15. règles de validation de la `demande d'avis`

Règles validation générale

- Les `code` d'énumération doivent existé et être valide en édition.
- La taille des champs doit être vérifier
- la taille et l'extension des fichiers doivent être vérifiés

Pour le dépôt

La demande d'avis

- Le `statut code` doit être `en cours de création` (code `EnCreation`) ou `À compléter` (code `ACompleter`).

Demande conjointe

- Le `titre code conjoint` doit être compatible (même `pouvoir`) avec le `titre code` principal.
- **Si** au moins un `titre conjoint` **alors** au moins un `mandat`.
- Il **ne** peut **pas** y avoir de `mandat` **sans** au moins un `titre conjoint`.

Le formulaire

- Une `norme` obligatoire
- Au moins un intitulé non vide
- Un `Délai` obligatoire
- Si `Art 84, 3° (5 j/d)` (code `Art84_3`) alors la `délai motivation` est non vide

Les délégués

- Au moins un `délégué destinataire` obligatoire
- Les champs `Prénom` , `Nom` , `Email` doivent être non vide
- Le `Gsm` **ou** le `Fixe` doit être non vide
- Le `type code` est obligatoire

Les documents

- Au moins le `document projet` est obligatoire

La signature (La validation)

- Un et un seul `document` faisant office de validation parmi les `type code` suivant [`LettreSignee` , `PreuveDeSignature` , `ValidationSMS`]
- La `ValidationSMS` n'est possible que si le `demandeur` a fourni un numéro de `GSM` (interface `Prolex` encodage des `demandeurs`)

`document-hub` étant l'adresse à laquelle le `Hub` est joignable.

Ensuite tout se passe dans le `endpoint` en injectant le `Hub` dans `Request Delegate` :

`DocumentEndpoints.cs`

```
group.MapPost("/add/{connectionId}",
    async (
        DocumentRepository repo,
        Document documentToAdd,
        IHubContext<DocumentHub> context,
        [FromHeader(Name = "Connection-Id")] string connectionId
    ) =>
{
    var document = await repo.AddAsync(documentToAdd);

    await context.Clients
        .Client(connectionId)
        .SendAsync(DocumentEvent.Added, document);

    await context.Clients
        .AllExcept(connectionId)
        .SendAsync(DocumentEvent.ChangeNotifyed);
});
```

Grâce à `SignalR`, le `endpoint` notifie à l'expéditeur (`connectionID` unique attribué à un canal `SignalR`) que le le document a bien été ajouté et renvoie le `document` avec maintenant un `Id` donné par la `DB`.

> Le `endpoint` pourrait traditionnellement renvoyer un `Results.Created`.

> On pourrait aussi passer le `connectionId` par les `headers` de la requête :
>
> ```
> group.MapPost("/add",
>  async (
>      DocumentRepository repo,
>      Document documentToAdd,
>      IHubContext<DocumentHub> context,
>      [FromHeader(Name = "Connection-Id")] string connectionId
>  ) =>
> ```
>
> Et dans le `client` :
>
> ```
> client = Factory.CreateClient("APIClient");
>
> client
>  .DefaultRequestHeaders
>  .Add("Connection-Id", Provider?.ConnectionId ?? "0");
>
> await client.PostAsJsonAsync("/add-item", Item);
> ```

DD `Re-render` une page

`StateHasChanged`

Pour forcer un re-rendu d'une page (un composant page), on utilise la méthode `StateHasChanged` :

```
public async void GetColour2()
{
    Colour2 = await httpClient.GetFromJsonAsync<Colour>(urlColour);

    StateHasChanged();
}
```

`InvokeAsync`

Dans le cas où `StateHasChanged` est appelé d'un autre `Thread`, par exemple une `callback` avec `SignalR`, on utilise alors `InvokeAsync` :

```
protected override async Task OnInitializedAsync()
{
    // ...
     Provider.Connection.On("RefreshTitreConjoint", () => LoadItems());
}
```

```
private async Task LoadItems(bool isAsyncChanged = false)
{
    _isLoading = true;
    await InvokeAsync(StateHasChanged);

    _titresAvailable = await Repo.GetTitres(DemandeAvisId);

    _isLoading = false;
    await InvokeAsync(StateHasChanged);
}
```