

# Créer des animations dans Flutter

# Table des matières

<b>I. b.a.-ba de l'animation Flutter</b>	<b>3</b>
A. Une animation Flutter, c'est quoi ? .....	3
B. Animations « drawing-based » .....	3
C. Animations « code-based » .....	5
<b>II. Exercice : Quiz</b>	<b>7</b>
<b>III. Animations implicites</b>	<b>7</b>
A. Animations implicites avec « built-in » : les AnimatedFoo .....	7
B. TweenAnimationBuilder .....	8
<b>IV. Exercice : Quiz</b>	<b>9</b>
<b>V. Animations explicites</b>	<b>10</b>
A. Animations explicites avec « built-in » : les FooTransition.....	10
B. AnimatedWidget .....	11
C. AnimatedBuilder .....	12
<b>VI. Exercice : Quiz</b>	<b>13</b>
<b>VII. Essentiel</b>	<b>14</b>
<b>VIII. Auto-évaluation</b>	<b>14</b>
A. Exercice : .....	14
B. Test .....	15
<b>Solutions des exercices</b>	<b>15</b>

## I. b.a.-ba de l'animation Flutter

### Contexte

Les animations sont de véritables atouts pour vos projets, puisqu'ils améliorent considérablement l'expérience utilisateur en rendant l'utilisation plus vivante et intuitive. Et ça tombe bien, car Flutter vous permet de créer vos premières animations en un temps record.

Dans ce cours, nous allons vous apprendre à créer cet effet « wow » en seulement quelques lignes de code, tout en vous donnant les outils pour créer des animations plus poussées et personnalisées.

Pour ce faire, nous allons commencer par le b.a.-ba de l'animation dans Flutter ; en apprenant notamment à différencier les animations « *drawing based* » et les « *code-based* », avant d'en aborder les deux principaux types : les animations implicites et explicites.

Tout au long de ce cours, vous aurez la possibilité d'expérimenter par vous-même ces différents types grâce à des exemples guidés, et de voir pour la première fois vos listes et containers se mouvoir.

### A. Une animation Flutter, c'est quoi ?

#### Rappel Le state dans Flutter

Le *State* correspond à la logique et à l'état interne d'un *StatefulWidget*. Il s'agit d'une information qui peut être lue de façon synchrone quand le widget est construit et qui peut changer pendant la durée de vie du widget. C'est grâce au state que vous pouvez reconstruire en permanence vos widgets et leur donner vie !

#### Définition

Une animation peut être définie de plusieurs façons. Cela peut signifier « *mettre de la vivacité* » dans quelque chose. Toutefois, une seconde définition peut nous apporter davantage d'indices sur la technique se cachant derrière une animation. Il s'agit également de « *toute méthode consistant à filmer image par image des dessins qui paraîtront animés sur l'écran* » (Larousse, 2021).

En résumé, une animation est une succession d'images à intervalles réguliers, permettant de donner vie à un ou plusieurs éléments.

Pour créer une animation, nous avons donc besoin d'un outil nous permettant de mettre des images à la suite les unes des autres, ainsi que d'images. En réalité, une animation dans Flutter correspond à une multitude d'états (ou « *states* ») d'un widget qui se succèdent quelque 60 ou 90 fois par seconde ! Petit bonus : Cela vous permet d'avoir des intégrations beaucoup plus performantes qu'en utilisant directement des vidéos par exemple.

### B. Animations « drawing-based »

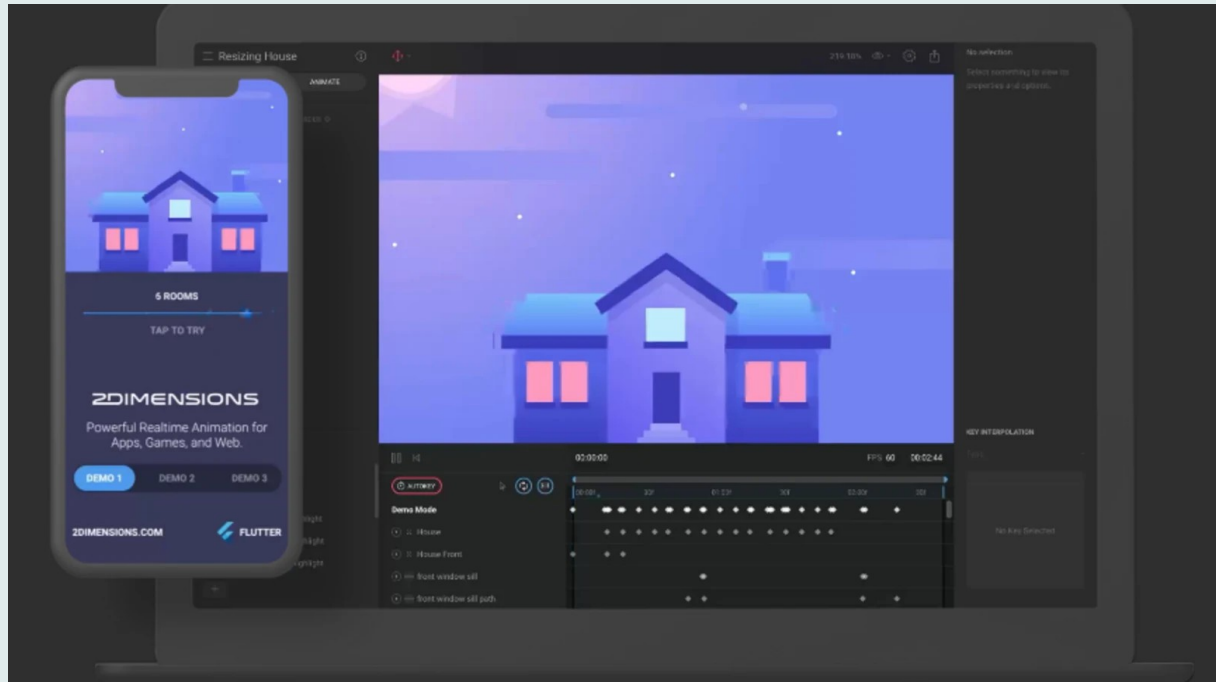
#### Définition

Une animation « *drawing-based* » est une animation consistant en l'animation d'un dessin. Ce sont des animations complexes, aussi bien en termes de design qu'en termes d'intégration.

## Fondamental

Nous pouvons différencier 2 types d'animations « *drawing-based* » : les animations « *bas-niveau* » et les animations basées sur des bibliothèques tierces.

Les animations « *bas-niveau* » sont des animations complexes retranscrites en code, grâce à la classe `CustomPainter`. Même si cela vous permet de créer une infinité d'animations très personnalisées, vous pourriez vous retrouver limités dans la réalisation d'animations ultra-spécifiques et complexes.



### Rive - Un des outils permettant de créer des animations

Les animations utilisant des bibliothèques tierces sont créées grâce à un outil externe puis importées dans Flutter. Cet outil est généralement une interface visuelle sur laquelle nous pouvons dessiner et animer sans nécessairement passer par du code.

Par la suite, il est parfois possible d'y apporter quelques modifications en code après l'importation dans notre projet Flutter ; en faisant varier par exemple la vitesse de succession des différentes images, ou la couleur de certains éléments.

**Exemple**

Source : Dribble - Planet Rotation Animation par Inside of Motion<sup>1</sup>

**Remarque**

Ce type d'animations est plus le plus compliqué et ne sera pas abordé dans la suite de ce cours. Pour les plus téméraires ou les motion designers en herbe, nous vous invitons à explorer la classe CustomPainter<sup>2</sup> et les librairies tierces telles que Rive<sup>3</sup> ou Lottie<sup>4</sup> !

**C. Animations « code-based »****Définition**

Une animation « *code-based* » est l'animation de widgets communs tels que des containers, des boutons, la couleur d'un texte, etc.

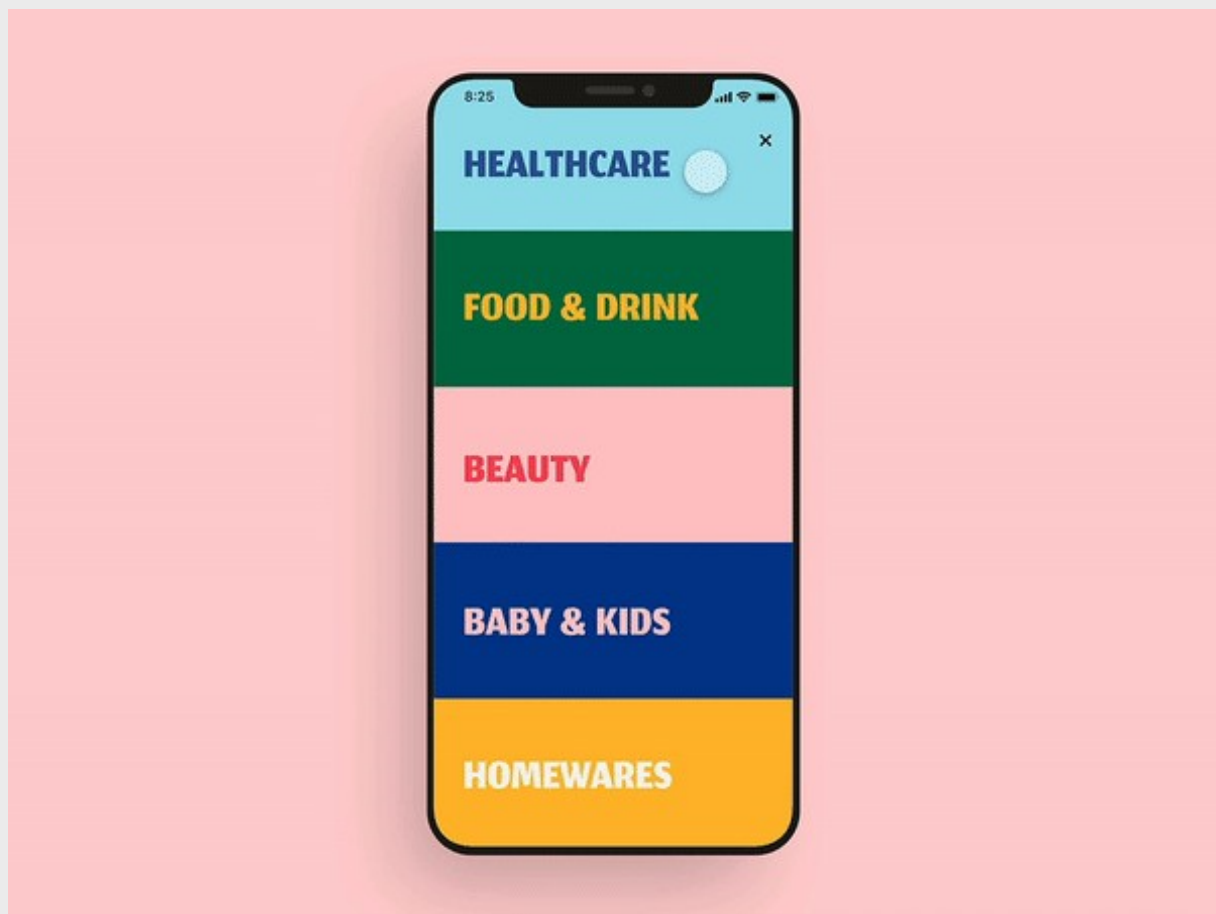
<sup>1</sup> <https://dribbble.com/shots/16104381-Planet-Rotation-Animation>

<sup>2</sup> <https://api.flutter.dev/flutter/rendering/CustomPainter-class.html>

<sup>3</sup> <https://rive.app/>

<sup>4</sup> <https://lottiefiles.com/>

**Exemple**



Menu UI animation by Interface Market

**Fondamental**

Il existe 2 principaux types d'animations « *code-based* » : le type implicite et le type explicite.

Les animations implicites sont créées grâce à des widgets qui gèrent déjà certaines animations pour vous, car ils implémentent la classe `ImplicitlyAnimatedWidget`. Il s'agit d'une classe qui anime les changements de propriétés. Par exemple, si un `AnimatedContainer` voit sa propriété largeur changer, la reconstruction ne se fera pas en un coup comme habituellement, mais en plusieurs fois afin de créer une animation qui se charge de la transition entre l'état initial et le nouvel état.

Les animations explicites sont, quant à elles, un ensemble de controllers qui indiquent à Flutter comment reconstruire rapidement l'arbre de widgets pour créer des animations.

Pour choisir le type adapté à votre projet, il y a une série de questions à se poser :

- Mon animation ne se répète pas en boucle. Vrai, faux ?
- Mon animation ne comprend pas de discontinuités dans les valeurs d'animation. Vrai, faux ?
- Mon animation est composée d'un seul child. Vrai, faux ?

Si toutes ces affirmations sont vraies concernant votre projet, vous pourrez vous tourner vers une animation implicite. Si, au contraire, au moins une affirmation est fausse, il faudra vous tourner vers une animation explicite.

**Remarque**

Les animations « *code-based* » sont les plus utilisées, et permettent de créer des effets en un temps record !

## Exercice : Quiz

### Question 1

Les animations « *drawing-based* » sont :

- ☐ Simples d'utilisation
- ☐ Consistent en l'animation de widgets communs
- ☐ Peuvent être réalisées grâce à la classe `CustomPainter`

### Question 2

Je souhaite animer un `Container`, quel type d'animation est-ce ?

- ☐ Une animation « *code-based* »
- ☐ Une animation « *drawing-based* »

### Question 3

Quels sont les deux types d'animation « *code-based* » ?

- ☐ Le `CustomPainter`
- ☐ L'implicite
- ☐ Le `TweenAnimationBuilder`
- ☐ L'explicite

### Question 4

Je souhaite créer une animation se répétant en boucle, j'utilise :

- ☐ Une animation implicite
- ☐ Une animation explicite

### Question 5

Je souhaite créer une animation discontinue, que je peux arrêter et reprendre, j'utilise :

- ☐ Une animation implicite
- ☐ Une animation explicite

## II. Animations implicites

### A. Animations implicites avec « **built-in** » : les `AnimatedFoo`

#### Qu'est-ce qu'un `Foo` ?

Tout au long de ce cours, vous pourrez être amené à voir le mot « *Foo* » apparaître, comme dans `AnimatedFoo` ou `FooTransition`. A vrai dire, cela signifie que « *Foo* » peut être remplacé par certains widgets communs. Ainsi, `Positioned` devient `AnimatedPositioned` ou encore `PositionedTransition` !

### Fondamental

Flutter possède une série de widgets, qui sont simplement des versions animées de widgets déjà existants. Ces widgets créent une animation à chaque fois qu'une des propriétés du widget change et se nomment (entre autres) les `AnimatedFoo`.

#### Liste des `AnimatedFoo` :

- `AnimatedContainer`
- `AnimatedAlign`
- `AnimatedDefaultTextStyle`
- `AnimatedOpacity`
- `AnimatedPadding`
- `AnimatedPhysicalModel`
- `AnimatedPositioned`
- `AnimatedPositionedDirectional`

[cf. créer-animations-flutter-vid01]

Nous allons ensemble réaliser cette toute première animation à l'aide d'un `AnimatedContainer`. Pour ceci, rien de plus simple : il suffit de rajouter `Animated` devant notre `Container`.

```
1 AnimatedContainer(  
2   height: _bigger ? 300 : 500,  
3   width: _bigger ? 300 : 500, color: Colors.blue,  
4   duration: Duration(seconds: 3),  
5   curve: Curves.elasticInOut),
```

2 propriétés supplémentaires s'offrent alors à nous : `duration` et `curve`.

`Duration` va nous permettre de déterminer combien de temps le changement de propriété jusqu'à la nouvelle valeur va prendre. Ici, il faudra 3 secondes pour que la largeur et la hauteur passent de 300 à 500 et vice-versa.

`Curve` nous permet, par la suite, de rendre cette animation plus personnalisée en créant des variations supplémentaires de valeur au sein même du temps imparti par la propriété `duration`. C'est ainsi que nous pouvons obtenir cette animation non-linéaire. Je vous invite vivement à explorer les différents effets proposés par la classe `Curves`<sup>1</sup> ! Ici, nous avons utilisé l'effet « *elasticInOut* ».

## B. TweenAnimationBuilder

Il existe parfois des widgets que nous souhaitons animer, mais qui ne possèdent pas de version « *built-in* ». C'est là que le `TweenAnimationBuilder` rentre en scène !

### Fondamental

Le `TweenAnimationBuilder` vous permet d'animer tous types de widgets communs. Pour ce faire, il est nécessaire de notamment définir trois paramètres : `duration`, `tween` et `builder`.

[cf. créer-animations-flutter-vid02]

`Tween` est un paramètre permettant d'indiquer entre quelles valeurs notre animations doit varier, tandis que le `builder`, retourne ce à quoi notre widget doit ressembler à un moment donné.

Comme exemple, nous allons essayer d'animer un widget `ColorFiltered`, qui ne possède donc pas de version animée « *built-in* ». Pour que nous puissions voir au mieux les transitions de couleurs, nous appliquons le filtre à un container blanc simple.

<sup>1</sup> <https://api.flutter.dev/flutter/animation/Curves-class.html>



**Attention**

L'animation ne se répète pas comme pourrait le laisser penser ce gif. Elle ne se fait qu'une fois au moment du hot reload. Sinon, ce serait une animation explicite !

**Fondamental**

```
1 TweenAnimationBuilder<Color>(  
2 tween: ColorTween(begin: Colors.red, end: Colors.yellow), duration: Duration(seconds: 3),  
3 builder: (BuildContext _, Color value, Widget child) { return ColorFiltered(  
4 colorFilter: ColorFilter.mode(value, BlendMode.modulate), child: Container(  
5 color: Colors.white, height: 300, width: 300));  
6 },  
7 )
```

Ici, puisque la propriété que nous souhaitons faire varier est une couleur, nous utilisons un *ColorTween*. Celui-ci nous permet d'indiquer 2 couleurs : une couleur de début d'animation et une couleur de fin.

Dans le builder, il ne nous manque plus qu'à indiquer le widget qui prendra comme paramètre la couleur variable ! Ici, il s'agit donc du *ColorFiltered*, qui prend la valeur de la couleur fournie par le tween en second paramètre. Et voilà, le tour est joué !

Il existe une multitude de Tween pour tous types de variables : *AlignmentGeometryTween*, *ConstantTween*, *SizeTween*, *TextStyleTween*, etc.

## Exercice : Quiz

### Question 1

Quelles sont les deux paramètres ajoutés à un *AnimatedFoo* par rapport à sa version classique ?

- ☐ Duration et builder
- ☐ Builder et curve
- ☐ Duration et curve

### Question 2

Si je souhaite créer une animation non-linéaire, que puis-je utiliser ?

- ☐ Le paramètre duration
- ☐ Le paramètre curve
- ☐ La variation des propriétés de mon widget avec setState

### Question 3

La classe *TweenAnimationBuilder* peut être utilisée si :

- ☐ Mon animation implicite ne possède pas de version « *built-in* »
- ☐ Mon animation implicite possède une version « *built-in* »
- ☐ Mon animation est trop détaillée

### Question 4

Mon animation se répète à l'infini depuis le début ! Qu'est-ce que cela signifie ?

- ☐ C'est une animation implicite avec TweenAnimationBuilder
- ☐ C'est une animation explicite

#### Question 5

Quelle est la classe que je vais utiliser comme variable dans mon TweenAnimationBuilder ?

- ☐ Tween<T extends dynamic>
- ☐ AnimationBuilder
- ☐ AnimatedFoo

### III. Animations explicites

#### A. Animations explicites avec « built-in » : les FooTransition

##### **Rappel** Qu'est-ce qu'un controller ?

Un controller nous permet de suivre l'état d'un widget et de, comme son nom l'indique, le controller. Par exemple, il est possible grâce à ScrollController de comprendre lorsque nous avons atteint le bout d'une liste, et de lui demander de remonter au début. Ici, nous utiliserons principalement le AnimationController, qui n'aura bientôt plus de secrets pour vous !

Info utile : la valeur d'un controller varie entre 0 et 1, et dans le cas d'un AnimationController, 0 correspond au début de l'animation et 1 au moment où l'animation est terminée. Donc si le paramètre duration est de 3 secondes, le controller aura 0 de valeur à la seconde 0, et 1 de valeur à la seconde 3.

Comme nous l'avons vu plus tôt dans ce cours, il est d'ores et déjà possible de créer des animations qui vont et viennent entre leurs valeurs de départ et leur valeur finale. Mais comment faisons-nous si nous souhaitons créer des animations qui se répètent depuis le début, encore et encore ? Les animations explicites « *built-in* » sont là pour vous !

##### **Fondamental**

Les animations explicites avec « *built-in* » (ou FooTransition) sont des extensions des *AnimatedWidget* que nous étudierons par la suite.

[cf. creer-animations-flutter-vid03]

Les FooTransition comportent notamment :

- SizeTransition
- FadeTransition
- ScaleTransition
- SlideTransition
- RotationTransition
- PositionedTransition
- DecoratedBoxTransition
- DefaultTextStyleTransition
- RelativePosititonedTransition

Pour l'exemple, nous allons ici utiliser le SizeTransition pour donner l'effet d'une balle rebondissante à notre *Container*.

```
1 class MyStatefulWidget extends StatefulWidget { @override
```

```

2 State<MyStatefulWidget> createState() => _MyStatefulWidgetState();
3 }
4
5 class _MyStatefulWidgetState extends State<MyStatefulWidget> with
  SingleTickerProviderStateMixin {
6   AnimationController _controller; Animation _animation;
7
8   @override
9   void initState() { super.initState();
10    _controller = AnimationController( duration: const Duration(seconds: 3), vsync: this,
11    ).repeat();
12    _animation = CurvedAnimation( parent: _controller,
13    curve: Curves.bounceInOut,
14    );
15  }
16
17  @override void dispose() {
18    super.dispose();
19    _controller.dispose();
20  }
21
22  @override
23  Widget build(BuildContext context) {
24    ... }
25  }

```

L'utilisation d'un controller implique ici l'utilisation d'un *SingleTickerProviderStateMixin*. Il est ensuite nécessaire d'initialiser notre *AnimationController* dans *initState()*, en lui indiquant notamment le paramètre *duration* de notre animation, ainsi que *vsync*. *vsync* s'occupe de garder le suivi de l'écran, afin de ne pas continuer à générer l'animation lorsque celle-ci n'apparaît pas.

Nous initialisons ensuite notre animation en lui donnant comme parent le controller que nous venons de créer. Ainsi, nous n'avons plus qu'à dire au controller ce que nous souhaitons, et il l'appliquera à l'animation ! Ici, nous lui demandons de se répéter à l'infini grâce à *..repeat()*. *AnimationController* propose ainsi toutes sortes d'actions que nous pouvons appliquer à nos animations, comme *stop()* pour arrêter l'animation ou *forward()* pour ne jouer l'animation qu'une fois.

Attention, il ne faut pas oublier de disposer le controller dans *dispose()* pour éviter les fuites mémoire !

Maintenant, passons à la création du widget à qui s'appliquera l'animation. Pour ça, rien de plus simple : il faut créer notre widget *SizeTransition*, lui donner en *sizeFactor* notre animation (qui dépend donc déjà de notre controller) et mettre le child (ici *Center*).

```

1 Widget build(BuildContext context) { return Scaffold(
2   body: SizeTransition( sizeFactor: _animation, axis: Axis.vertical, axisAlignment: 1,
3   child: Center( child: ClipRRect(
4     borderRadius: BorderRadius.circular(100),
5     child: Container(height: 200, width: 200, color: Colors.red)),
6   ),
7   ),
8 );
9 }

```

## B. AnimatedWidget

Nous venons déjà d'étudier certains *AnimatedWidget* à travers des extensions déjà disponibles dans flutter. Mais si vous n'y avez pas encore trouvé votre bonheur, il est tout à fait possible de créer vos propres extensions de la classe *AnimatedWidget* !

**Fondamental**

Nous allons maintenant essayer de créer notre propre *AnimatedWidget* de la classe *Button*, puisque la classe « *AnimatedButton* » n'existe pas encore.

```
1 class ButtonTransition extends AnimatedWidget { @override
2   Widget build(BuildContext context) { (...)
3
4 }
5 }
```

Vous venez de créer votre propre *AnimatedButton* ! Toutefois, pour le moment, il ne fait pas grand chose. Pour y remédier, l'objectif est de faire varier la bordure extérieure de votre bouton. Pour ce faire, il est nécessaire de return un bouton (ici *OutlineButton*) mais aussi d'indiquer à votre *AnimatedWidget* à quel changement de paramètre il doit s'animer.

Ici, puisque nous souhaitons faire varier la largeur de la bordure, nous indiquons au widget que *width* est un *listenable*.

Ensuite, nous n'avons plus qu'à mettre *width* comme variable de largeur dans le paramètre *borderSide* du bouton, et le tour est joué !

```
1 class ButtonTransition extends AnimatedWidget {
2   const ButtonTransition({width} : super(listenable: width);
3   Animation<double> get width => listenable @override
4   Widget build(BuildContext context) { return OutlineButton(
5     child: Text('Appuie !'),
6     borderSide: BorderSide(width: width.value),
7   );
8 }
9 }
```

Maintenant, vous n'avez plus qu'à utiliser votre *AnimatedButton* comme n'importe quel autre *AnimatedWidget* « *built-in* » !

## C. AnimatedBuilder

**Définition**

Un *AnimatedBuilder* est utile pour les widgets plus complexes qui souhaitent inclure une animation comme partie d'une fonction plus large.

**Fondamental**

La création d'un *AnimatedBuilder* ressemble beaucoup à la création d'un *AnimatedWidget*, du moins dans les premières étapes : l'ajout d'un *SingleTickerProviderStateMixin*, l'initialisation d'un *controller*, etc.

Ici, nous allons faire tourner un *container* sur lui-même.

[cf. creer-animations-flutter-vid04]

```
1 class _MyStatefulWidgetState extends State<MyStatefulWidget> with
2   SingleTickerProviderStateMixin {
3   AnimationController _controller;
4
5   @override
6   void initState() { super.initState();
7     _controller = AnimationController( duration: const Duration(seconds: 5), vsync: this,
8   )..repeat();
9
10  @override void dispose() {
```

```
11 _controller.dispose(); super.dispose();
12 }
13
14 @override
15 Widget build(BuildContext context) { return AnimatedBuilder( animation: _controller,
16 child: Container(width: 250.0, height: 250.0, color: Colors.purple), builder: (BuildContext
17 context, Widget child) {
18 return Transform.rotate(
19 angle: _controller.value * 2.0 * math.pi, child: child,
20 );
21 );
22 }
23 }
```

Le builder a reçu en paramètre le child défini plus haut, à savoir le Container. Comme nous l'avons dit plus haut, la valeur d'un controller varie entre 0 et 1. Nous utilisons donc cette valeur pour savoir où en est le controller, mais aussi pour déterminer où doit en être la rotation de notre Container avec « *angle: \_controller.value \* 2.0 \* math.pi* ».

**Remarque**

Il est possible de faire à peu près les mêmes choses avec un *AnimatedBuilder* qu'avec un *AnimatedWidget*. Le type que vous choisirez sera plus une question de goût ! La plupart du temps, l'*AnimatedBuilder* peut vite représenter un certain nombre de lignes de codes, alors qu'un *AnimatedWidget* pourrait faire le même travail en moins de lignes.

## Exercice : Quiz

### Question 1

Si une animation dure 5 secondes, quelle valeur aura son controller à la seconde 5 ?

- ☐ 1
- ☐ 0
- ☐ 5

### Question 2

Si je souhaite animer le widget Positioned, que puis-je utiliser ?

- ☐ Un AnimatedPositioned
- ☐ Je crée mon propre AnimatedWidget
- ☐ J'utilise AnimatedBuilder

### Question 3

Qu'est-ce qu'un listenable ?

- ☐ C'est une classe qui prévient les clients lorsqu'une valeur a changé
- ☐ C'est une classe qui prévient le widget parent qu'une valeur a changé C'est un paramètre d'une fonction

### Question 4

Si je souhaite faire tourner un container sur lui-même en continu, que puis-je utiliser ?

- ☐ Un AnimatedWidget
- ☐ Un AnimatedBuilder
- ☐ Un TweenAnimationBuilder

Question 5

Faut-il mieux utiliser un AnimatedWidget ou un AnimatedContainer ?

- ☐ Tout dépend du projet et de la fonction dans lequel il s'intègre
- ☐ Un AnimatedWidget (c'est moins de lignes de code)
- ☐ Un AnimatedBuilder

## IV. Essentiel

- Les animations « *drawing-based* » sont des animations complexes et nécessitent très souvent des bibliothèques tierces.
- Les animations « *code-based* » sont les plus courantes et permettent d'animer les widgets communs.
- Les animations « *code-based* » se divisent en deux types : les animations implicites et les animations explicites.
- Les animations implicites sont les plus rapides à implémenter. Elles permettent d'animer un child unique, qui ne se répète pas en boucle, et n'est pas discontinue (ne s'arrête pas pour reprendre après etc.).
- Les animations implicites comprennent les AnimatedFoo (qui sont des « *built-in* ») et les TweenAnimationBuilder.
- Les animations explicites sont plus complexes à implémenter, mais permettent de créer des animations discontinues, avec plusieurs child ou qui se répètent à l'infini.
- Les animations explicites utilisent un AnimatedWidget (ou un AnimatedFoo « *built-in* ») ou un AnimatedBuilder.
- Les animations explicites nécessitent l'utilisation d'un controller, tandis que les animations implicites n'ont besoin que d'être dans un StatefulWidget.

## V. Auto-évaluation

### A. Exercice :

Maintenant que vous avez appris à créer des animations explicites et implicites, il est temps de le mettre en pratique !

#### Question

Nous allons simplement créer la superposition de 3 animations différentes dans une colonne :

- La première est un container vert carré de 100 x 100, qui tourne sur lui-même en permanence.
- La deuxième est un container rond, qui change de couleur une seule fois, au moment où le widget est créé.
- Le troisième est un container qui s'agrandit ou rétrécit uniquement lorsqu'un bouton est activé.

Toutes ses animations doivent se faire en 7 secondes.

Avant de vous lancer, prenez le temps de vous poser les bonnes questions ! Mon widget doit-il se relancer en permanence ? A-t-il des discontinuités ?

À vous de jouer !

**B. Test****Exercice : Quiz**

## Question 1

Quel type d'animation sera un container qui se déplace de droite à gauche puis recommence à l'infini ?

- ☐ Une animation « *drawing-based* » implicite
- ☐ Une animation « *drawing-based* » explicite
- ☐ Une animation « *code-based* » implicite
- ☐ Un AnimatedContainer

## Question 2

J'ai besoin d'animer un container seulement une fois lorsqu'un bouton est activé, comment faire au plus simple ?

- ☐ Utiliser AnimatedBuilder
- ☐ Utiliser AnimatedContainer
- ☐ Utiliser un TweenAnimationBuilder

## Question 3

J'ai besoin de créer une animation pour laquelle il n'existe pas de « *built-in* », que puis-je faire ?

- ☐ Créer ma propre animation avec TweenAnimationBuilder si elle est explicite, ou avec AnimatedBuilder si elle est implicite
- ☐ Créer ma propre animation avec TweenAnimationBuilder si elle est implicite, ou avec AnimatedBuilder si elle est explicite

## Question 4

Je souhaite arrêter et reprendre une animation selon sa position sur mon écran, qu'est-ce qui me permet de faire cela ?

- ☐ setState()
- ☐ Un controller
- ☐ Tween

## Question 5

Que permet vsync ?

- ☐ De garder le suivi de l'écran
- ☐ De synchroniser des paramètres
- ☐ De créer des animations asynchrones

**Solutions des exercices**






**Exercice p. 7 Solution n°1**


## Question 1

Les animations « *drawing-based* » sont :

- ☐ Simples d'utilisation
- ☐ Consistent en l'animation de widgets communs
- ☒ Peuvent être réalisées grâce à la classe `CustomPainter`
-  Les animations « *drawing-based* » sont des animations complexes, mais pour certaines il est tout de même possible de les coder intégralement grâce à la classe `CustomPainter`.

## Question 2


Je souhaite animer un `Container`, quel type d'animation est-ce ?

- ☒ Une animation « *code-based* »
- ☐ Une animation « *drawing-based* »
-  Un `Container` est un widget commun, qui peut facilement être animé de façon implicite ou explicite.

## Question 3


Quels sont les deux types d'animation « *code-based* » ?

- ☐ Le `CustomPainter`
- ☒ L'implicite
- ☐ Le `TweenAnimationBuilder`
- ☒ L'explicite

 La classe `CustomPainter` permet de créer des animations « *drawing-based* », tandis que la classe `TweenAnimationBuilder` permet de faire des animations implicites. Les deux types d'animations « *code-based* » sont les types implicites et explicites !


## Question 4

Je souhaite créer une animation se répétant en boucle, j'utilise :

- ☐ Une animation implicite
- ☒ Une animation explicite
-  Il est impossible de créer une animation se répétant en boucle de façon implicite, mais de façon explicite oui !

## Question 5

Je souhaite créer une animation discontinue, que je peux arrêter et reprendre, j'utilise :

- ☐ Une animation implicite
- ☒ Une animation explicite
-  Une animation implicite ne permet pas d'arrêter et de reprendre une animation par exemple, car cela nécessite l'utilisation d'un controller qui n'est utilisé que dans les animations explicites.

### Exercice p. 9 Solution n°2

#### Question 1

Quelles sont les deux paramètres ajoutés à un AnimatedFoo par rapport à sa version classique ?

- ☐ Duration et builder
- ☐ Builder et curve
- ☒ Duration et curve



Les deux paramètres ajoutés à la version AnimatedFoo d'un widget commun sont les paramètres duration et curve.

#### Question 2

Si je souhaite créer une animation non-linéaire, que puis-je utiliser ?

- ☐ Le paramètre duration
- ☒ Le paramètre curve
- ☐ La variation des propriétés de mon widget avec setState



Une animation « *non-linéaire* » contient des variations supplémentaires dans le passage entre la valeur de début et la valeur de fin, qui sont dictées par le paramètre curve. Par exemple, la largeur d'un rectangle peut passer de 50 à 100, puis revenir à 80 pour terminer à 100, donnant ainsi un effet « *élastique* » à l'animation.

#### Question 3

La classe TweenAnimationBuilder peut être utilisée si :

- ☒ Mon animation implicite ne possède pas de version « *built-in* »
- ☒ Mon animation implicite possède une version « *built-in* »
- ☐ Mon animation est trop détaillée



Que l'animation possède déjà une version « *built-in* » ou non, il nous est toujours possible d'utiliser la classe TweenAnimationBuilder. Mais si la version « *built-in* » existe, elle vous fera gagner du temps et quelques lignes de code !

#### Question 4

Mon animation se répète à l'infini depuis le début ! Qu'est-ce que cela signifie ?

- ☐ C'est une animation implicite avec TweenAnimationBuilder
- ☒ C'est une animation explicite



Seules les animations explicites peuvent se répéter à l'infini depuis le début.

#### Question 5

Quelle est la classe que je vais utiliser comme variable dans mon TweenAnimationBuilder ?

- ☒ Tween<T extends dynamic>
- ☐ AnimationBuilder
- ☐ AnimatedFoo



La classe Tween vous permet de faire varier aussi bien des couleurs, des tailles, etc. À vous de trouver le Tween qui correspond à la valeur que vous souhaitez faire varier dans votre animation !

**Exercice p. 13 Solution n°3**

## Question 1

Si une animation dure 5 secondes, quelle valeur aura son controller à la seconde 5 ?

- ☒ 1
- ☐ 0
- ☐ 5



La valeur d'un controller varie entre 0 et 1. Si une animation dure 5 secondes, cela signifie que le controller aura la valeur 1 à la seconde 5.

## Question 2

Si je souhaite animer le widget Positioned, que puis-je utiliser ?

- ☒ Un AnimatedPositioned
- ☒ Je crée mon propre AnimatedWidget
- ☒ J'utilise AnimatedBuilder



Tout est possible ! A vous de déterminer ce qui permettra un code plus clair ou plus court, c'est une question de goût.

## Question 3

Qu'est-ce qu'un listenable ?

- ☒ C'est une classe qui prévient les clients lorsqu'une valeur a changé
- ☐ C'est une classe qui prévient le widget parent qu'une valeur a changé C'est un paramètre d'une fonction



La classe listenable est incontournable lorsqu'il s'agit de créer des animations avec AnimatedWidget. Lorsque la valeur rattachée à un listenable est modifiée, les clients sont alors prévenus.

## Question 4

Si je souhaite faire tourner un container sur lui-même en continu, que puis-je utiliser ?

- ☒ Un AnimatedWidget
- ☒ Un AnimatedBuilder
- ☐ Un TweenAnimationBuilder



Seules les animations explicites permettent cela, or TweenAnimationBuilder permet de créer des animations implicites !

## Question 5

Faut-il mieux utiliser un AnimatedWidget ou un AnimatedContainer ?

- ☒ Tout dépend du projet et de la fonction dans lequel il s'intègre
- ☐ Un AnimatedWidget (c'est moins de lignes de code)
- ☐ Un AnimatedBuilder



Les deux classes permettent de faire les mêmes choses. Encore une fois, c'est principalement une question de goût !

**Exercice p. Solution n°4**

```

1 import 'package:flutter/material.dart'; import 'dart:math' as math;
2
3 void main() => runApp(MyApp());
4
5 class MyApp extends StatefulWidget { @override
6 State<MyApp> createState() => _MyAppState();
7 }
8
9 class _MyAppState extends State<MyApp> with SingleTickerProviderStateMixin {
10   AnimationController _controller;
11   bool _bigger = true;
12
13   @override
14   void initState() { super.initState();
15     _controller = AnimationController( duration: const Duration(seconds: 7), vsync: this,
16     ).repeat();
17   }
18
19   @override void dispose() {
20     _controller.dispose(); super.dispose();
21   }
22
23   @override
24   Widget build(BuildContext context) { return MaterialApp(
25     home: Column( children: [ AnimatedBuilder(
26       animation: _controller,
27       child: Container(width: 100.0, height: 100.0, color: Colors.green), builder: (BuildContext
28       context, Widget child) {
29         return Transform.rotate(
30           angle: _controller.value * 2.0 * math.pi, child: child,
31         );
32       },
33       TweenAnimationBuilder<Color>(
34         tween: ColorTween(begin: Colors.red, end: Colors.yellow), duration: Duration(seconds: 7),
35         builder: (BuildContext _, Color value, Widget child) { return ColorFiltered(
36           colorFilter: ColorFilter.mode(value, BlendMode.modulate), child: ClipRect(
37             borderRadius: BorderRadius.circular(100), child: Container(
38               color: Colors.white, height: 100, width: 100)));
39         },
40       ),
41       Column( children: [
42         AnimatedContainer(
43           height: _bigger ? 100 : 200,
44           width: _bigger ? 100 : 200, color: Colors.blue,
45           duration: Duration(seconds: 7), curve: Curves.elasticInOut),
46         Container(height: 16), ElevatedButton(
47           onPressed: () {
48             setState(() => _bigger = !_bigger);
49           },
50           child: Text("Changer la taille"))
51       ],
52     ),
53   ],
54 ),
55 );
56 }

```

57 }

**Exercice p. 15 Solution n°5**

## Question 1

Quel type d'animation sera un container qui se déplace de droite à gauche puis recommence à l'infini ?

- ☐ Une animation « *drawing-based* » implicite
- ☒ Une animation « *drawing-based* » explicite
- ☐ Une animation « *code-based* » implicite
- ☐ Un AnimatedContainer



Il s'agit toujours d'une animation explicite lorsque celle-ci peut se répéter à l'infini depuis le début. Un AnimatedContainer est une animation implicite.

## Question 2

J'ai besoin d'animer un container seulement une fois lorsqu'un bouton est activé, comment faire au plus simple ?

- ☐ Utiliser AnimatedBuilder
- ☒ Utiliser AnimatedContainer
- ☐ Utiliser un TweenAnimationBuilder



L'AnimatedContainer permet d'animer très simplement un container lorsqu'il subit un changement de State.

## Question 3

J'ai besoin de créer une animation pour laquelle il n'existe pas de « *built-in* », que puis-je faire ?

- ☐ Créer ma propre animation avec TweenAnimationBuilder si elle est explicite, ou avec AnimatedBuilder si elle est implicite
- ☒ Créer ma propre animation avec TweenAnimationBuilder si elle est implicite, ou avec AnimatedBuilder si elle est explicite



Les TweenAnimationBuilder (implicite) et AnimatedBuilder (explicite) permettent de créer vos propres animations lorsqu'il n'y a pas d'AnimatedFoo ou de FooTransition correspondant au widget que vous souhaitez animer.

## Question 4

Je souhaite arrêter et reprendre une animation selon sa position sur mon écran, qu'est-ce qui me permet de faire cela ?


- ☐ setState()
- ☒ Un controller
- ☐ Tween



Le controller est ce qui vous permet d'arrêter ou de reprendre des animations.

## Question 5

Que permet vsync ?

- ☒ De garder le suivi de l'écran
- ☐ De synchroniser des paramètres
- ☐ De créer des animations asynchrones
-  Vsync s'occupe de garder le suivi de l'écran, afin de ne pas continuer à générer l'animation lorsque celle-ci n'apparaît pas.