field => {
            this.form.addControl(`${type}_${field}`, new FormControl('', Validators.required));
          });
        });
      } else {
        this.insertTemplate(resolved);
      }
    });
  }

  onSubmitManual() {
    const manualData: Record<string, Record<string, string>> = {};

    Object.keys(this.form.value).forEach(key => {
      const [type, field] = key.split('_');
      if (!manualData[type]) manualData[type] = {};
      manualData[type][field] = this.form.value[key];
    });

    this.insertTemplate(manualData);
  }

  insertTemplate(data: Record<string, Record<string, string>>) {
    const allPlaceholders = extractPlaceholderGroups(this.template);
    const finalData = { ...data };

    // Ajouter manuellement les champs manquants si besoin
    this.missingData.forEach((fields, type) => {
      if (!finalData[type]) finalData[type] = {};
      fields.forEach(field => {
        finalData[type][field] = this.form.value[`${type}_${field}`];
      });
    });

    const filled = replacePlaceholders(this.template, finalData);
    Office.context.mailbox.item.body.setSelectedDataAsync(filled, { coercionType: Office.CoercionTypeText });
  }
}
```

```
}  
}
```

4. Template HTML associé (`template-insertter.component.html`)

```
html  
  
<form *ngIf="missingData.size > 0" [formGroup]="form" (ngSubmit)="onSubmitManual()">  
  <div *ngFor="let type of Array.from(missingData.keys())">  
    <h3>{{ type }}</h3>  
    <div *ngFor="let field of Array.from(missingData.get(type) || [])">  
      <label>{{ field }}</label>  
      <input [formControlName]="type + '_' + field" />  
    </div>  
  </div>  
  <button type="submit" [disabled]="form.invalid">Insérer</button>  
</form>
```

Souhaites-tu que je t'envoie ce composant packagé dans un projet Angular 19 standalone prêt à tester ?