MobApp – (Ré)introduction à React Native

Laurent Daverio, Olivier Hermant

Centre de recherche en informatique
MINES Paris, Université PSL

Plan

1. JSX
2. React Native
3. Placement des composants
4. Fonctions
5. Hooks
6. Requêtes à un serveur

1. JSX

JSX : JavaScript and XML

- javascript, balises XML au milieu
- JS : pour la logique
- balises XML : pas ~~du HTML~~ (mais y ressemble)
  - composants React Native
  - servent au rendu
  - liées aux composants graphiques (et composants *natifs*) de l'appareil mobile
- créer ses **propres balises**
  - = développer ses **propres fonctions**
  - doivent retourner un rendu

2. React Native

L'app KivAppA

Mode d'emploi (cf. premier TP)

1. prendre son projet (`git`)
2. connecter la tablette/un téléphone android
3. démarrer le packager de React Native
   (`npx react-native start`)
4. construire le projet et le lancer
   (`npx react-native run-android`)

Dans le fichier `App.js`:

- `imports`
- `export` du composant principal de la page,
- `KivAppA` : la **fonction de rendu**
  - retourne les **composants RN** qui doivent être affichés
  - balises XML, avec propriétés
- [DEMO]

Développer son composant

- donner un nom : `<Matin>`
- l'appeler : RN s'occupe de tout, **sauf**
- développer de la fonction de rendu éponyme :

```
const Matin = (props) => {
  /* du code : logique du composant Matin */
  return (
    /* d'autres composants : rendu du composant Mati
  )
}
```

- **un seul argument : `props`**
  - objet avec des champs
  - correspondance avec les **attributs de la balise XML**
- NB :
  - entre { `accolades` } , le code JS,
  - entre { `accolades` } , dans du code JS, un objet en ligne,
  - conséquence : parfois deux paires d'accolades

3. Placement des composants

Placement des composants : flexboxes

- composant P (conteneur) : propriété de style `flex : n`
- composants fils $F_1, \cdots, F_k$: propriété de style `flex : n_i`
- place prise par F_i dans P :

$$\frac{n_i}{\sum_{j=1}^{k} n_j}$$

- composant P : propriété de style `flexDirection : "column"` (ou `"row"`, etc)
- `<Button>` remplacés par `<TouchableOpacity>`

4. Fonctions

Les fonctions

- fonction = citoyen de 1^{e} classe = variable comme une autre

```
(x,y) => { return x*y }
```

- fonction nommée = affectation de la fonction anonyme à une variable

```
const mult = (x,y) => { return x*y }
```

- on peut passer une fonction en paramètre, exemples

```
<Button onPress={(evt) => {/* ... */}}
<DemiJournee changeDemi={() => {setMatin(!matin)}}/>
```

5. Hooks

Ajouter un état à un composant : `useState`

Chaque composant `<Matin>` a un compteur qui lui est propre.

- nécéssité d'utiliser un **hook** ($\approx$ fonctionalité) particulier,

  le hook d'état

- ajoute des **variables d'état** au composant

  ```
  const [compteur, setCompteur] = useState(0)
  ```

  - variable d'état `compteur` (lecture seule)
  - fonction modificatrice de l'état `setCompteur`
  - valeur initiale : `0`
- variables et fonctions d'état dans les `props`
  - "faire remonter" l'état $\Rightarrow$ partage avec sous-composants
- le `<Button>` demande une **fonction de callback** dans l'attribut `changeDemi`
- appelée au moment opportun (clic sur le bouton)

6. Requêtes à un serveur

Requête au serveur local

1. lancer le serveur local
2. reverse port forwarding
   - tablette $\xrightarrow{\text{requête}}$ VM
   - demander à `adb` de faire le job
   - **reverse port forwarding**

     ```
     adb reverse tcp:5000 tcp:5000
     ```
3. configurer l'URL dans l'app et la lancer

Requêtes en JS

- utilisation de `fetch`

  (simplifiée) : `fetch(url)`

  (complète) : `fetch(url,{ /* paramètres dans un objet */ })`
- objet de paramètres

  `{attr1 : valeur1, attr2: valeur2,...}`
  - `method` : la méthode HTTP (`GET`, `POST`, `DELETE`...)
  - `headers` : un objet contenant les informations de header (`Authorization`, `Content-Type`...)
  - `body` : le corps de la requête (si besoin)
- requête ⇒ attente réponse ⇒ asynchrone ⇒ callbacks
- `fetch` retourne un objet : une promesse

Les promesses

- type d'objet "asynchrone"
- un jour
  - soit sera **tenue**,
  - soit sera **rompue**.
- **à ce moment**, callback appelé
- **il faut enregistrer cette fonction** !
  - `.then` enregistre le callback de succès
  - `.catch` enregistre le callback d'erreur

Promesse tenue

- soit p la promesse créée et renvoyée par fetch
- p appelle le callback en lui fournissant un objet de type `Response`
- la fonction enregistrée par `promise.then((response) => { /* ... */ })` l'utilise
  - les champs de `response` contienent des infos
    - `p.ok`,
    - p.status
    - l'en-tête HTTP de la réponse
  - **mais pas encore le corps** (long à charger... $\Rightarrow$ re-asynchrone !)
  - `return(response.json())` et `return(response.text())` forment donc une **nouvelle promesse** (nouvel objet)
  - qui se résoudra (tenue/rompue) plus tard
- enchaîner les `then`

Promesse rompue

- *p* crée un objet d'erreur
- enregistrer le callback gestionnaire avec `promise.catch((error) => { /* ... */ })`
- on peut inspecter `error` dans le callback
- enregistrement **en dernier** dans la chaîne
- NB :
  - requête envoyée correctement, mais (400,401,404,...) refusée par le serveur = promesse tenue
  - présentation sciemment simplifiée