

# L'animation avec React Native

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. L'API Animated</b>	<b>3</b>
<b>III. Exercice : Appliquez la notion</b>	<b>7</b>
<b>IV. Composer des animations entre elles</b>	<b>7</b>
<b>V. Exercice : Appliquez la notion</b>	<b>10</b>
<b>VI. Animer rapidement et simplement avec LayoutAnimation</b>	<b>11</b>
<b>VII. Exercice : Appliquez la notion</b>	<b>13</b>
<b>VIII. Essentiel</b>	<b>15</b>
<b>IX. Auto-évaluation</b>	<b>15</b>
A. Exercice final .....	15
B. Exercice : Défi.....	17
<b>Solutions des exercices</b>	<b>17</b>

## I. Contexte

**Durée :** 1 h

**Environnement de travail :** Une application React Native initialisée ou utiliser <https://snack.expo.io/>

**Pré-requis :** Notions de styles en React Native et de mise en page, la gestion des props et du state

### Contexte

Nous avons précédemment utilisé `react-navigation`, qui nous a permis de découvrir les premières animations dans notre application. Maintenant, nous allons nous-mêmes réaliser nos propres animations, comme le fait `react-navigation`.

React Native propose deux systèmes d'animation complémentaires : `Animated` pour le contrôle granulaire et interactif de valeurs spécifiques, et `LayoutAnimation` pour les transitions de mises en page globales animées.

Cependant, un point d'attention sur les performances : les animations peuvent tourner dans le thread natif ou le thread JavaScript selon leur nature, ce qui peut impacter la dynamique de l'application.

## II. L'API Animated

### Objectif

- Découvrir les animations avec React Native

### Mise en situation

React Native met à disposition une API permettant d'animer des valeurs numériques, mais également des propriétés graphiques, comme la taille ou la couleur des éléments. On peut également utiliser une des nombreuses fonctions de lissage (`easing`) définissant de quelle manière s'anime cette valeur dans le temps. Celles-ci peuvent être nativement gérées dans React Native, ou définies spécifiquement par le développeur.

### Comment fonctionne l'API Animated ?

Cette API est conçue pour exprimer de manière concise et très performante, via une approche impérative, une grande variété de modèles d'animations et d'interactions. `Animated` se concentre sur les relations déclaratives entre des entrées et des sorties de données, avec des transformations configurables entre les deux (on parle alors d'interpolation), ainsi que des méthodes de démarrage/arrêt pour contrôler l'exécution de l'animation en fonction du temps.

`Animated` exporte six types de composants qui sont animables. Ce sont des versions améliorées des composants de base de React Native, qui peuvent recevoir des propriétés animées :

- `View`
- `Text`
- `Image`
- `ScrollView`
- `FlatList`
- `SectionList`

Et, bien que cela suffise dans la plupart des cas, il est tout à fait possible de créer les nôtres en utilisant `Animated.createAnimatedComponent()`.

## Notre première animation

Commençons simplement en adaptant la taille d'une boîte de manière linéaire. Pour cela, il nous faut deux choses :

- Une valeur numérique à animer : on va utiliser `Animated.Value`
- Une fonction d'animation : on va utiliser `Animated.Timing`

Par défaut, l'animation subit un lissage linéaire, ce qui signifie que sa valeur va varier à la même vitesse tout au long de l'animation. Nous n'avons donc pas besoin de le spécifier.

### Exemple

```
1 import * as React from "react";
2 import { Animated, useWindowDimensions } from "react-native";
3
4 export default function App() {
5   // On récupère la taille de l'écran que l'on arrondit à l'entier.
6   const windowHeight = Math.round(useWindowDimensions().height);
7
8   // On définit une référence de valeur animée qui commence à 0
9   const heightAnim = React.useRef(new Animated.Value(0)).current;
10
11   React.useEffect(() => {
12     // On anime notre valeur jusqu'à la hauteur de la fenetre
13     Animated.timing(heightAnim, {
14       toValue: windowHeight,
15       duration: 10000, // Durant 10 secondes
16       useNativeDriver: false, // Cela sera abordé plus tard
17     }).start();
18   }, [heightAnim, windowHeight]);
19
20   return (
21     <Animated.View
22       style={{ backgroundColor: "red", width: "100%", height: heightAnim }}
23     />
24   );
25 }
26
```

[cf. anim-2.mp4]

Notons l'utilisation du hook `useRef` qui permet de maintenir un état entre les cycles de rendu du composant. Cependant, son altération ne déclenche pas un nouveau rendu, contrairement à `useState` et il permet de communiquer entre les objets du DOM.

`useRef` renvoie un objet ref modifiable dont la propriété `current` est initialisée avec l'argument fourni (`initialValue`). L'objet renvoyé persistera pendant toute la durée de vie du composant.

La grande différence avec `useState`, c'est que `useRef` ne va pas effectuer de render à chaque modification de la donnée, il va la stocker et la conserver pour pouvoir la réutiliser plus loin dans le code. En gros, `useRef` est comme une « boîte » qui pourrait contenir une valeur modifiable dans sa propriété `.current`.

Par exemple, lorsque qu'on utilise un `useState` pour enregistrer les modifications d'un input, celui-ci va être rendu à chaque frappe du clavier alors qu'avec un `useRef`, à chaque frappe du clavier, la donnée va être sauvegardée dans cette "boîte" puis, au moment du clic, être envoyé en une seul fois.

Pour plus d'informations, on peut consulter la documentation à ce sujet<sup>1</sup>.

Dans cet exemple, on utilise le hook `useWindowDimensions` qui permet de récupérer la taille de l'écran. De cette manière, on va pouvoir faire grandir un voile rouge du haut vers le bas de l'écran. On définit une valeur à animer qui commence à 0 (ce qui correspond à la bordure en haut de l'écran) et qui s'animera jusqu'à atteindre `toValue` (qui est la taille de l'écran). La croissance de la hauteur de l'élément se fera sur une période de `duration: 10000`, soit dix secondes. La valeur s'animera donc linéairement de 0 à X (X étant la taille de l'écran) en dix secondes.

On précise également que l'on n'utilise pas le driver natif, car il n'est pas supporté sur la propriété `height` que nous animons.

Enfin, on peut observer l'instruction `start()` qui permet de dire que l'on déclenche notre animation tout de suite. On aurait pu stocker l'animation dans une variable et déléguer son départ si on le souhaitait.

#### Complément Native Driver

Le driver natif permet de déléguer le calcul d'animation au thread natif plutôt qu'au thread JavaScript. De cette manière, on accélère le fonctionnement de notre application et on s'assure de toujours avoir un taux de rafraîchissement maximum.

Bien que cela soit super, il n'est cependant pas possible d'utiliser le driver natif avec toutes les propriétés : il faut surveiller la console d'erreur de React Native pour voir si la propriété que l'on souhaite utiliser est supportée. Le cas échéant, une erreur nous en avertira.

Maintenant, reprenons l'exemple de code ci-dessus et changeons la fonction de lissage en utilisant une fonction de lissage, nommée `Easing.bounce`, permettant de donner un effet de rebond à la fin de notre animation. Plus d'infos ici<sup>2</sup>.

```
1 Animated.timing(heightAnim, {
2   toValue: windowHeight,
3   duration: 10000, // Durant 10 secondes
4   useNativeDriver: false, // Cela sera abordé plus tard
5   easing: Easing.bounce
6 }).start();
```

[cf. anim-6.mp4]

Rechargeons l'application et testons... Génial, n'est-ce pas ?

### Interpolation de valeurs

Parfois, on souhaite qu'une animation influe sur plusieurs propriétés, comme la taille et la couleur en même temps. Pour cela, on pourrait utiliser deux animations différentes, mais il existe en réalité une solution bien plus adaptée : l'utilisation de la fonction `interpolate`. Cette fonction permet de prendre une valeur en entrée et de faire varier sa sortie. On peut donc, avec une même valeur animée, présenter différents types de répercussions.

Reprenons l'exemple plus haut et essayons :

#### Exemple

```
1 import * as React from "react";
2 import { Animated, useWindowDimensions, Easing } from "react-native";
3
4 export default function App() {
5   // On récupère la taille de l'écran que l'on arrondi à l'entier.
6   const windowHeight = Math.round(useWindowDimensions().height);
7 }
```

1 <https://fr.reactjs.org/docs/hooks-reference.html#userref>

2 <https://reactnative.dev/docs/easing#bounce>

```

8 // On définit une référence de valeur animée qui commence à 0
9 const animatedValue = React.useRef(new Animated.Value(0)).current;
10
11 React.useEffect(() => {
12   // On anime notre valeur jusqu'à la hauteur de la fenetre
13   Animated.timing(animatedValue, {
14     toValue: 1,
15     duration: 10000, // Durant 10 secondes
16     useNativeDriver: false, // Cela sera abordé plus tard
17     easing: Easing.bounce,
18   }).start();
19 }, [animatedValue]);
20
21 return (
22   <Animated.View
23     style={{
24       backgroundColor: animatedValue.interpolate({
25         inputRange: [0, 1],
26         outputRange: ["red", "blue"],
27       }),
28       width: "100%",
29       height: animatedValue.interpolate({
30         inputRange: [0, 1],
31         outputRange: [0, windowHeight],
32       }),
33     }}
34   />
35 );
36 }
37

```

[cf. anim-3.mp4]

Nous avons simplifié le fonctionnement de notre code en n'animant la valeur initiale que de 0 à 1. Ensuite, nous interpolons à deux endroits, en passant deux paramètres : une plage d'entrée sous forme de tableau à deux éléments entre 0 et 1 (qui sont les valeurs d'animation), et une plage de sortie de la même taille que notre plage d'entrée, qui sont nos valeurs mappées utilisées par notre composant. On peut animer des couleurs, des nombres et plein d'autres valeurs : c'est très pratique.

On pourrait également spécifier que le composant grandit de 50 % de l'écran en deux secondes, puis reste à cette valeur jusqu'à la septième seconde, avant de mettre trois secondes à prendre 100 % de l'écran.

Pour cela, il suffit de changer le code comme suit : 0.2 et 0.7, représentant les fractions d'avancement, sachant que notre animation dure dix secondes et que l'on anime de 0 à 1.

```

1 height: animatedValue.interpolate({
2   inputRange: [0, 0.2, 0.7, 1],
3   outputRange: [0, windowHeight / 2, windowHeight / 2, windowHeight],
4 }),

```

[cf. anim-7.mp4]

**Syntaxe**   **À retenir**

L'`Animated` API est un outil très puissant qui permet de gérer les animations dans React Native. Elle utilise des outils tels que :

- Une ou des valeurs à animer
- Une fonction d'animation
- Une fonction de lissage
- Des fonctions d'interpolation qui font correspondre des plages d'entrées et de sorties

Grâce à cela, on peut faire varier des propriétés, comme la couleur de l'arrière-plan ou la taille d'un élément, très simplement et sur commande.

**Complément**

- <https://reactnative.dev/docs/animations>
- <https://reactnative.dev/docs/easing>

### III. Exercice : Appliquez la notion

**Question**

[solution n°1 p.19]

Créez une animation qui change l'opacité de l'arrière-plan bleu de l'application au clic sur un bouton « **Disparition en fondu** ». Cette animation devra être linéaire et prendre cinq secondes à s'effectuer. On peut annuler cette action grâce à un bouton « **Stop** » utilisant la méthode `stop`, qui est une des méthodes disponibles, tout comme `start`.

[cf. anim-5.mp4]

### IV. Composer des animations entre elles

**Objectifs**

- Apprendre le séquençage d'animation
- Voir des exemples de composition

**Mise en situation**

Maintenant que nous savons animer des éléments de notre interface dans le temps, nous pouvons passer à l'étape suivante et utiliser de la composition dans nos animations. Par exemple, on peut choisir de déclencher une animation une fois qu'une autre animation a terminé, ou bien nous pourrions faire tourner une animation en boucle. Avec l'API `Animated`, tout cela est trivial.

**Composer plusieurs animations entre elles**

Pour ce premier exemple, voyons un enchaînement d'animations, qui s'exécutent les unes après les autres sous la forme d'une pile d'exécution.

Nous allons voir un composant **carte** qui possède une opacité de 50 %. Au clic sur la carte, nous allons faire varier son opacité à 100 %, puis, une seconde après, faire tourner la carte à 360 degrés en trois secondes. Et cela avant de reprendre son état initial.

[cf. anim-1.mp4]

Exemple

```

1 import * as React from 'react';
2 import {
3   Animated,
4   Text,
5   TouchableOpacity,
6   View,
7   StyleSheet,
8 } from 'react-native';
9
10 function UserCard() {
11   const [isAnimating, setAnimating] = React.useState(false);
12   const animatedOpacity = React.useRef(new Animated.Value(0.5)).current;
13   const animatedRotation = React.useRef(new Animated.Value(0)).current;
14
15   return (
16     <TouchableOpacity
17       disabled={isAnimating}
18       onPress={() => {
19         setAnimating(true);
20
21         // On crée une séquence d'enchaînement
22         Animated.sequence([
23           // Cette animation affecte l'opacité
24           Animated.timing(animatedOpacity, {
25             toValue: 1,
26             duration: 1000,
27             useNativeDriver: true,
28           }),
29           // Puis celle-ci affecte la rotation
30           Animated.timing(animatedRotation, {
31             toValue: 1,
32             duration: 3000,
33             delay: 1000, // On attends une seconde avant de la démarrer
34             useNativeDriver: true,
35           }),
36         ]).start(() => {
37           // On empêche de relancer l'animation pendant qu'elle tourne grâce à cette callback
38           de fin d'animation
39           setAnimating(false);
40
41           // On réinitialise les valeurs des animations
42           animatedOpacity.setValue(0.5);
43           animatedRotation.setValue(0);
44         });
45       <<>
46       <Animated.View
47         style={[
48           userCardStyles.card,
49           {
50             opacity: animatedOpacity,
51             transform: [
52               {
53                 rotate: animatedRotation.interpolate({
54                   inputRange: [0, 1],
55                   outputRange: ['0deg', '360deg'],
56                 }

```



```

57     ],
58   },
59   ]}]>
60   <Text>Andréas HANSS</Text>
61   <Text>
62     Je suis le créateur de ce cours et je vous apprend à utiliser React
63     Native
64   </Text>
65 </Animated.View>
66 </TouchableOpacity>
67 );
68 }
69
70 const userCardStyles = StyleSheet.create({
71   card: {
72     backgroundColor: 'white',
73     borderRadius: 5,
74     padding: 20,
75     shadowColor: '#000',
76     shadowOffset: {
77       width: 0,
78       height: 2,
79     },
80     shadowOpacity: 0.25,
81     shadowRadius: 3.84,
82     elevation: 5,
83   },
84 });
85
86 export default function App() {
87   return (
88     <View style={{ margin: 20 }}>
89       <UserCard />
90     </View>
91   );
92 }

```

Dans cet exemple, on passe un tableau d'animation à la méthode `Animated.sequence`, qui va enchaîner les animations dans le tableau. On pourrait également exécuter les animations de manière parallèle : pour cela, on utiliserait `Animated.parallel`<sup>1</sup>. Enfin, on peut faire boucler une animation avec `Animated.loop`<sup>2</sup>. Chacune de ces fonctions est utilisable les unes dans les autres, ce qui permet de composer des animations complexes.

### Exemple

En reprenant l'exemple précédent, si on remplace `Animated.sequence` par `Animated.parallel`, toutes les animations présentes dans le tableau sont exécutées en même temps. Pour une meilleure visualisation de cette méthode, on peut enlever le délai d'exécution de la rotation et allonger la durée de l'animation sur l'opacité :

```

1 [... ]
2 // On crée une parallélisation des animations
3 Animated.parallel([
4   // Cette animation affecte l'opacité
5   Animated.timing(animatedOpacity, {
6     toValue: 1,

```

<sup>1</sup> <https://reactnative.dev/docs/animated#parallel>

<sup>2</sup> <https://reactnative.dev/docs/animated#loop>

```

7     duration: 3000,
8     useNativeDriver: true,
9   }),
10  // Puis celle-ci affecte la rotation
11  Animated.timing(animatedRotation, {
12    toValue: 1,
13    duration: 3000,
14    useNativeDriver: true,
15  }),
16  ]).start(() => {
17    // On empeche de relancer l'animation pendant qu'elle tourne grace à cette callback de fin
18    d'animation
19    setAnimating(false);
20
21    // On réinitialise les valeurs des animations
22    animatedOpacity.setValue(0.5);
23    animatedRotation.setValue(0);
24  });
25  [...]
```

Andréas HANSS  
Je suis le créateur de ce cours et je vous apprend à utiliser React Native

### Syntaxe À retenir

- Grâce à l'API `Animated` et ses nombreuses fonctions, on peut créer des animations complexes et avancées par la combinaison de tous les outils mis à notre disposition.
- Chaque animation est rattachable à une autre et on peut contrôler leur exécution à la demande, grâce à la méthode `start` et à la méthode `stop`.

### Complément

- <https://reactnative.dev/docs/animations#composing-animations>

## V. Exercice : Appliquez la notion

### Question

[solution n°2 p.19]

En vous aidant de ce qui a été vu dans ce chapitre et de la documentation de `Animated.loop`<sup>1</sup>, affichez une liste de cartes grâce à un composant personnalisé qui prend en props un titre, un texte de prévisualisation et un booléen pour savoir si la carte en est une nouvelle.

Si c'est une carte nouvelle, affichez dans le coin haut droit un texte « *nouveau* » vert, qui doit clignoter à intervalle régulier pour mettre en évidence le caractère nouveau de la carte.

Voici un exemple du rendu escompté :

[cf. demo-exercice -2.mp4]

<sup>1</sup> <https://reactnative.dev/docs/animated#loop>

## VI. Animer rapidement et simplement avec LayoutAnimation

### Objectifs

- Comprendre l'utilité de l'API `LayoutAnimation`
- Animer un changement d'état très simplement

### Mise en situation

Maintenant que nous savons créer des animations complexes en utilisant toute la puissance de l'API `Animated` de React Native, nous allons voir une façon très rapide de créer des animations simples pour nos changements d'états applicatifs, cela sans avoir à coder une animation complète spécifique. De cette manière, on peut ajouter un peu d'ambiance dans une application, très rapidement et sans effort.

### LayoutAnimation à la rescousse

`LayoutAnimation` est une API destinée à animer des éléments au changement d'un état dans l'interface. Son utilisation est triviale, elle ne possède que deux méthodes : `configureNext` et `create`. Voyons un exemple, que nous allons expliquer.

#### Remarque

Sur Android, il est nécessaire d'activer une API expérimentale, sans quoi cela ne fonctionnera pas. Pour cela, il faudra utiliser l'instruction suivante :

```
1 if (  
2   Platform.OS === "android" &&  
3   UIManager.setLayoutAnimationEnabledExperimental  
4 ) {  
5   UIManager.setLayoutAnimationEnabledExperimental(true);  
6 }
```

#### Exemple

```
1 import React, { useState } from "react";  
2 import {  
3   View,  
4   Platform,  
5   UIManager,  
6   LayoutAnimation,  
7   StyleSheet,  
8   Button,  
9 } from "react-native";  
10  
11 if (  
12   Platform.OS === "android" &&  
13   UIManager.setLayoutAnimationEnabledExperimental  
14 ) {  
15   UIManager.setLayoutAnimationEnabledExperimental(true);  
16 }  
17  
18 const App = () => {  
19   const [boxPosition, setBoxPosition] = useState("left");  
20  
21   const toggleBox = () => {
```

```

22   LayoutAnimation.configureNext(LayoutAnimation.Presets.spring);
23   setBoxPosition(boxPosition === "left" ? "right" : "left");
24 };
25
26 return (
27   <View style={styles.container}>
28     <View style={styles.buttonContainer}>
29       <Button title="Toggle Layout" onPress={toggleBox} />
30     </View>
31     <View
32       style={[styles.box, boxPosition === "left" ? null : styles.moveRight]}
33     />
34   </View>
35 );
36 };
37
38 const styles = StyleSheet.create({
39   container: {
40     flex: 1,
41     alignItems: "flex-start",
42     justifyContent: "center",
43   },
44   box: {
45     height: 100,
46     width: 100,
47     borderRadius: 5,
48     margin: 8,
49     backgroundColor: "blue",
50   },
51   moveRight: {
52     alignSelf: "flex-end",
53     height: 200,
54     width: 200,
55   },
56   buttonContainer: {
57     alignSelf: "center",
58   },
59 });
60
61 export default App;

```

[cf. anim-4.mp4]

L'intégralité du code d'exécution de l'animation se résume en une seule ligne d'instruction : `LayoutAnimation.configureNext(LayoutAnimation.Presets.spring);`.

Ici, on utilise une pré-configuration qui donne l'effet « ressort », comme on peut le constater en essayant ce code. Il en existe trois pré-embarquées dans React Native :

- `easeInEaseOut` : exécute l'animation de plus en plus vite avant de ralentir en terminant l'animation
- `linear` : exécute l'animation de manière linéaire
- `spring` : exécute l'animation et donne un effet de rebond à la fin de son exécution

Si jamais ça ne suffisait pas, on peut utiliser `create` pour personnaliser l'animation. Par exemple, ici, nous prenons une animation de lissage à laquelle on donne un temps d'exécution de deux secondes.

```
1 LayoutAnimation.configureNext(
2   LayoutAnimation.create(2000, LayoutAnimation.Types.easeIn)
3 );
```

### Qu'est-ce qui déclenche l'animation ?

L'instruction `configureNext` sert à indiquer qu'à la prochaine mise à jour d'un état (via le setter d'un hook `useState`, dans notre cas), il faudra animer l'interface entre l'état précédent et le nouveau. Ici, le moteur de React Native est suffisamment intelligent pour déterminer les nouvelles positions des éléments et donc animer automatiquement, selon notre règle, les éléments. Pratique, n'est-ce pas ?

#### Syntaxe À retenir

- L'API `LayoutAnimation` est un outil extrêmement puissant qui permet de réaliser des animations entre différents états très simplement. C'est une solution à privilégier pour insérer rapidement des animations dans ses écrans.
- Par défaut, elle vient avec trois *presets* d'animation ; cependant, on peut personnaliser ces derniers ou créer nos propres *presets* grâce aux outils mis à disposition.

#### Complément

- <https://reactnative.dev/docs/layoutanimation>

## VII. Exercice : Appliquez la notion

### Question

[solution n°3 p.22]

Reprenons notre liste de cartes de l'exercice précédent (en indice ci-dessous) et ajoutons deux boutons : un pour supprimer une carte, et un pour en ajouter une. Les données sont rendues dans un composant `FlatList` de React Native.

À chaque modification, il faut animer avec un effet `LayoutAnimation.Presets.spring`, qui fera un fondu.

Le rendu escompté dans l'exemple ci-dessous :

[cf. demo-exercice-3.mp4]

### Indice :

Le code de la solution de l'exercice précédent :

```
1 import React, { useRef, useEffect } from "react";
2 import { Animated, View, Text, StyleSheet } from "react-native";
3
4 const App = () => {
5   return (
6     <View
7       style={{ flex: 1, justifyContent: "space-around", alignItems: "center" }}
8     >
9       <ArticleCard
10         isNew
11         title="Un titre comme les autres"
12         previewText="lorem ipsum dolor vegata blablblfdsfs fsfsdf hdhfgh sdf shggsrfsdgs fdg
13         gdgdhdfh"
14       />
15       <ArticleCard
16         title="Un titre comme les autres"
17         previewText="lorem ipsum dolor vegata blablblfdsfs fsfsdf hdhfgh sdf shggsrfsdgs fdg
```

```

18     gdgdhdfh"
19   />
20   <ArticleCard
21     isNew
22     title="Un titre comme les autres"
23     previewText="lorem ipsum dolor vegata blablblfdsfs fsfsdf hdhfgf sdf shggsrfsdgs fdg
24     gdgdhdfh"
25   />
26 </View>
27 );
28 };
29
30 export function ArticleCard(props) {
31   const newPostAnimatedValue = useRef(new Animated.Value(0)).current;
32
33   useEffect(() => {
34     const animation = Animated.loop(
35       Animated.sequence([
36         Animated.timing(newPostAnimatedValue, {
37           toValue: 1,
38           duration: 2000,
39           useNativeDriver: true,
40         }),
41         Animated.timing(newPostAnimatedValue, {
42           toValue: 0,
43           duration: 2000,
44           useNativeDriver: true,
45         }),
46       ])
47     );
48
49     // Si c'est un type new on affiche une animation
50     if (props.isNew) {
51       animation.start();
52     }
53
54     // On nettoie l'animation
55     return () => {
56       animation.stop();
57     };
58   }, [props.isNew]);
59
60   return (
61     <View style={styles.card}>
62       <Text style={styles.title}>{props.title}</Text>
63       <Text style={styles.text}>{props.previewText}</Text>
64       {props.isNew ? (
65         <Animated.Text style={[{ opacity: newPostAnimatedValue }, styles.new]}>
66           Nouveau
67         </Animated.Text>
68       ) : null}
69     </View>
70   );
71 }
72
73 const styles = StyleSheet.create({
74   card: {
75     position: "relative",

```

```
76   backgroundColor: "white",
77   borderWidth: 0.5,
78   borderColor: "#d3d3d3",
79   elevation: 5,
80   shadowOpacity: 0.0015 * 5 + 0.18,
81   shadowRadius: 0.54 * 5,
82   shadowOffset: {
83     height: 0.6 * 5,
84   },
85   width: 300,
86   padding: 20,
87 },
88 title: {
89   fontSize: 24,
90 },
91 text: {
92   marginTop: 10,
93 },
94 new: {
95   position: "absolute",
96   top: 10,
97   right: 10,
98   fontWeight: "400",
99   color: "green",
100 },
101 });
102
103 export default App;
```

## VIII. Essentiel

On peut aller très vite et utiliser `LayoutAnimation` qui anime l'interface sur un changement d'état, ou bien on peut partir sur du très spécifique avec une grande liberté, en utilisant l'API `Animated` et les différentes fonctions de lissage disponibles.

Grâce à cette dernière, on peut également composer les animations entre elles pour créer des boucles ou des enchaînements parallèles.

## IX. Auto-évaluation

### A. Exercice final

#### Exercice 1

[solution n°4 p.25]

Exercice

Combien d'API sont mises à disposition par React Native pour gérer les animations ?

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4

Exercice

De quelle manière peut-on récupérer les dimensions de l'écran ?

- ☐ En utilisant l'API `Dimensions` de React Native
- ☐ En utilisant `View.getDimension()`
- ☐ En utilisant `useWindowDimensions`

Exercice

La propriété `useNativeDriver` est-elle vivement recommandée, ou optionnelle ?

- ☐ Vivement recommandée
- ☐ Optionnelle

Exercice

Une animation...

- ☐ Démarre toute seule
- ☐ Doit être démarrée grâce à `start()`
- ☐ Doit être démarrée grâce à `launch()`

Exercice

La fonction `interpolate`...

- ☐ Peut recevoir un nombre différent de valeurs en entrée et sortie
- ☐ Doit recevoir le même nombre de valeurs en entrée et sortie
- ☐ S'arrête à la plage définie, bien que la valeur d'entrée grandisse

Exercice

Pour effectuer un effet rebond, j'utilise...

- ☐ `Easing.rebound`
- ☐ Rien, cela se fait par défaut
- ☐ `Easing.bounce`

Exercice

Pour faire boucler une animation, j'utilise...

- ☐ `Animated.loop`
- ☐ `Animated.loopAnimation`
- ☐ Je passe le paramètre `true` à `start(true)`

Exercice

Que fait la fonction `configureNext` de `LayoutAnimation` ?

- ☐ Elle prépare une animation que l'on peut lancer manuellement avec `start()`
- ☐ Elle prévient le moteur qu'il devra déclencher une animation pour le prochain changement d'état
- ☐ Elle déclenche une animation aléatoire par défaut



### Exercice

Pour enchaîner des animations à la suite, j'utilise...

- ☐ `Animated.sequence`
- ☐ `Animated.queue`
- ☐ `Animated.join`

### B. Exercice : Défi

Maintenant que nous maîtrisons les animations, créons une petite animation défi.

#### Question

[solution n°5 p.27]

L'objectif de cet exercice est de créer une barre de chargement personnalisée qui avance de 5 % chaque seconde. Pour ce faire, on pourrait être tenté d'utiliser `setInterval`, mais il est recommandé d'utiliser un package comme `use-interval`.

Le package est disponible ici<sup>1</sup> et on peut également consulter une explication du pourquoi ici.<sup>2</sup>

La barre indique aussi de façon textuelle le pourcentage d'avancement du chargement.

Et ci-dessous un exemple du rendu escompté :

[cf. demo-exercice-final.mp4]

#### Indice :

Le plus simple est de créer une boîte qui servira à représenter la barre de chargement vide et de positionner en absolu une autre boîte par-dessus qui charge progressivement en augmentant sa largeur.

## Solutions des exercices

---

1 <https://www.npmjs.com/package/use-interval>

2 <https://overreacted.io/making-setinterval-declarative-with-react-hooks/>



## p.7 Solution n°1

```

1 import * as React from "react";
2 import { Animated, Button } from "react-native";
3
4 export default function App() {
5   // On définit une référence de valeur animée qui commence à 1
6   const animatedValue = React.useRef(new Animated.Value(1)).current;
7   const opacityAnimation = React.useRef(
8     Animated.timing(animatedValue, {
9       toValue: 0,
10      duration: 5000, // Durant 5 secondes
11      useNativeDriver: true, // opacity accepte le driver natif
12    })
13  ).current;
14
15  return (
16    <Animated.View
17      style={{
18        opacity: animatedValue,
19        width: "100%",
20        height: "100%",
21        backgroundColor: "blue",
22      }}
23    >
24      <Button
25        title="Disparition en fondu"
26        onPress={() => opacityAnimation.start()}
27      />
28      <Button title="STOP" onPress={() => opacityAnimation.stop()} />
29    </Animated.View>
30  );
31 }
32

```

## p.10 Solution n°2

```

1 import React, { useRef, useEffect } from "react";
2 import { Animated, View, Text, StyleSheet } from "react-native";
3
4 const App = () => {
5   return (
6     <View
7       style={{ flex: 1, justifyContent: "space-around", alignItems: "center" }}
8     >
9       <ArticleCard
10        isNew
11        title="Un titre comme les autres"
12        previewText="lorem ipsum dolor vegata blablblfdsfs fsfsdf hdhfhg sdf shggsrfsdgs fdg
13          gdgdhdfh"
14      />
15       <ArticleCard
16        title="Un titre comme les autres"
17        previewText="lorem ipsum dolor vegata blablblfdsfs fsfsdf hdhfhg sdf shggsrfsdgs fdg
18          gdgdhdfh"

```

```

19     />
20     <ArticleCard
21       isNew
22       title="Un titre comme les autres"
23       previewText="lorem ipsum dolor vegata blablblfdsfs fsfsdf hdhfgh sdf shggsrfsdgs fdg
24       gdgdhdfh"
25     />
26   </View>
27 );
28 };
29
30 export function ArticleCard(props) {
31   const newPostAnimatedValue = useRef(new Animated.Value(0)).current;
32
33   useEffect(() => {
34     const animation = Animated.loop(
35       Animated.sequence([
36         Animated.timing(newPostAnimatedValue, {
37           toValue: 1,
38           duration: 2000,
39           useNativeDriver: true,
40         }),
41         Animated.timing(newPostAnimatedValue, {
42           toValue: 0,
43           duration: 2000,
44           useNativeDriver: true,
45         }),
46       ])
47     );
48
49     // Si c'est un type new on affiche une animation
50     if (props.isNew) {
51       animation.start();
52     }
53
54     // On nettoie l'animation
55     return () => {
56       animation.stop();
57     };
58   }, [props.isNew]);
59
60   return (
61     <View style={styles.card}>
62       <Text style={styles.title}>{props.title}</Text>
63       <Text style={styles.text}>{props.previewText}</Text>
64       {props.isNew ? (
65         <Animated.Text style={[{ opacity: newPostAnimatedValue }, styles.new]}>
66           Nouveau
67         </Animated.Text>
68       ) : null}
69     </View>
70   );
71 }
72
73 const styles = StyleSheet.create({
74   card: {
75     position: "relative",
76     backgroundColor: "white",

```

```

77     borderWidth: 0.5,
78     borderColor: "#d3d3d3",
79     elevation: 5,
80     shadowOpacity: 0.0015 * 5 + 0.18,
81     shadowRadius: 0.54 * 5,
82     shadowOffset: {
83       height: 0.6 * 5,
84     },
85     width: 300,
86     padding: 20,
87   },
88   title: {
89     fontSize: 24,
90   },
91   text: {
92     marginTop: 10,
93   },
94   new: {
95     position: "absolute",
96     top: 10,
97     right: 10,
98     fontWeight: "400",
99     color: "green",
100  },
101  });
102
103  export default App;

```

Afin de réaliser cet exercice, il est nécessaire de définir l'affichage d'un élément de la liste dans un composant spécifique. Ce dernier recevra en props les informations à afficher, qui contiendront un booléen, `isNew`, indiquant s'il s'agit d'une nouvelle carte. Ce n'est que si cette valeur vaut `true` que l'animation sera lancée :

```

1  export function ArticleCard(props) {
2    const newPostAnimatedValue = useRef(new Animated.Value(0)).current;
3
4    useEffect(() => {
5      const animation = Animated.loop(
6        Animated.sequence([
7          Animated.timing(newPostAnimatedValue, {
8            toValue: 1,
9            duration: 2000,
10           useNativeDriver: true,
11         }),
12         Animated.timing(newPostAnimatedValue, {
13           toValue: 0,
14           duration: 2000,
15           useNativeDriver: true,
16         }),
17       ])
18     });
19
20     // Si c'est un type new on affiche une animation
21     if (props.isNew) {
22       animation.start();
23     }
24
25     // On nettoie l'animation
26     return () => {
27       animation.stop();

```

```

28   };
29   }, [props.isNew]);
30
31   return (
32     <View style={styles.card}>
33       <Text style={styles.title}>{props.title}</Text>
34       <Text style={styles.text}>{props.previewText}</Text>
35       {props.isNew ? (
36         <Animated.Text style={[{ opacity: newPostAnimatedValue }, styles.new]}>
37           Nouveau
38         </Animated.Text>
39       ) : null}
40     </View>
41   );
42 }

```

Pour l'exécution de cette animation, une valeur initiée à 0 par l'utilisation du hook `useRef` va croître tout d'abord jusqu'à 1, avant de diminuer pour revenir à 0. Cette valeur servira à déterminer l'opacité de l'élément, ce qui le fera clignoter. Pour obtenir cet effet, il est tout d'abord nécessaire de définir une séquence d'animation avec `Animated.sequence`, puis de répéter cette séquence dans une boucle avec `Animated.loop`.

```

1 const animation = Animated.loop(
2   Animated.sequence([
3     Animated.timing(newPostAnimatedValue, {
4       toValue: 1,
5       duration: 2000,
6       useNativeDriver: true,
7     }),
8     Animated.timing(newPostAnimatedValue, {
9       toValue: 0,
10      duration: 2000,
11      useNativeDriver: true,
12    }),
13   ])
14 );

```

### p. 13 Solution n°3

```

1 import React, { useRef, useEffect } from "react";
2 import {
3   Animated,
4   View,
5   Button,
6   Text,
7   StyleSheet,
8   LayoutAnimation,
9   FlatList,
10 } from "react-native";
11 import Constants from "expo-constants";
12
13 if (
14   Platform.OS === "android" &&
15   UIManager.setLayoutAnimationEnabledExperimental
16 ) {
17   UIManager.setLayoutAnimationEnabledExperimental(true);
18 }
19

```

```

20 const App = () => {
21   const [cards, setCards] = React.useState([]);
22
23   return (
24     <React.Fragment>
25       <Button
26         style={{ marginTop: Constants.statusBarHeight }}
27         title="Ajouter un élément"
28         onPress={() => {
29           // On configure la prochaine animation
30           LayoutAnimation.configureNext(LayoutAnimation.Presets.spring);
31           setCards((previousCards) =>
32             previousCards.concat([
33               {
34                 isNew: true,
35                 title: "New card" + Math.random(),
36                 previewText:
37                   "lorem ipsum dolor vegata blablblfdfsfs fsfsdf hdhfhg sdf shggsrfsdgs
38                   fdggdgdhdfh",
39               },
40             ])
41           );
42         }}
43       />
44       <Button
45         title="Supprimer le dernier élément"
46         onPress={() => {
47           // On configure la prochaine animation
48           LayoutAnimation.configureNext(LayoutAnimation.Presets.spring);
49           setCards((previousCards) => previousCards.slice(0, -1));
50         }}
51       />
52       <FlatList
53         data={cards}
54         keyExtractor={(item) => item.title}
55         renderItem={({ item }) => (
56           <ArticleCard
57             isNew={item.isNew}
58             title={item.title}
59             previewText={item.previewText}
60           />
61         )}
62         contentContainerStyle={{ alignItems: "center" }}
63       />
64     </React.Fragment>
65   );
66 };
67
68 export function ArticleCard(props) {
69   const newPostAnimatedValue = useRef(new Animated.Value(0)).current;
70
71   useEffect(() => {
72     const animation = Animated.loop(
73       Animated.sequence([
74         Animated.timing(newPostAnimatedValue, {
75           toValue: 1,
76           duration: 2000,
77           useNativeDriver: true,

```

```

77     }},
78     Animated.timing(newPostAnimatedValue, {
79       toValue: 0,
80       duration: 2000,
81       useNativeDriver: true,
82     }},
83   ])
84 );
85
86 // Si c'est un type new on affiche une animation
87 if (props.isNew) {
88   animation.start();
89 }
90
91 // On nettoie l'animation
92 return () => {
93   animation.stop();
94 };
95 }, [props.isNew]);
96
97 return (
98   <View style={styles.card}>
99     <Text style={styles.title}>{props.title}</Text>
100     <Text style={styles.text}>{props.previewText}</Text>
101     {props.isNew ? (
102       <Animated.Text style={[{ opacity: newPostAnimatedValue }, styles.new]}>
103         Nouveau
104       </Animated.Text>
105     ) : null}
106   </View>
107 );
108 }
109
110 const styles = StyleSheet.create({
111   card: {
112     position: "relative",
113     backgroundColor: "white",
114     borderWidth: 0.5,
115     borderColor: "#d3d3d3",
116     elevation: 5,
117     shadowOpacity: 0.0015 * 5 + 0.18,
118     shadowRadius: 0.54 * 5,
119     shadowOffset: {
120       height: 0.6 * 5,
121     },
122     width: 300,
123     padding: 20,
124   },
125   title: {
126     fontSize: 24,
127   },
128   text: {
129     marginTop: 10,
130   },
131   new: {
132     position: "absolute",
133     top: 10,
134     right: 10,

```




```
135     fontWeight: "400",  
136     color: "green",  
137   },  
138 });  
139  
140 export default App;
```

### Exercice p. 15 Solution n°4

#### Exercice

Combien d'API sont mises à disposition par React Native pour gérer les animations ?


- ☐ 1
- ☒ 2
- ☐ 3
- ☐ 4

 Les deux API sont `Animated` et `LayoutAnimation`.

#### Exercice


De quelle manière peut-on récupérer les dimensions de l'écran ?

- ☒ En utilisant l'API `Dimensions` de React Native
- ☐ En utilisant `View.getDimension()`
- ☒ En utilisant `useWindowDimensions`

 La seconde n'existe pas, les autres servent à cela.


#### Exercice

La propriété `useNativeDriver` est-elle vivement recommandée, ou optionnelle ?

- ☒ Vivement recommandée
  - ☐ Optionnelle
-  En effet, l'interpréteur lèvera un avertissement si elle est manquante.


#### Exercice

Une animation...

- ☐ Démarre toute seule
  - ☒ Doit être démarrée grâce à `start()`
  - ☐ Doit être démarrée grâce à `launch()`
-  Une animation a besoin d'être déclenchée manuellement grâce à `start()`.


### Exercice

La fonction `interpolate`...

- ☐ Peut recevoir un nombre différent de valeurs en entrée et sortie
- ☒ Doit recevoir le même nombre de valeurs en entrée et sortie
- ☐ S'arrête à la plage définie, bien que la valeur d'entrée grandisse
-  On doit spécifier le même nombre de valeurs en entrée et sortie. Par contre, si on n'utilise pas `extrapolate: "clamp"`, la fonction d'interpolation pourra générer des comportements étranges, car elle n'aura pas de limite. C'est surtout vrai avec une interpolation de couleurs ou de rotation en degrés.


### Exercice

Pour effectuer un effet rebond, j'utilise...

- ☐ `Easing.rebound`
- ☐ Rien, cela se fait par défaut
- ☒ `Easing.bounce`
-  Par défaut, l'animation est linéaire et l'effet rebond s'obtient grâce à `easing.bounce`.


### Exercice

Pour faire boucler une animation, j'utilise...

- ☒ `Animated.loop`
- ☐ `Animated.loopAnimation`
- ☐ Je passe le paramètre `true` à `start(true)`
-  On ne peut pas passer de booléen à `start`, et `loopAnimation` n'existe pas.

### Exercice

Que fait la fonction `configureNext` de `LayoutAnimation` ?

- ☐ Elle prépare une animation que l'on peut lancer manuellement avec `start()`
- ☒ Elle prévient le moteur qu'il devra déclencher une animation pour le prochain changement d'état
- ☐ Elle déclenche une animation aléatoire par défaut
-  En effet, elle déclenchera une animation uniquement au prochain changement d'état.

### Exercice

Pour enchaîner des animations à la suite, j'utilise...

- ☒ `Animated.sequence`
- ☐ `Animated.queue`
- ☐ `Animated.join`

Q La seule réponse existante est la première.

### p. 17 Solution n°5

```

1 import React, { useRef, useState, useEffect } from "react";
2 import { Text, View, StyleSheet, Animated } from "react-native";
3 import useInterval from "use-interval";
4
5 const App = () => {
6   let animation = useRef(new Animated.Value(0));
7   const [progress, setProgress] = useState(0);
8   useInterval(() => {
9     if (progress < 100) {
10       setProgress((previousProgress) => previousProgress + 5);
11     }
12   }, 1000);
13
14   useEffect(() => {
15     Animated.timing(animation.current, {
16       toValue: progress,
17       duration: 100,
18       useNativeDriver: false, // Width n'est pas supporté par le driver natif
19     }).start();
20   }, [progress]);
21
22   const width = animation.current.interpolate({
23     inputRange: [0, 100],
24     outputRange: ["0%", "100%"],
25     extrapolate: "clamp", // Permet d'éviter que la fonction extrapole aussi la plage de
26     // sortie si la valeur d'entrée dépasse 100,
27   });
28   return (
29     <View style={styles.container}>
30       <Text>Chargement en cours..</Text>
31       <View style={styles.progressBar}>
32         <Animated.View
33           style={
34             ([StyleSheet.absoluteFill], { backgroundColor: "#8BED4F", width })
35           >
36         </View>
37         <Text>`${progress}%`</Text>
38       </View>
39     );
40   };
41
42   export default App;
43
44   const styles = StyleSheet.create({
45     container: {
46       flex: 1,
47       justifyContent: "center",
48       alignItems: "center",
49       backgroundColor: "#ecf0f1",
50       padding: 8,
51     },

```

```
52 progressBar: {
53   flexDirection: "row",
54   height: 20,
55   width: "100%",
56   backgroundColor: "white",
57   borderColor: "#000",
58   borderWidth: 2,
59   borderRadius: 5,
60 },
61 });
```

Afin de créer une barre de chargement personnalisée, il va falloir faire varier conjointement l'avancement du remplissage de la barre et la valeur de cet avancement sous la forme d'un pourcentage.

Pour imaginer une solution, il faut entendre « avancement du remplissage de la barre » comme une variation de la largeur d'un élément coloré. En effet, il s'agit ni plus ni moins de positionner un rectangle de couleur, contenu dans un élément avec des bordures apparentes.

```
1 <View style={styles.container}>
2   <Text>Chargement en cours.</Text>
3   <View style={styles.progressBar}>
4     <Animated.View
5       style={
6         ([StyleSheet.absoluteFill], { backgroundColor: "#8BED4F", width })
7       }
8     />
9   </View>
10  <Text>`${progress}%`</Text>
11 </View>
12 [...]
13 const styles = StyleSheet.create({
14   container: {
15     flex: 1,
16     justifyContent: "center",
17     alignItems: "center",
18     backgroundColor: "#ecf0f1",
19     padding: 8,
20   },
21   progressBar: {
22     flexDirection: "row",
23     height: 20,
24     width: "100%",
25     backgroundColor: "white",
26     borderColor: "#000",
27     borderWidth: 2,
28     borderRadius: 5,
29   },
30 });
```

Ici, une première `<View>` parent est positionnée au centre de l'écran, puis une vue dont les bordures sont visibles et occupant 100 % de la largeur du conteneur est ajoutée. Une vue animée, `<Animated.View>`, configurée en position absolue et remplissant entièrement son parent grâce à `[StyleSheet.absoluteFill]`, est alors créée. C'est la largeur de cette vue qui va varier avec une animation : sa largeur sera égale à la valeur de `width` tout au long de l'animation.

Le texte symbolisant l'état d'avancement du chargement prendra quant à lui la valeur de `progress` à chacune de ses itérations.

L'évolution de la valeur de `progress` se fait de 5 en 5 toutes les secondes jusqu'à atteindre 100.

```
1 const [progress, setProgress] = useState(0);
2 useInterval(() => {
3   if (progress < 100) {
4     setProgress((previousProgress) => previousProgress + 5);
5   }
6 }, 1000);
```

La largeur de la barre de chargement sera stockée dans une variable initiée à 0 grâce au hook `useRef` :

```
1 let animation = useRef(new Animated.Value(0));
```

À chaque modification de la valeur de `progress`, une animation faisant varier la valeur de `animation` jusqu'à ce qu'elle atteigne `progress` est lancée :

```
1 useEffect(() => {
2   Animated.timing(animation.current, {
3     toValue: progress,
4     duration: 100,
5     useNativeDriver: false, // Width n'est pas supporté par le driver natif
6   }).start();
7 }, [progress]);
```

Une interpolation de la valeur numérique de `animation` vers une valeur en pourcentage se fait grâce à :

```
1 const width = animation.current.interpolate({
2   inputRange: [0, 100],
3   outputRange: ["0%", "100%"],
4   extrapolate: "clamp", // Permet d'éviter que la fonction n'extrapole aussi la plage de
5   // sortie si la valeur d'entrée dépasse 100,
6 });
```

C'est au final cette valeur textuelle qui est utilisée pour animer la largeur de la barre de chargement.