Lignes directrices pour le développement

Général

Nommage des variables et méthodes

Mettre des noms explicites qui permettent, à la lecture d'une ligne, de comprendre immédiatement ce que contient une variable ou ce que fait une fonction (sans avoir lu le code avant ou après cette ligne).

> Exemple de mauvais nommage: item.

Pas de franglish, tout mettre en anglais correctement traduit.

Rester constant

Rester constant dans les manières de faire, l'architecture, le nommage, etc… Se baser sur ce qui existe déjà dans le projet en tant qu'exemple.

Prioriser les couches basses de l'architecture

Se poser la question de savoir si tout ou partie d'un algorithme ne pourrait pas être déplacé plus bas dans l'architecture. Plus c'est bas dans l'architecture, plus l'application est stable et facile à maintenir.

> Exemples de déplacement d'algorithmes :
>
> - Frontend Backend
> - Service Repository
> - Composant Service (voir le cas spécifique Ionic/Angular ci-dessous)

Ionic/Angular (frontend)

Séparation composant/service

Essayer de bien séparer les responsabilités en poussant le maximum de logique et de traitements dans le service. Le composant doit uniquement servir pour la logique d'affichage.

Par exemple :

- Appel REST, interactions avec le state Elf Service
- Affichage d'un loader Composant

Exemple de mise en oeuvre

Ces extraits de code font la même chose, le second est clairement plus facile à lire (et donc à maintenir).

Avant

```
async createContact(user: Contact) {
 if (Capacitor.isNativePlatform()) {
  const list = await Contacts.getContacts({projection: {emails: true}})
  list.contacts.map(async (contact) => {
    if (contact.emails.some(email => user.mailAdresses.includes(email.address))) {
      const toast = await this.toastController.create({
        message: this.translateService.instant('CONTACTS.ALERT.ERROR.EXIST'),
        duration: 1500,
        position: 'middle',
        color: 'warning'
      });
      toast.present();
      return;
    }
  })
    const phones = []
    const emails = []
    user.mobileNumbers.map(phone => (
      phones.push({
        type: PhoneType.Mobile,
        label: 'mobile',
        number: phone,
      })
    ));
    user.mailAdresses.map(email => (
      emails.push({
        type: EmailType.Home,
        label: 'email',
        address: email,
      })
    ))
    await Contacts.createContact({
      contact: {
        name: { given: user.firstname, family: user.name },
        phones: phones,
        emails: emails,
      },
    });
    const toast = await this.toastController.create({
      message: this.translateService.instant('CONTACTS.ALERT.SUCCESS.MESSAGE'),
      duration: 1500,
      position: 'middle',
      color: 'success'
    });
    toast.present();
  } else {
    const alert = await this.alertController.create({
      header: this.translateService.instant('CONTACTS.ALERT.ERROR.HEADER'),
      subHeader: this.translateService.instant('CONTACTS.ALERT.ERROR.SUBHEADER'),
      message: this.translateService.instant('CONTACTS.ALERT.ERROR.MESSAGE'),
      buttons: [this.translateService.instant('CONTACTS.ALERT.ERROR.BUTTON1')],
    });
    await alert.present();
  }
}
```

Après

```
async createContact(user: Contact) {
 if (!Capacitor.isNativePlatform()) {
    const alert = await this.alertController.create({
      header: this.translateService.instant('CONTACTS.ALERT.ERROR.HEADER'),
      subHeader: this.translateService.instant('CONTACTS.ALERT.ERROR.SUBHEADER'),
      message: this.translateService.instant('CONTACTS.ALERT.ERROR.MESSAGE'),
      buttons: [this.translateService.instant('CONTACTS.ALERT.ERROR.BUTTON1')],
    });
    await alert.present();
    return;
 }
 if (this.contactsService.contactAlreadyExists(user)) {
  const alreadyExist = await this.toastController.create({
    message: this.translateService.instant('CONTACTS.ALERT.ERROR.EXIST'),
    duration: 1500,
    position: 'middle',
    color: 'warning'
  });
  await alreadyExist.present();
  return;
 }
 this.contactsService.createContact(user);
  const toast = await this.toastController.create({
    message: this.translateService.instant('CONTACTS.ALERT.SUCCESS.MESSAGE'),
    duration: 1500,
    position: 'middle',
    color: 'success'
  });
  toast.present();
}
```

Template HTML

- Utiliser au maximum les composants Ionic.
- Eviter les balises HTML "classiques" (ex: div, ul, li, etc…).
- Eviter autant que possible d'utiliser du CSS personnalisé (plutôt favoriser les classes prédéfinies d'Ionic).
- Essayer de garder le DOM aussi léger en minimisant autant que possible le nombre de balises.
- Si la logique dans le template devient trop complexe, faire les traitements dans le composant afin de simplifier au maximum le template.

Exemple de mise en oeuvre (logique trop complexe dans le template)

On déplace ici une partie de la logique dans le composant afin de simplifier et rendre plus lisible le template.

Avant	Après

```
<ng-container *ngIf="(translatedChannels$ | async) as translatedChannels">
  <ng-container *ngIf="notificationChanelIsFilterable(notification.channel, translatedChannels)">
    <ng-container *ngIf="unsubscribedChannels$ | async as unsubscribedChannels ">

      <ng-container *ngIf="!unsubscribedChannels.includes(notification.channel)">
        <ion-item (click)="onSubscribeOrUnsubscribeChannelClick(notification.channel, false)">
          <ion-icon name="close-circle-outline" slot="start"></ion-icon>
          <ion-label class="ion-text-wrap">
            {{ 'NOTIFICATIONS.UNSUBSCRIBE_TO_CHANNEL' | translate }}
            <ng-container *ngFor="let channel of translatedChannels">
              <ion-text *ngIf="channel.code === notification.channel">
                '{{ channel.label }}'
              </ion-text>
            </ng-container>
          </ion-label>
        </ion-item>
      </ng-container>

      <ion-item *ngIf="unsubscribedChannels.includes(notification.channel)" button
        (click)="onSubscribeOrUnsubscribeChannelClick(notification.channel, true)">
        <ion-icon name="notifications-outline" slot="start"></ion-icon>
        <ion-label class="ion-text-wrap">
          {{ 'NOTIFICATIONS.SUBSCRIBE_TO_CHANNEL' | translate }}
          <ng-container *ngFor="let channel of translatedChannels">
            <ion-text *ngIf="channel.code === notification.channel">
              '{{ channel.label }}'
            </ion-text>
          </ng-container>
        </ion-label>
      </ion-item>
    </ng-container>
  </ng-container>
</ng-container>
```

```
<ng-container *ngIf="(notificationOptions$ | async) as options">
  <ion-item button *ngIf="options.isSubscribed && options.filterable"
    (click)="onSubscribeOrUnsubscribeChannelClick(notification.channel
    <ion-icon name="close-circle-outline" slot="start"></ion-icon>
    <ion-label class="ion-text-wrap">
      {{ 'NOTIFICATIONS.UNSUBSCRIBE_TO_CHANNEL' | translate }}
      <ion-text> '{{ options.channel.label }}' </ion-text>
    </ion-label>
  </ion-item>

  <ion-item button *ngIf="!options.isSubscribed && options.filterable"
    (click)="onSubscribeOrUnsubscribeChannelClick(notification.channel
    <ion-icon name="notifications-outline" slot="start"></ion-icon>
    <ion-label class="ion-text-wrap">
      {{ 'NOTIFICATIONS.SUBSCRIBE_TO_CHANNEL' | translate }}
      <ion-text> '{{ options.channel.label }}' </ion-text>
    </ion-label>
  </ion-item>
</ng-container>
```

Rxjs

- Privilégier l'utilisation du pipe async avec un Observable & pipe , plutôt que de faire un `subscribe` qui met à jour une variable "synchrone".

Elf

Distinction Props/Entities

Dans les stores Elf, privilégier les `entities` plutôt que les `props` quand il s'agit de collection d'objet qui disposent d'un identifiant (ID numérique, code unique, etc...) car les entities disposent de méthodes de manipulation de la collection (suppression par l'ID, etc…).

Nest (backend)