
Page 2

Flutter pour les débutants

Un guide d'introduction à la création de mobiles multiplateformes applications avec Flutter et Dart 2

Alessandro Biessek

BIRMINGHAM - MUMBAI

Page 3

Flutter pour les débutants

Droits d'auteur © 2019 Packt Publishing

Tous les droits sont réservés. Aucune partie de ce livre ne peut être reproduite, stockée dans un système de récupération ou transmise sous quelque forme que ce soit

https://translate.googleusercontent.com/translate_f

1/364

ou par quelque moyen que ce soit, sans l'autorisation écrite préalable de l'éditeur, sauf en cas de brèves citations intégré dans des articles ou des critiques critiques.

Tous les efforts ont été faits dans la préparation de ce livre pour assurer l'exactitude des informations présentées. Cependant, les informations contenues dans ce livre sont vendues sans garantie, expresse ou implicite. Ni le l'auteur, ni Packt Publishing ou ses revendeurs et distributeurs, ne seront tenus responsables de tout dommage causé ou allégué qui ont été causés directement ou indirectement par ce livre.

Packt Publishing s'est efforcé de fournir des informations sur les marques de commerce sur toutes les sociétés et produits mentionné dans ce livre par l'utilisation appropriée des majuscules. Cependant, Packt Publishing ne peut garantir l'exactitude de ces informations.

Rédacteur de la mise en service: Amarabha Banerjee

Rédactrice en chef: Larissa Pinto

Éditeur du développement de contenu: Akhil Nair

Rédacteur technique: Sachin Sunilkumar

Éditeur de copie: Safis Editing

Coordinateur de projet: Manthan Patel

Relecteur : Safis Editing

Indexeur : Pratik Shirodkar

Chef décorateur: Jyoti Chauhan

Première publication: septembre 2019

Référence de production: 1120919

Édité par Packt Publishing Ltd.

Lieu de livraison

35 rue Livery

Birmingham

B3 2PB, Royaume-Uni.

ISBN 978-1-78899-608-2

www.packt.com

À ma mère, Antonina, et à mon père, Euclide, pour leurs sacrifices et pour avoir illustré le pouvoir de détermination

- Alessandro Biessek

Page 5

[mapt.io](#)

Abonnez-vous à notre bibliothèque numérique en ligne pour un accès complet à plus de 7 000 livres et vidéos. En tant qu'outils de pointe pour vous aider à planifier votre développement personnel et à faire progresser votre carrière. Pour plus d'information, veuillez visiter notre site web.

Pourquoi vous abonner?

Passez moins de temps à apprendre et plus de temps à coder avec des livres électroniques et des vidéos pratiques de plus de 4000 professionnels de l'industrie

Améliorez votre apprentissage avec des plans de compétences conçus spécialement pour vous

Obtenez un eBook ou une vidéo gratuit chaque mois

Entièrement consultable pour un accès facile aux informations vitales

Copier et coller, imprimer et ajouter du contenu

Packt.com

Saviez-vous que Packt propose des versions électroniques de chaque livre publié, avec PDF et Fichiers ePub disponibles? Vous pouvez passer à la version eBook sur www.packt.com et sous forme d'impression client livre, vous avez droit à une réduction sur la copie de l'eBook. Contactez-nous au customercare@packtpub.com pour plus de détails.

À www.packt.com, vous pouvez également lire une collection d'articles techniques gratuits, vous inscrire à un des newsletters gratuites et recevez des remises et offres exclusives sur les livres Packt et eBooks.

Contributeurs

A propos de l'auteur

Alessandro Biessek est né dans la belle ville de Chapecó, dans l'état de Santa Catarina, sud du Brésil, en 1993. Il travaille actuellement sur une application mobile développement pour Android et iOS dans sa ville natale. Il a plus de 7 ans de expérience en développement, du développement de bureau avec Delphi au backend avec PHP, Node.js, Golang, développement mobile avec Apache Flex et Java / Kotlin. La plupart de son temps est consacré au développement d'applications Android. Toujours intéressé par les nouvelles technologies, il suit le framework Flutter depuis longtemps, témoignant de sa croissance et adoption ces derniers mois.

Tout d'abord, merci à l'équipe Flutter pour leur outil incroyable qui aide le développeur communauté pour aider les autres.

Je suis reconnaissant à tous ceux avec qui j'ai eu le plaisir de travailler pendant cette projet, tous les relecteurs et toute l'équipe Packt qui m'a aidé dans ce travail.

Je voudrais remercier mes amis, mes collègues et ma famille, en particulier ma mère Antonina, ma père Euclide, ma sœur Hellen et mon frère Alan, pour leur soutien et ont été tenir le fort pendant que je travaillais dur sur le livre. Aussi, grâce à mes professeurs diplômés, qui m'a encouragé à affronter des défis comme ce livre d'une manière plus naturelle et courageuse. Enfin, je tiens à vous remercier, le lecteur. Votre soutien à des livres comme celui-ci, à travers votre achat, permet à tous ceux qui souhaitent partager leurs expériences de continuer.

À propos du critique

Ugurcan Yildirim est un passionné du développement d'applications mobiles Android et Flutter cadres. Il est diplômé en tant que major de promotion avec un BSc en informatique de Université Bilkent, Ankara. Depuis 2015, il travaille en tant qu'ingénieur Android chez Accenture Industry X.0, Istanbul. Avec la tendance haussière prometteuse de Flutter qui a commencé en 2018, il a commencé à se préoccuper des particularités de Flutter et à les expérimenter. Depuis puis, il a contribué à la communauté open source de Flutter en écrivant des articles sur Medium (@ugurcanY) et faire des présentations. Sa dernière contribution est de revoir ce

livre, qui, selon lui, devrait être consulté et référencé par les développeurs Flutter de tout niveau.

Je tiens à remercier Packt de m'avoir donné l'opportunité de contribuer à la élargir l'univers Flutter en passant en revue l'un des premiers et des plus complets Flutter livres publiés. Je voudrais également remercier mes parents et ma femme, Karsu, pour leur soutien et patience tout au long de la lecture de ce livre.

Packt recherche des auteurs comme vous

Si vous souhaitez devenir auteur pour Packt, veuillez visiter auteurs.packtpub.com et postulez aujourd'hui. Nous avons travaillé avec des milliers de développeurs et de professionnels de la technologie, tout comme vous, pour les aider à partager leurs connaissances avec la communauté technologique mondiale. Vous pouvez faire une candidature générale, postuler pour un sujet d'actualité spécifique pour lequel nous recrutons un auteur pour, ou soumettez votre propre idée.

Page 8

Table des matières

| | |
|---|-----|
| Préface | 1 |
| Section 1: Introduction à Dart | |
| Chapitre 1: Une introduction à Dart | |
| Premiers pas avec Dart | |
| L'évolution de Dart | 8 |
| Comment fonctionne Dart | 9 |
| Compilation de Dart VM et JavaScript | 9 |
| Fléchette pratique | |
| DartPad | dix |
| Outils de développement de fléchettes | dix |
| Bonjour le monde | 12 |
| Comprendre pourquoi Flutter utilise Dart | |
| Ajouter de la productivité | 13 |
| Compilation d'applications Flutter et recharge à chaud | 14 |
| Apprentissage facile | 14 |
| Maturité | 15 |
| Présentation de la structure du langage Dart | |
| Opérateurs de fléchettes | 17 |
| Opérateurs arithmétiques | 19 |
| Opérateurs d'incrémentation et de décrémentation | 19 |
| Égalité et opérateurs relationnels | 20 |
| Vérification de type et moulage | 21 |
| Opérateurs logiques | 21 |
| Manipulation des bits | 22 |
| Opérateurs à sécurité nulle et prenant en charge les valeurs nulles | 22 |
| Types et variables de fléchettes | 23 |

| | |
|--|-----------|
| <u>final et const</u> | <u>23</u> |
| <u>Types intégrés</u> | <u>24</u> |
| <u>Nombres</u> | <u>24</u> |
| <u>BigInt</u> | <u>25</u> |
| <u>Booléens</u> | <u>25</u> |
| <u>Les collections</u> | <u>25</u> |
| <u>Cordes</u> | <u>26</u> |
| <u>Interpolation de chaîne</u> | <u>26</u> |
| <u>Littéraux</u> | <u>27</u> |
| <u>Inférence de type - apportant du dynamisme au spectacle</u> | <u>27</u> |
| <u>Contrôle des flux et des boucles</u> | <u>29</u> |
| <u>Les fonctions</u> | <u>29</u> |
| <u>Paramètres de fonction</u> | <u>30</u> |
| <u>Structures de données, collections et génériques</u> | <u>33</u> |
| <u>Génériques</u> | <u>34</u> |

Page 9*Table des matières*

| | |
|---|-----------|
| <u>Quand et pourquoi utiliser des génériques</u> | <u>34</u> |
| <u>Génériques et littéraux Dart</u> | <u>35</u> |
| Introduction à la POO dans Dart | <u>35</u> |
| <u>Fonctionnalités de Dart OOP</u> | <u>36</u> |
| <u>Objets et classes</u> | <u>37</u> |
| <u>Encapsulation</u> | <u>38</u> |
| <u>Héritage et composition</u> | <u>38</u> |
| <u>Abstraction</u> | <u>38</u> |
| <u>Polymorphisme</u> | <u>39</u> |
| Sommaire | <u>39</u> |
| Lectures complémentaires | <u>40</u> |
| Chapitre 2: Programmation intermédiaire des fléchettes | <u>41</u> |
| <u>Classes et constructeurs de fléchettes</u> | <u>42</u> |
| <u>Le type enum</u> | <u>42</u> |
| <u>La notation en cascade</u> | <u>44</u> |
| <u>Constructeurs</u> | <u>44</u> |
| <u>Constructeurs nommés</u> | <u>45</u> |
| <u>Constructeurs d'usine</u> | <u>46</u> |
| <u>Accesseurs de terrain - getters et setters</u> | <u>47</u> |
| <u>Champs et méthodes statiques</u> | <u>48</u> |
| <u>Héritage de classe</u> | <u>50</u> |
| <u>La méthode toString()</u> | <u>51</u> |
| <u>Interfaces, classes abstraites et mixins</u> | <u>51</u> |
| <u>Classes abstraites</u> | <u>52</u> |
| <u>Interfaces</u> | <u>53</u> |
| <u>Mixins - ajouter un comportement à une classe</u> | <u>54</u> |
| <u>Classes appelables, fonctions de niveau supérieur et variables</u> | <u>56</u> |
| <u>Classes appelables</u> | <u>57</u> |
| <u>Fonctions et variables de niveau supérieur</u> | <u>58</u> |
| <u>Comprendre les bibliothèques et les packages Dart</u> | <u>58</u> |
| <u>Importer et utiliser une bibliothèque</u> | <u>59</u> |
| <u>Importer afficher et masquer</u> | <u>60</u> |
| <u>Importer des préfixes dans des bibliothèques</u> | <u>61</u> |
| <u>Importer des variantes de chemin</u> | <u>62</u> |
| <u>Création de bibliothèques Dart</u> | <u>63</u> |
| <u>Confidentialité des membres de la bibliothèque</u> | <u>64</u> |
| <u>La définition de la bibliothèque</u> | <u>65</u> |
| <u>Une bibliothèque à un seul fichier</u> | <u>65</u> |
| <u>Diviser les bibliothèques en plusieurs fichiers</u> | <u>66</u> |
| <u>Une bibliothèque multi-fichiers - l'instruction d'exportation</u> | <u>69</u> |
| <u>Paquets de fléchettes</u> | <u>72</u> |
| <u>Packages d'application et packages de bibliothèque</u> | <u>72</u> |
| <u>Structures de package</u> | <u>73</u> |
| <u>Stagehand - Le générateur de projet Dart</u> | <u>75</u> |
| <u>Le fichier pubspec</u> | <u>77</u> |
| <u>Dépendances de paquet - pub</u> | <u>79</u> |
| <u>Spécifier les dépendances</u> | <u>80</u> |

Table des matières

| | |
|--|-----|
| <u>La contrainte de version</u> | 81 |
| <u>La contrainte source</u> | 82 |
| Présentation de la programmation asynchrone avec Futures et Isolates | 84 |
| <u>Futures de fléchettes</u> | 84 |
| <u>Isolats de fléchettes</u> | 88 |
| Présentation des tests unitaires avec Dart | 89 |
| <u>Le package de test Dart</u> | 90 |
| <u>Rédaction de tests unitaires</u> | 90 |
| Sommaire | 93 |
| Chapitre 3: Une introduction à Flutter | 94 |
| Comparaisons avec d'autres frameworks de développement d'applications mobiles | 95 |
| <u>Les problèmes que Flutter veut résoudre</u> | 95 |
| <u>Differences entre les cadres existants</u> | 96 |
| <u>Haute performance</u> | 97 |
| <u>Contrôle total de l'interface utilisateur</u> | 97 |
| <u>Dard</u> | 99 |
| <u>Être soutenu par Google</u> | 100 |
| <u>OS Fuchsia et Flutter</u> | 100 |
| <u>Framework open source</u> | 101 |
| <u>Ressources et outils pour les développeurs</u> | 101 |
| <u>Compilation Flutter (Dart)</u> | 103 |
| <u>Compilation de développement</u> | 104 |
| <u>Compilation de publication</u> | 104 |
| <u>Plateformes prises en charge</u> | 104 |
| Rendu flottant | 105 |
| <u>Technologies basées sur le Web</u> | 106 |
| <u>Framework et widgets OEM</u> | 107 |
| <u>Flutter - rendu par lui-même</u> | 108 |
| Présentation des widgets | 108 |
| <u>Composabilité</u> | 109 |
| <u>Immutabilité</u> | 109 |
| <u>Tout est un widget</u> | 109 |
| <u>L'arborescence des widgets</u> | 110 |
| Bonjour Flutter | 111 |
| <u>fichier pubspec</u> | 114 |
| <u>Exécution du projet généré</u> | 117 |
| <u>fichier lib / main.dart</u> | 117 |
| <u>Flutter run</u> | 117 |
| Sommaire | 120 |
| Section 2: L'interface utilisateur Flutter - Tout est un widget | |
| Chapitre 4: Widgets: Création de dispositions dans Flutter | 122 |
| Widgets avec état ou sans état | 122 |
| <u>Widgets sans état</u> | 123 |

[iii]

Table des matières

| | |
|--|-----|
| <u>Widgets avec état</u> | 124 |
| <u>Widgets avec état et sans état dans le code</u> | 124 |
| <u>Widget sans état dans le code</u> | 126 |
| <u>Widgets avec état dans le code</u> | 127 |
| <u>Widgets hérités</u> | 130 |
| <u>Propriété de la clé du widget</u> | 132 |
| Widgets intégrés | 132 |
| <u>Widgets de base</u> | 132 |
| <u>Le widget Texte</u> | 133 |
| <u>Le widget Image</u> | 133 |
| <u>Widgets Material Design et iOS Cupertino</u> | 135 |
| <u>Boutons</u> | 136 |
| <u>Échafaud</u> | 136 |
| <u>Dialogues</u> | 137 |
| <u>Champs de texte</u> | 137 |
| <u>Widgets de sélection</u> | 138 |
| <u>Sélecteurs de date et d'heure</u> | 138 |
| <u>Autres composants</u> | 138 |

| | |
|---|------------|
| Comprendre les widgets de mise en page intégrés | <u>139</u> |
| Conteneurs | <u>139</u> |
| Stylisme et positionnement | <u>140</u> |
| Autres widgets (gestes, animations et transformations) | <u>140</u> |
| Création d'une interface utilisateur avec des widgets (application de gestion de faveur) | <u>141</u> |
| Les écrans de l'application | <u>141</u> |
| Le code de l'application | <u>141</u> |
| Favorise l'écran d'accueil de l'application | <u>143</u> |
| Le code de mise en page | <u>145</u> |
| L'écran de demande de faveur | <u>153</u> |
| Le code de mise en page | <u>154</u> |
| Créer des widgets personnalisés | <u>157</u> |
| Sommaire | <u>159</u> |
| Chapitre 5: Gestion des entrées et des gestes de l'utilisateur | <u>160</u> |
| Gérer les gestes de l'utilisateur | <u>160</u> |
| Pointeurs | <u>161</u> |
| Gestes | <u>162</u> |
| Robinet | <u>162</u> |
| Tapez deux fois | <u>163</u> |
| Appuyez et maintenez | <u>164</u> |
| Faire glisser, déplacer et mettre à l'échelle | <u>164</u> |
| Trainée horizontale | <u>164</u> |
| Glissement vertical | <u>165</u> |
| La poêle | <u>166</u> |
| Échelle | <u>166</u> |
| Gestes dans les widgets matériels | <u>167</u> |
| Widgets d'entrée | <u>168</u> |
| FormField et TextField | <u>168</u> |
| Utilisation d'un contrôleur | <u>169</u> |
| Accéder à l'état de FormField | <u>169</u> |

[iv]

Page 12*Table des matières*

| | |
|--|------------|
| Forme | <u>170</u> |
| Accéder à l'état du formulaire | <u>171</u> |
| Utiliser une clé | <u>171</u> |
| Utilisation d'InheritedWidget | <u>171</u> |
| Validation des entrées (formulaires) | <u>173</u> |
| Validation de l'entrée utilisateur | <u>173</u> |
| Entrée personnalisée et TextFormField | <u>174</u> |
| Créer des entrées personnalisées | <u>174</u> |
| Exemple de widget d'entrée personnalisé | <u>175</u> |
| Créer un widget d'entrée | <u>176</u> |
| Transformez le widget en widget TextFormField | <u>177</u> |
| Mettre tous ensemble | <u>179</u> |
| Écran des faveurs | <u>179</u> |
| Appuyez sur les gestes sur l'onglet Favoris | <u>181</u> |
| Appuyez sur les gestes sur FavorCards | <u>182</u> |
| Faire des FavorsPage un StatefulWidget | <u>182</u> |
| Refuser la gestion des actions | <u>184</u> |
| Faire la gestion des actions | <u>185</u> |
| Appuyez sur le bouton Demander une faveur | <u>186</u> |
| L'écran Demander une faveur | <u>187</u> |
| Le bouton de fermeture | <u>188</u> |
| Le bouton SAVE | <u>188</u> |
| Validation de l'entrée à l'aide du widget Formulaire | <u>188</u> |
| Sommaire | <u>190</u> |
| Chapitre 6: Thème et style | <u>191</u> |
| Widgets de thème | <u>191</u> |
| Widget de thème | <u>192</u> |
| ThèmeDonnées | <u>193</u> |
| Luminosité | <u>194</u> |
| Theming en pratique | <u>195</u> |
| Classe de plateforme | <u>197</u> |
| Conception matérielle | <u>198</u> |
| Widget MaterialApp | <u>199</u> |
| Widget d'échafaudage | <u>201</u> |
| Thème personnalisé | <u>203</u> |
| iOS Cupertino | <u>205</u> |
| CupertinoApp | <u>206</u> |
| Cupertino en pratique | <u>206</u> |

| | |
|---|------------|
| <u>Utilisation de polices personnalisées</u> | <u>208</u> |
| Remplacer la police par défaut dans l'application | <u>210</u> |
| Style dynamique avec MediaQuery et LayoutBuilder | <u>211</u> |
| LayoutBuilder | <u>211</u> |
| MediaQuery | <u>214</u> |
| Exemple MediaQuery | <u>214</u> |
| Classes réactives supplémentaires | <u>218</u> |

[v]

Page 13*Table des matières*

| | |
|---|------------|
| Sommaire | <u>218</u> |
| Chapitre 7: Routage: navigation entre les écrans | <u>219</u> |
| Comprendre le widget Navigator | <u>219</u> |
| Navigateur | <u>220</u> |
| Recouvrir | <u>220</u> |
| Pile de navigation / historique | <u>221</u> |
| Route | <u>221</u> |
| RouteSettings | <u>221</u> |
| MaterialPageRoute et CupertinoPageRoute | <u>222</u> |
| Mettre tous ensemble | <u>222</u> |
| La manière WidgetsApp | <u>226</u> |
| Itinéraires nommés | <u>228</u> |
| Déplacement vers des itinéraires nommés | <u>229</u> |
| Arguments | <u>230</u> |
| Récupération des résultats de Route | <u>231</u> |
| Transitions d'écran | <u>232</u> |
| PageRouteBuilder | <u>233</u> |
| Transitions personnalisées en pratique | <u>233</u> |
| Animations de héros | <u>235</u> |
| Le widget Hero | <u>235</u> |
| Implémentation des transitions Hero | <u>236</u> |
| Sommaire | <u>244</u> |

Section 3: Développement d'applications complètes

| | |
|--|------------|
| Chapitre 8: Plugins Firebase | <u>246</u> |
| Présentation de Firebase | <u>246</u> |
| Configurer Firebase | <u>247</u> |
| Connexion de l'application Flutter à Firebase | <u>252</u> |
| Configurer une application Android | <u>252</u> |
| Configurer l'application iOS | <u>255</u> |
| FlutterFire | <u>255</u> |
| Ajout de la dépendance FlutterFire au projet Flutter | <u>255</u> |
| Authentification Firebase | <u>256</u> |
| Activation des services d'authentification dans Firebase | <u>257</u> |
| Écran d'authentification | <u>259</u> |
| Connexion avec Firebase | <u>260</u> |
| Envoi du code de vérification | <u>261</u> |
| Vérification du code SMS | <u>263</u> |
| Mise à jour du profil et du statut de connexion | <u>264</u> |
| Base de données NoSQL avec Cloud Firestore | <u>265</u> |
| Activation de Cloud Firestore sur Firebase | <u>266</u> |
| Cloud Firestore et Flutter | <u>268</u> |
| Chargement des faveurs de Firestore | <u>269</u> |
| Mettre à jour les faveurs sur Firebase | <u>272</u> |
| Enregistrer une faveur sur Firebase | <u>272</u> |

[vi]

Page 14*Table des matières*275

| | |
|--|-----|
| Cloud Storage avec Firebase Storage | 275 |
| Ajout de dépendances de stockage Flutter | 276 |
| Téléchargement de fichiers sur Firebase | 276 |
| annonces avec Firebase AdMob | 280 |
| Compte AdMob | 281 |
| Créer un compte AdMob | 282 |
| AdMob dans Flutter | 285 |
| Note d'accompagnement sur Android | 287 |
| Note d'accompagnement sur iOS | 287 |
| Affichage des annonces dans Flutter | 287 |
| ML avec Firebase ML Kit | 290 |
| Ajout du kit ML à Flutter | 290 |
| Utilisation du détecteur d'étiquettes dans Flutter | 291 |
| Sommaire | 294 |
| Chapitre 9: Développer votre propre plugin Flutter | 295 |
| Créer un projet de package / plugin | 295 |
| Paquets Flutter par rapport aux paquets Dart | 296 |
| Démarrer un projet de package Dart | 296 |
| Démarrer un package de plugins Flutter | 298 |
| Une structure de projet de plugin | 298 |
| MethodChannel | 300 |
| Implémentation du plugin Android | 301 |
| Implémentation du plugin iOS | 302 |
| L'API Dart | 303 |
| Un exemple de package de plugin | 304 |
| Utilisation du plugin | 304 |
| Ajout de documentation au package | 305 |
| Fichiers de documentation | 306 |
| Documentation de la bibliothèque | 306 |
| Générer de la documentation | 307 |
| Publier un package | 308 |
| Recommandations de développement de projets de plugins | 308 |
| Sommaire | 309 |
| Chapitre 10: Accéder aux fonctionnalités de l'appareil depuis l'application Flutter | 310 |
| Lancer une URL depuis l'application | 310 |
| Afficher un lien | 311 |
| Le plugin flutter_linkify | 312 |
| Lancer une URL | 314 |
| Le plugin url_launcher | 314 |
| Gérer les autorisations des applications | 315 |
| Gérer les autorisations sur Flutter | 316 |
| Utilisation du plugin permission_handler | 317 |
| Importer un contact depuis le téléphone | 317 |

[vii]

Page 15

Table des matières

| | |
|--|-----|
| Importer un contact avec contact_picker | 318 |
| Autorisation de contact avec permission_handler | 320 |
| Autorisation de contact sur Android | 320 |
| Autorisation de contact sur iOS | 321 |
| Vérification et demande d'autorisation dans Flutter (permission_handler) | 321 |
| Intégration de la caméra du téléphone | 322 |
| Prendre des photos avec image_picker | 323 |
| Autorisation de la caméra avec permission_handler | 323 |
| Autorisation de la caméra sur Android | 324 |
| Autorisation de la caméra sur iOS | 324 |
| Demande d'autorisation de caméra dans Flutter (permission_handler) | 325 |
| Sommaire | 326 |
| Chapitre 11: Vues de la plate-forme et intégration de la carte | 327 |
| Afficher une carte | 328 |
| Vues de la plateforme | 329 |
| Activation des vues de plate-forme sur iOS | 330 |
| Créer un widget de vue de plate-forme | 330 |
| Créer une vue Android | 331 |
| Créer une vue iOS | 333 |
| Utilisation d'un widget de vue de plate-forme | 334 |
| Premiers pas avec le plugin google_maps_flutter | 336 |
| Afficher une carte avec le plugin google_maps_flutter | 338 |

| | |
|--|-----|
| Activation de l'API Maps sur Google Cloud Console | 338 |
| Intégration de l'API Google Maps sur Android | 341 |
| Intégration de l'API Google Maps sur iOS | 341 |
| Afficher une carte sur Flutter | 342 |
| Ajout de marqueurs à la carte | 343 |
| La classe Marker | 344 |
| Ajout de marqueurs dans le widget GoogleMap | 345 |
| Ajout d'interactions cartographiques | 347 |
| Ajouter des marqueurs dynamiquement | 347 |
| GoogleMapController | 348 |
| Obtenir GoogleMapController | 348 |
| Animation d'une caméra cartographique vers un emplacement | 349 |
| Utilisation de l'API Google Places | 349 |
| Activer l'API Google Places | 350 |
| Premiers pas avec le plugin google_maps_webservice | 351 |
| Obtenir une adresse de lieu à l'aide du plug-in google_maps_webservice | 351 |
| Sommaire | 354 |

Section 4: Flutter avancé - Ressources au complexe applications

| | |
|---|-----|
| Chapitre 12: Test, débogage et déploiement | 356 |
| Test de scintillement - tests unitaires et widgets | 356 |
| Tests de widgets | 357 |

[viii]

Page 16

Table des matières

| | |
|--|-----|
| Le package flutter_test | 357 |
| La fonction testWidgets | 357 |
| Exemple de test de widget | 358 |
| Débogage des applications Flutter | 360 |
| Observatoire | 360 |
| Fonctionnalités de débogage supplémentaires | 363 |
| DevTools | 363 |
| Profilage des applications Flutter | 365 |
| Le profiteur de l'Observatoire | 365 |
| Mode profil | 366 |
| Superposition de performances | 367 |
| Inspection de l'arborescence des widgets Flutter | 368 |
| Inspecteur de widgets | 369 |
| L'inspecteur Flutter dans DevTools | 369 |
| Préparation des applications pour le déploiement | 370 |
| Mode de libération | 371 |
| Publier des applications pour Android | 371 |
| AndroidManifest et build.gradle | 371 |
| AndroidManifest - autorisations | 372 |
| AndroidManifest - balises métadonnées | 372 |
| AndroidManifest - nom et icône de l'application | 373 |
| build.gradle - ID et versions de l'application | 374 |
| build.gradle - signature de l'application | 375 |
| Publier des applications pour iOS | 377 |
| Connecter l'App Store | 377 |
| Xcode | 378 |
| Xcode - détails de l'application et ID du bundle | 378 |
| Xcode - AdMob | 378 |
| Xcode - signature de l'application | 378 |
| Sommaire | 379 |
| Chapitre 13: Amélioration de l'expérience utilisateur | 380 |
| Accessibilité dans Flutter et ajout de traductions aux applications | 380 |
| Prise en charge de Flutter pour l'accessibilité | 381 |
| Internationalisation de Flutter | 381 |
| Le paquet intl | 382 |
| Le package intl_translation | 382 |
| Le package flutter_localizations | 382 |
| Ajouter des localisations à une application Flutter | 382 |
| Dépendances | 383 |
| La classe AppLocalization | 383 |
| Génération de fichiers .arb avec intl_translation | 385 |
| Utiliser des ressources traduites | 387 |
| Communication entre natif et Flutter avec les canaux de la plateforme | 389 |
| Chaîne de plate-forme | 390 |
| Codecs de message | 391 |

Création de processus d'arrière-plan

La fonction flutter compute()

392
392

[ix]

Page 17

Table des matières

| | |
|--|------------|
| <u>SendPort et ReceivePort</u> | <u>393</u> |
| <u>IsolateNameServer</u> | <u>394</u> |
| <u>Un exemple de compute()</u> | <u>394</u> |
| <u>Processus d'arrière-plan complet</u> | <u>395</u> |
| <u>Initier le calcul</u> | <u>397</u> |
| <u>L'arrière-plan isoler</u> | <u>398</u> |
| Ajout de code spécifique à Android pour exécuter du code Dart en arrière-plan | 400 |
| <u>La classe HandsOnBackgroundProcessPlugin</u> | <u>400</u> |
| <u>La classe BackgroundProcessService</u> | <u>402</u> |
| <u>La propriété PluginRegistrantCallback</u> | <u>405</u> |
| Ajout de code spécifique à iOS pour exécuter du code Dart en arrière-plan | 406 |
| <u>La classe SwiftHandsOnBackgroundProcessPlugin</u> | <u>407</u> |
| Sommaire | 412 |
| Chapitre 14: Manipulations graphiques des widgets | 413 |
| Transformer des widgets avec la classe Transform | 413 |
| <u>Le widget Transform</u> | <u>414</u> |
| <u>Comprendre la classe Matrix4</u> | <u>415</u> |
| Explorer les types de transformations | 415 |
| <u>Faire pivoter la transformation</u> | <u>416</u> |
| <u>Transformation d'échelle</u> | <u>417</u> |
| <u>Traduire la transformation</u> | <u>418</u> |
| <u>Transformations composées</u> | <u>420</u> |
| Appliquer des transformations à vos widgets | 421 |
| <u>Rotation des widgets</u> | <u>421</u> |
| <u>Mise à l'échelle des widgets</u> | <u>422</u> |
| <u>Traduire des widgets</u> | <u>422</u> |
| <u>Application de plusieurs transformations</u> | <u>423</u> |
| Utilisation de peintres et de toiles personnalisés | 424 |
| <u>La classe Canvas</u> | <u>424</u> |
| <u>Transformations du canevas</u> | <u>425</u> |
| <u>Clip de toileRect</u> | <u>425</u> |
| <u>Méthodes</u> | <u>425</u> |
| <u>L'objet Paint</u> | <u>426</u> |
| <u>Le widget CustomPaint</u> | <u>426</u> |
| <u>Détails de construction CustomPaint</u> | <u>426</u> |
| <u>L'objet CustomPainter</u> | <u>428</u> |
| <u>La méthode de peinture</u> | <u>428</u> |
| <u>La méthode shouldRepaint</u> | <u>428</u> |
| <u>Un exemple pratique</u> | <u>429</u> |
| <u>Définition d'un widget</u> | <u>430</u> |
| <u>Définition de CustomPainter</u> | <u>431</u> |
| <u>Remplacer la méthode shouldRepaint</u> | <u>431</u> |
| <u>Remplacer la méthode de peinture</u> | <u>432</u> |
| <u>La variante du diagramme radial</u> | <u>435</u> |
| <u>Définition d'un widget</u> | <u>436</u> |
| <u>Définition de CustomPainter</u> | <u>436</u> |

[X]

Page 18

Table des matières

| | |
|-------------------------------------|------------|
| Sommaire | 440 |
| Chapitre 15: Animations | 441 |
| Présentation des animations | 441 |
| <u>La classe Animation<T></u> | <u>441</u> |
| <u>AnimationContrôleur</u> | <u>443</u> |
| <u>TickerProvider et Ticker</u> | <u>444</u> |

| | |
|---|---------------------|
| CourbéAnimation | 444 |
| Tween | 445 |
| Utiliser des animations | 445 |
| Faire pivoter l'animation | 446 |
| Animation d'échelle | 450 |
| Traduire l'animation | 452 |
| Transformations multiples et Tween personnalisé | 453 |
| Tween personnalisé | 455 |
| Utilisation d'AnimatedBuilder | 458 |
| La classe AnimatedBuilder | 458 |
| Revisiter notre animation | 459 |
| Utilisation d'AnimatedWidget | 462 |
| La classe AnimatedWidget | 463 |
| Réécrire l'animation avec AnimatedWidget | 463 |
| Sommaire | 464 |
| Autres livres que vous pourriez apprécier | 465 |
| Indice | 468 |

[xi]

Page 19

Préface

Flutter for Beginners vous aide à entrer dans le monde du framework Flutter et à créer de superbes Applications mobiles. Je vais vous emmener d'une introduction au langage Dart à une exploration de tous les blocs Flutter nécessaires pour créer une application de haut niveau. Ensemble, nous allons créer une application complète. Avec des exemples de code clairs, vous apprendrez comment démarrer un petit projet Flutter, ajoutez des widgets, appliquez des styles et des thèmes, connectez-vous à la télécommande des services tels que Firebase, obtenir des entrées utilisateur, ajouter des animations pour améliorer l'expérience utilisateur, et plus. De plus, vous apprendrez à ajouter des fonctionnalités avancées, des intégrations de cartes, travailler avec du code spécifique à la plate-forme avec des langages de programmation natifs et créer de fantastiques UI avec animations personnalisées. Bref, ce livre vous préparera à l'avenir de développement mobile avec ce cadre étonnant.

À qui s'adresse ce livre

Ce livre est destiné aux développeurs qui souhaitent apprendre le cadre révolutionnaire de Google, Flutter de zéro. Aucune connaissance de Flutter ou Dart n'est requise. Cependant, la programmation de base la connaissance de la langue sera utile.

Ce que couvre ce livre

[Chapitre 1](#), *An Introduction to Dart*, présente les bases du langage Dart.

[Chapitre 2](#), *Intermediate Dart Programming*, examine les fonctionnalités de programmation orientée objet et des concepts avancés de Dart, bibliothèques, packages et programmation asynchrone.

[chapitre 3](#), *An Introduction to Flutter*, vous présente le monde de Flutter.

[Chapitre 4](#), *Widgets: Construire des mises en page dans Flutter*, explique comment créer des mises en page dans Flutter.

[Chapitre 5](#), *Gestion des entrées et gestes utilisateur*, vous montre comment gérer les entrées utilisateur avec Widgets Flutter.

Page 20

Préface

[Chapitre 6](#), *Theming and Styling*, vous apprend à appliquer différents styles à Flutter widgets.

[Chapitre 7](#), *Routage: Naviguer entre les écrans*, explore comment ajouter la navigation à l'application écrans.

[Chapitre 8](#), *Firebase Plugins*, explique comment utiliser les plugins Firebase dans les applications Flutter.

[Chapitre 9](#), *Developing Your Own Flutter Plugin*, explique comment créer votre propre Flutter plugins.

[Le chapitre 10](#), *Accès aux fonctionnalités de l'appareil depuis l'application Flutter*, explique comment interagir avec les fonctionnalités de l'appareil telles que les caméras et les listes de contacts.

[Le chapitre 11](#), *Vues de la plate-forme et intégration de la carte*, vous montre comment ajouter des vues de carte à Flutter applications.

[Le chapitre 12](#), *Test, débogage et déploiement*, explore les outils Flutter pour améliorer productivité.

[Le chapitre 13](#), *Amélioration de l'expérience utilisateur*, explore comment améliorer l'expérience utilisateur en utilisant des fonctionnalités telles que l'exécution et l'internationalisation de Dart en arrière-plan.

[Le chapitre 14](#), *Manipulations graphiques des widgets*, explique comment créer des visuels uniques avec manipulations graphiques.

[Le chapitre 15](#), *Animations*, vous donne un aperçu de la façon d'ajouter des animations à Flutter widgets.

Pour tirer le meilleur parti de ce livre

Vous serez présenté aux exigences au fur et à mesure que nous avancerons dans les chapitres. Obtenir démarré, vous devez avoir accès à un navigateur pour pouvoir accéder au site Web DartPad et jouer avec le code Dart.

Pour développer et publier des applications iOS de manière professionnelle, vous avez besoin d'une licence de développeur (payante annuellement), un Mac et au moins un appareil pour tester les applications. Tout cela n'est pas strictement nécessaire pour apprendre Flutter, mais cela pourrait vous être utile.

L'ensemble du processus d'installation et les exigences de l'environnement Flutter sont disponible sur le site officiel (<https://flutter.dev/docs/>) ne vous inquiétez pas: vous pouvez commencer avec le nécessaire.

Page 21

Préface

Téléchargez les fichiers de code d'exemple

Vous pouvez télécharger les fichiers d'exemple de code pour ce livre à partir de votre compte à l'adresse

www.packt.com. Si vous avez acheté ce livre ailleurs, vous pouvez visiter
www.packtpub.com/support et inscrivez-vous pour recevoir les fichiers directement par e-mail.

Vous pouvez télécharger les fichiers de code en suivant ces étapes:

1. Connectez-vous ou enregistrez-vous sur www.packt.com .
2. Sélectionnez l'onglet **Support** .
3. Cliquez sur **Téléchargements de code** .
4. Entrez le nom du livre dans la zone de **recherche** et suivez les instructions.

Une fois le fichier téléchargé, assurez-vous de décompresser ou d'extraire le dossier à l'aide du dernière version de:

WinRAR / 7-Zip pour Windows
Zipeg / iZip / UnRarX pour Mac
7-Zip / PeaZip pour Linux

Le faisceau de code pour le livre est également hébergé sur GitHub à <https://github.com/Packt Publishing/Flutter-débutant> mise à jour du code, ce sera mis à jour sur le référentiel

Nous avons également d'autres ensembles de codes de notre riche catalogue de livres et de vidéos disponibles à <https://github.com/Packt Publishing/Flutter-débutant>

Téléchargez les images couleur

Nous fournissons également un fichier PDF contenant des images en couleur des captures d'écran / diagrammes utilisés dans ce livre. Vous pouvez le télécharger ici: <https://www.packtpub.com/9781788996082/colorimetry>

Conventions utilisées

Il existe un certain nombre de conventions de texte utilisées dans ce livre.

[3]

Page 22

Préface

CodeInText: indique les mots de code dans le texte, les noms de table de base de données, les noms de dossier, les noms de fichiers, extensions de fichier, noms de chemin, URL factices, entrées utilisateur et poignées Twitter. Voici un exemple: "Il évalue et renvoie la valeur de expression2: expression1 ?? expression2. "

Un bloc de code est défini comme suit:

```
principale() {
    var yeahDartIsGreat = "Evidemment!";
    var dartIsGreat = yeahDartIsGreat ?? "Je ne sais pas";
    imprimer(dartIsGreat); // imprime évidemment!
}
```

Lorsque nous souhaitons attirer votre attention sur une partie particulière d'un bloc de code, les lignes pertinentes ou les éléments sont mis en gras:

```
principale() {
    var someInt = 1;
    print(refléter(someInt).type.reflectedType.toString()); // imprime: int
}
```

Toute entrée ou sortie de ligne de commande est écrite comme suit:

code de fléchettes.dart

Gras : indique un nouveau terme, un mot important ou des mots que vous voyez à l'écran. Pour Par exemple, les mots des menus ou des boîtes de dialogue apparaissent dans le texte comme ceci. Voici un exemple: "De plus, le bouton d'action flottant situé en bas devrait vous rediriger vers la **demande de favorisez l'** écran. "

Les avertissements ou remarques importantes apparaissent comme ceci.

Les trucs et astuces apparaissent comme ceci.

[4]

Page 23

Préface

Entrer en contact

Les commentaires de nos lecteurs sont toujours les bienvenus.

Commentaires généraux : si vous avez des questions sur un aspect quelconque de ce livre, mentionnez le livre titre dans le sujet de votre message et envoyez-nous un e-mail à customercare@packtpub.com.

Errata : Bien que nous ayons pris tout le soin d'assurer l'exactitude de notre contenu, des erreurs arriver. Si vous avez trouvé une erreur dans ce livre, nous vous serions reconnaissants de bien vouloir signalez-nous cela. Veuillez visiter www.packtpub.com/support/errata, en sélectionnant votre livre, en cliquant sur le lien Formulaire de soumission d'errata et en saisissant les détails.

Piratage : si vous rencontrez des copies illégales de nos œuvres sous quelque forme que ce soit sur Internet, nous vous seriez reconnaissant de bien vouloir nous fournir l'adresse de l'emplacement ou le nom du site Web. Veuillez nous contacter à copyright@packt.com avec un lien vers le matériel.

Si vous souhaitez devenir auteur : s'il y a un sujet dans lequel vous avez une expertise et que vous souhaitez écrire ou contribuer à un livre, veuillez visiter auteurs.packtpub.com.

Commentaires

Veuillez laisser un commentaire. Une fois que vous avez lu et utilisé ce livre, pourquoi ne pas laisser un avis sur le site sur lequel vous l'avez acheté? Les lecteurs potentiels peuvent alors voir et utiliser votre avis pour prendre des décisions d'achat, chez Packt, nous pouvons comprendre ce que vous pensez de notre produits, et nos auteurs peuvent voir vos commentaires sur leur livre. Merci!

Pour plus d'informations sur Packt, veuillez visiter packt.com.

[5]

Page 24

1

Section 1: Introduction à Dart

Dans cette section, vous comprendrez le cœur du framework Flutter, explorez les bases du langage Dart, apprenez à configurer votre propre environnement et enfin, Apprenez à vous en servir.

Les chapitres suivants sont inclus dans cette section:

- [Chapitre 1](#), *Une introduction à Dart*
- [Chapitre 2](#), *Programmation de fléchettes intermédiaire*
- [chapitre 3](#), *Une introduction au Flutter*

Page 25

1

Une introduction à Dart

Le langage Dart est présent au cœur du framework Flutter. Un cadre moderne comme Flutter nécessite un langage moderne de haut niveau pour être capable de fournir le meilleur expérience au développeur et permettant de créer de superbes applications mobiles. Comprendre Dart est fondamental pour travailler avec Flutter; les développeurs doivent connaître le origines du langage Dart, comment la communauté y travaille, ses atouts et pourquoi est le langage de programmation choisi pour développer avec Flutter. Dans ce chapitre, vous allez passer en revue les bases du langage Dart et recevoir des liens vers des ressources qui peut vous aider dans votre voyage Flutter. Vous passerez en revue les types et opérateurs intégrés de Dart et comment Dart fonctionne avec la **programmation orientée objet (POO)**. En comprenant ce que Le langage Dart fournit, vous serez en mesure d'expérimenter confortablement avec le Dart l'environnement par vous-même et élargissez vos connaissances.

Nous couvrirons les sujets suivants dans ce chapitre:

- Apprendre à connaître les principes et les outils du langage Dart
- Comprendre pourquoi Flutter utilise Dart
- Apprendre les bases de la structure du langage Dart
- Présentation de la POO avec Dart

Premiers pas avec Dart

Le langage Dart, développé par Google, est un langage de programmation qui peut être utilisé pour développer des applications Web, de bureau, côté serveur et mobiles. Dart est la programmation langue utilisée pour coder les applications Flutter, ce qui lui permet de fournir la meilleure expérience au développeur pour la création d'applications mobiles de haut niveau. Alors, explorons ce que Dart fournit et comment cela fonctionne afin que nous puissions appliquer plus tard ce que nous apprenons dans Flutter.

Piste 26

Une introduction à Dart

Chapitre 1

Dart vise à regrouper les avantages de la plupart des langages de haut niveau avec des fonctionnalités linguistiques, notamment les suivantes:

- Outilage productif**: Cela comprend des outils pour analyser le code, le développement intégré les plugins d' environnement (**IDE**) et les grands écosystèmes de paquets.
- Garbage collection** : Cela gère ou traite la désallocation de la mémoire (principalement mémoire occupée par des objets qui ne sont plus utilisés).
- Tapez des annotations** (facultatif): c'est pour ceux qui veulent la sécurité et la cohérence pour contrôler toutes les données d'une application.
- Typé statique** : bien que les annotations de type soient facultatives, Dart est de type sécurisé et utilise l'inférence de type pour analyser les types lors de l'exécution. Cette fonctionnalité est importante pour trouver des bogues pendant la compilation.
- Portabilité** : ce n'est pas seulement pour le Web (transpilé en JavaScript), mais cela peut être compilé nativement en code ARM et x86.

L'évolution de Dart

Dévoilé en 2011, Dart évolue depuis. Dart a vu sa sortie stable en 2013 avec changements majeurs inclus dans la sortie de Dart 2.0 vers la fin de 2018:

- Il était axé sur le développement Web dans sa conception, avec l'objectif principal de remplacement de JavaScript :** Cependant, Dart se concentre désormais sur le développement mobile zones ainsi que sur Flutter.
- Il a essayé de résoudre les problèmes de JavaScript:** JavaScript ne fournit pas la robustesse que de nombreuses langues consolidées le font, alors Dart a voulu apporter une successeur de JavaScript.
- Il offre les meilleures performances et de meilleurs outils pour les projets à grande échelle :** Dart a outils modernes et stables fournis par les plugins IDE. Il a été conçu pour obtenir le les meilleures performances possibles tout en gardant la sensation d'un langage dynamique.
- Il est moulé pour être robuste et flexible :** en gardant les annotations de type facultatives et en ajoutant des fonctionnalités POO, Dart équilibre les deux mondes de la flexibilité et robustesse.

Dart est un excellent langage moderne multiplateforme et polyvalent qui s'améliore continuellement ses caractéristiques, ce qui le rend plus mature et flexible. C'est pourquoi l'équipe du framework Flutter a choisi le langage Dart avec lequel travailler.

[8]

Page 27

Une introduction à Dart

Chapitre 1

Comment fonctionne Dart

Pour comprendre d'où vient la flexibilité de la langue, nous devons savoir comment nous pouvons exécuter le code Dart. Cela se fait de deux manières:

- Machines virtuelles Dart (VM)**
- Compilations JavaScript

Jetez un œil au diagramme suivant:

Compilation de Dart VM et JavaScript

Le code Dart peut être exécuté dans un *environnement compatible Dart*. Un environnement compatible Dart fournit fonctionnalités essentielles d'une application, telles que les suivantes:

- Systèmes d'exécution
- Bibliothèques de base de Dart
- Éboueurs

L'exécution du code Dart fonctionne selon deux modes: la compilation **Just-In-Time (JIT)** ou **Compilation Ahead-Of-Time (AOT)**:

Une compilation JIT est l'endroit où le code source est chargé et compilé en natif code machine par la VM Dart à la volée. Il est utilisé pour exécuter du code dans la commande-interface de ligne ou lorsque vous développez une application mobile afin d'utiliser des fonctionnalités comme le débogage et le recharge à chaud.

[9]

Page 28

Une introduction à Dart

Chapitre 1

Une compilation AOT est l'endroit où la machine virtuelle Dart et votre code sont précompilés et la machine virtuelle fonctionne plus comme un système d'exécution Dart, fournissant un ramasse-miettes et diverses méthodes natives du kit de développement logiciel (**SDK**) Dart à L'application.

Dart contribue à la fonction la plus célèbre de Flutter, le recharge à chaud, qui est basé sur le compilateur Dart JIT, permettant des interactions rapides avec le code en direct swaps. Consultez la section *Comprendre pourquoi Flutter utilise Dart* pour plus de détails.

Fléchette pratique

La façon dont Flutter est conçu est fortement influencée par le langage Dart. Alors, sachant cela la langue est cruciale pour réussir dans le cadre. Commençons par écrire du code dans comprendre les bases de la syntaxe et les outils disponibles pour le développement de Dart.

DartPad

La façon la plus simple de commencer à coder est d'utiliser l'outil DartPad (<https://.. / dartpad>). C'est un excellent outil en ligne pour apprendre et expérimenter les fonctionnalités ling Les bibliothèques de base de Dart, à l'exception des bibliothèques de VM telles que dart: io.

Voici à quoi ressemble l'outil:

[dix]

Page 29

Une introduction à Dart

Chapitre 1

Outils de développement de fléchettes

PartPad est un moyen idéal pour commencer à expérimenter la langue sans aucun effort supplémentaire. PartPad vous offrira bientôt l'apprentissage des choses avancées telles que l'écriture sur des fichiers ou l'utilisation de bibliothèques, vous devrez avoir un environnement de développement configuré pour cela.

Flutter est basé sur Dart et vous pouvez développer du code Dart en ayant un Environnement de développement Flutter. Pour savoir comment configurer un Flutter environnement de développement, il suffit de se référer au site officiel pour le tutoriel d'installation (<https://flutter.dev/>)

Les IDE les plus couramment utilisés pour le développement de Dart et Flutter sont Visual Studio Code ou VS Code (pour le Web et Flutter) et Android Studio ou tout IDE JetBrains tel comme WebStorm (qui est axé sur le Web). Toutes les fonctionnalités Dart de ces IDE sont basées sur les outils officiels, peu importe ce que vous choisissez - les outils fournis seront principalement le même. Le SDK Dart fournit des outils spécialisés pour chaque écosystème de développement, comme comme programmation Web et côté serveur.

Le SDK Dart autonome est livré avec les outils suivants:

| | |
|--|--|
| fléchette (https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette.dart) exécute le code Dart | tonome Dart VM; ce la commande suivante: |
|--|--|

code de fléchettes.dart

| | |
|---|--|
| dart2js (https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette.dart) Compilateur JavaScript. | dartanalyzer (https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette.dart) Ce code aider à détecter les erreurs tôt. |
|---|--|

Lint, ou un linter, est un outil qui analyse le code source pour signaler les erreurs, les bogues, erreurs stylistiques et constructions suspectes.

| | | |
|--|--|--|
| dartdoc (https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette.dart) la documentation de réfé | pub (https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette.dart) qui peut être utilis | dartfmt (https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette.dart) directives de style au coc |
|--|--|--|

[11]

Page 30

Une introduction à Dart

Chapitre 1

Pour le développement web Dart ajoute des outils (avec des étapes d'installations supplémentaires à l' [adresse https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette](https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette)).

webdev (<https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette>) runner utilisé dans les tests ou lorsque plus de configuration est requise que ce que fournit webdev.
dartdevc (<https://github.com/dart-lang/sdk/blob/master/bin/dev/flechette>) Dart-intégration de type compil

dart2js est également un outil Web, bien qu'il soit livré avec la norme SDK. Pour le développement côté serveur, les outils SDK standard sont les seuls ceux dont nous avons besoin.

Tous les plugins IDE utilisent ces outils dans les coulisses, vous pouvez donc profiter de la ensemble d'outils complet pour le développement de Dart.

Bonjour le monde

Le code suivant est un script Dart de base, alors jetons un coup d'œil:

```
main () { // le point d'entrée d'une application Dart
    var a = 'monde'; // déclaration et initialisation de la variable
    print ('bonjour $ a'); // appelle la fonction à imprimer pour afficher la sortie
}
```

Ce code contient quelques fonctionnalités de base du langage qui doivent être mises en évidence:

Chaque application Dart doit avoir une fonction de premier niveau de point d'entrée (vous pouvez au [chapitre 2](#), *Programmation de fléchettes intermédiaire*, pour plus d'informations sur fonctions), c'est-à-dire la fonction main ().

Si vous choisissez d'exécuter ce code localement sur votre machine préconfigurée avec Dart SDK, enregistrez le contenu dans un fichier Dart, puis exécutez-le avec un outil Dart dans un terminal, par exemple, lancez hello_world.dart. Cela exécutera la fonction principale du script Dart.

Comme nous l'avons vu précédemment, bien que Dart soit de type sécurisé, les annotations de type sont optionnel. Ici, nous déclarons une variable sans type et affectons un littéral String à il.

[12]

Piste 31

Une introduction à Dart

Chapitre 1

Un littéral String peut être entouré de guillemets simples ou doubles, par exemple, «bonjour le monde» ou «bonjour le monde». Pour afficher la sortie sur la console, vous pouvez utiliser la fonction print () (qui est une autre fonction de niveau supérieur). Avec la technique d'interpolation de chaîne, l'instruction \$ a dans un littéral String résout la valeur de la variable a. Dart appelle la méthode toString () de l'objet.

Nous explorerons plus en détail l'interpolation de chaîne plus loin dans ce chapitre, dans la Section *types et variables de fléchettes*, lorsque nous parlons du type de chaîne.

Nous pouvons utiliser la syntaxe // comment pour écrire des commentaires sur une seule ligne. Dart a aussi commentaires multilignes avec la syntaxe /* comment */ , comme suit:

```
// ceci est un commentaire sur une seule ligne

/*
    Ceci est un long commentaire multiligne
*/
```

Notez le type de retour de la fonction principale; comme il a été omis dans l'exemple, il suppose que type dynamique spécial, que nous explorerons plus tard.

Comprendre pourquoi Flutter utilise Dart

Le framework Flutter vise à changer la donne dans le développement d'applications mobiles, en fournissant tous les outils nécessaires au développeur pour créer des applications géniales sans inconvénient en termes de performances et d'évolutivité. Flutter a, dans sa structure de base, plusieurs concepts axés sur les performances de l'application et l'interface utilisateur. Pour offrir le meilleur du monde du développement avec des performances élevées comparables à celles des SDK natifs officiels, Flutter utilise le support de Dart pour fournir des outils contribuant à la productivité des développeurs en phase de développement et pour créer des applications optimisées pour la publication.

Comme nous l'avons vu précédemment dans la section *Premiers pas avec Dart*, Dart est suffisamment mature et robuste avec de nombreux outils qui contribuent au succès de Flutter. Comprendons pourquoi Dart était le choix parfait pour le cadre Flutter.

[13]

Piste 32*Une introduction à Dart**Chapitre 1*

Ajouter de la productivité

Dart n'est pas seulement un langage, du moins pas dans le concept. Le SDK Dart est livré avec un ensemble d'outils (vu dans la section précédente sur *les outils de développement de Dart*) dont Flutter bénéficie pour aider avec tâches courantes pendant la phase de développement, telles que les suivantes:

Les compilateurs Dart JIT et AOT

Profilage, débogage et journalisation avec Dart DevTools et Observatory
(plus dans [Chapitre 12](#), *Test, débogage et déploiement*).

Analyse de code statique avec son analyseur intégré: <https://.../> fléchette de langue/[Options](#)

Compilation d'applications Flutter et recharge à chaud

Lorsque vous écrivez ou déboguez du code, vous utiliserez la machine virtuelle Dart avec JIT. CA aide pour utiliser des fonctionnalités telles que les outils de profilage, le recharge à chaud (vous pouvez vous référer au [chapitre 3](#), *Une Introduction à Flutter*), et plus encore.

Lors de la création de votre application pour la publication, le code sera compilé dans AOT et votre application livré avec une version minuscule de la machine virtuelle Dart (qui ressemble plus à une bibliothèque d'exécution) avec Dart Capacités du SDK telles que les bibliothèques principales et les garbage collector.

Cette différence, au premier abord, ne semble pas importante du point de vue du développeur, car nous voulons simplement écrire et exécuter l'application, non? Cependant, quand il s'agit de productivité, cela devient l'un des atouts de Dart les plus fondamentaux utilisés par Flutter.

Le **recharge à chaud** de Flutter est l'une de ses caractéristiques les plus célèbres et montre la productivité promise en action. Il s'appuie sur la compilation JIT pour effectuer des échanges de code Dart en direct lors de l'exécution de l'application, afin que nous puissions changer notre code d'application et voir le résultat presque en temps réel. Avec IDE plugins, cela devient encore plus rapide car, après avoir enregistré une modification, le plugin distribue le recharge et le résultat est vu rapidement.

Dans le [chapitre 3](#), *Une introduction à Flutter*, nous allons vérifier le recharge à chaud et d'autres fonctionnalités plus en détail.

Aucune image ne peut décrire le potentiel de cette fonctionnalité incroyable. Donc, après avoir vérifié [chapitre 3](#), *Une introduction à Flutter*, je vous suggère d'exécuter le projet de démarrage Flutter pour avoir premier contact avec cette fonctionnalité incroyable.

[14]

Piste 33*Une introduction à Dart**Chapitre 1*

L'analyseur de fléchettes est un autre outil très cool de Dart:

Cet outil aide à comprendre les problèmes potentiels avec les types et la syntaxe recommandée avant d'exécuter le code.

DevTools ajoute également une valeur importante à la productivité offerte par le Cadre Flutter; en savoir plus au [chapitre 12, Test, Débogage et déploiement](#).

Apprentissage facile

Dart est un nouveau langage pour de nombreux développeurs, et l'apprentissage d'un nouveau framework et d'un nouveau langage en même temps peut être difficile. Cependant, Dart simplifie cette tâche en ne réinventer les concepts, juste les affiner et essayer de les rendre aussi efficaces que possible pour les tâches désignées.

[15]

Piste 34

Une introduction à Dart

Chapitre 1

Dart est inspiré de nombreux langages modernes et matures tels que Java, JavaScript, C #, Swift, et Kotlin, comme vous pouvez le voir ici:

Dans cet esprit, lire le code Dart, même sans connaître profondément la langue, est possible. Jetez également un œil à la page de démarrage de la documentation officielle:

[16]

Piste 35

Une introduction à Dart

Chapitre 1

La documentation et les guides fournis sont des matériaux très propres et informatifs, aussi, le awesome community aide le développeur à apprendre sans mal de tête.

Consultez les guides officiels Dart sur l' apprentissage: <https://fléchette.dev>

Maturité

En dépit d'être une langue relativement nouvelle, Dart n'est pas pauvre ou manque de ressources. Sur le au contraire, en version 2, il dispose déjà de diverses ressources en langage moderne qui aident le développeur pour écrire un code efficace de haut niveau.

Une fonctionnalité parfaite pour illustrer cela est la fonction d' **attente asynchrone** :

Cela permet au développeur d'écrire des appels non bloquants avec une syntaxe très simple, permettant l'application pour continuer à rendre sans inconvénients.

[17]

Piste 36*Une introduction à Dart**Chapitre 1*

Alors que Dart se concentre sur le développeur, une autre chose importante pour les développeurs mobiles et Web construit des interfaces utilisateur. Dans cet esprit, la syntaxe de Dart est facile à comprendre lorsque vous pensez en termes d'interface utilisateur. Voyons un exemple:

Ces captures d'écran sont tirées du site officiel de Dart: [dart.dev.](https://dart.dev/)

L'opérateur **collection if** qui peut être vu dans la capture d'écran précédente est un excellent exemple d'une nouvelle fonctionnalité facile à comprendre même si vous êtes nouveau sur Dart.

Dart évolue aux côtés de Flutter, et ce ne sont que quelques-uns des atouts importants du langage fournit au cadre. Tant que vous réalisez que Dart est facile à apprendre et comment beaucoup cela contribue au pouvoir de Flutter, le défi d'apprendre une nouvelle langue ensemble avec un nouveau cadre devient plus facile et même agréable.

Dans ce livre, nous n'allons pas plonger trop profondément dans les détails de la syntaxe de Dart. Vous pouvez vérifier le code source de ce chapitre sur GitHub pour des exemples de syntaxe et utilisez-le comme étude guide ou un parcours d'apprentissage de la langue. Plus tard, vous pouvez explorer une syntaxe ou des fonctionnalités spécifiques pendant que vous avancez dans votre voyage avec le framework Flutter.

[18]

Piste 37*Une introduction à Dart**Chapitre 1*

Présentation de la structure du Dart Langue

Si vous connaissez déjà certains langages de programmation inspirés de l'ancien langage C ou si vous avez une certaine expérience de JavaScript, une grande partie de la syntaxe Dart vous sera facile à comprendre. Dart fournit les opérateurs les plus courants pour manipuler les variables. Ses types intégrés sont les plus courants trouvés dans les langages de programmation de haut niveau, avec quelques particularités. En outre, les flux et les fonctions de contrôle sont très similaires à ceux typiques. Allons Passez en revue une partie de la structure du langage de programmation Dart avant de plonger dans Flutter.

Si vous connaissez déjà Dart, vous pouvez utiliser cette section pour revoir la syntaxe de Dart; sinon, vous pouvez consulter cette introduction et vous référer à la visite linguistique de Dart pour une

guide rapide et facile d'apprentissage sur Dart: <https:///. / fléchette tour .>

Opérateurs de fléchettes

Dans Dart, les opérateurs ne sont rien de plus que des méthodes définies dans des classes avec une syntaxe particulière. Ainsi, lorsque vous utilisez des opérateurs tels que `x == y`, c'est comme si vous invoquez le `x. == (y)` méthode pour comparer l'égalité.

Comme vous l'avez peut-être noté, nous invoquons une méthode sur `x`, ce qui signifie `x` est une instance d'une classe qui a des méthodes. Dans Dart, tout est un objet exemple; tout type que vous définissez est également une instance Object. Il y a plus sur cela dans la section *Introduction à la POO dans Dart*.

Ce concept signifie que les opérateurs peuvent être remplacés afin que vous puissiez écrire votre propre logique pour eux. Encore une fois, si vous avez une certaine expérience en Java, C #, JavaScript ou des langages similaires, vous pouvez ignorer la plupart des opérateurs, car ils sont très similaires dans plusieurs langues.

Nous n'allons pas entrer dans tous les détails spécifiques de la syntaxe de Dart dans ce livre. Vous pouvez vous référer au code source sur GitHub pour de nombreux exemples sur le Syntaxe de Dart.

[19]

Piste 38

Une introduction à Dart

Chapitre 1

Dart a les opérateurs suivants:

- Arithmétique
- Incrémenter et décrémenter
- Égalité et relationnel
- Vérification de type et moulage
- Opérateurs logiques
- Manipulation des bits
- Null-safe et null-aware (les langages de programmation modernes fournissent cet opérateur pour faciliter la gestion des valeurs nulles)

Examinons chacun d'eux plus en détail.

Opérateurs arithmétiques

Dart est livré avec de nombreux opérateurs typiques qui fonctionnent comme de nombreux langages; cela inclut le Suivant:

- `+`: C'est pour l'addition de nombres.
- `-`: C'est pour la soustraction.
- `*`: Ceci est pour la multiplication.
- `/`: C'est pour la division.
- `~/`: Ceci est pour la division entière. Dans Dart, toute division simple avec `/` entraîne un double valeur. Pour obtenir uniquement la partie entière, vous devez créer une sorte de transformation (c'est-à-dire de type cast) dans d'autres langages de programmation; cependant, ici, l'opérateur de division entier effectue cette tâche.
- `%`: Ceci est pour les opérations modulo (le reste de la division entière).
- `-expression`: C'est pour la négation (qui inverse le signe de l'expression).

Certains opérateurs ont un comportement différent selon le type d'opérande de gauche; par exemple,

L'opérateur + peut être utilisé pour additionner des variables de type num, mais aussi pour concaténer des chaînes. C'est parce qu'ils ont été implémentés différemment dans les classes correspondantes comme indiqué avant.

Dart fournit également des opérateurs de raccourci pour combiner une affectation à un variable après une autre opération. Le raccourci arithmétique ou d'affectation les opérateurs sont +=, -=, *=, /= et ~/=.

[20]

Piste 39

Une introduction à Dart

Chapitre 1

Opérateurs d'incrémantation et de décrémantation

Les opérateurs d'incrémantation et de décrémantation sont également des opérateurs courants et sont implémentés en type de nombre, comme suit:

++ var ou var ++ pour incrémenter / dans var
--var ou var-- pour décrémenter / de var

Les opérateurs d'incrémantation et de décrémantation de Dart n'ont rien de différent du standard langues. Une bonne application des opérateurs d'incrémantation et de décrémantation est pour count opérations sur les boucles.

Égalité et opérateurs relationnels

Les opérateurs d'égalité Dart sont les suivants:

==: Pour vérifier si les opérandes sont égaux
!=: Pour vérifier si les opérandes sont différents

Pour les tests relationnels, les opérateurs sont les suivants:

>: Pour vérifier si l'opérande gauche est *supérieur* à l' opérande droit
<: Pour vérifier si l'opérande gauche est *inférieur* à l' opérande droit
>=: Pour vérifier si l'opérande gauche est *supérieur ou égal* à l' opérande droit
=<: Pour vérifier si l'opérande gauche est *inférieur ou égal* à l' opérande droit

Dans Dart, contrairement à Java et à de nombreux autres langages, l'opérateur == ne comparer les références mémoire mais plutôt le contenu de la variable.

Vérification de type et moulage

Dart a une saisie facultative, comme vous le savez déjà, donc les opérateurs de vérification de type peuvent être utiles pour vérifier les types lors de l'exécution:

is: pour vérifier si l'opérande a le type testé
is!: Pour vérifier si l'opérande n'a pas le type testé

[21]

Piste 40

Une introduction à Dart

Chapitre 1

La sortie de ce code sera différente selon le contexte de l'exécution. Dans DartPad, la sortie est vraie pour le contrôle du type double; cela est dû à la façon dont JavaScript traite les nombres et, comme vous le savez déjà, Dart pour le Web est précompilé pour JavaScript pour exécution sur les navigateurs Web.

Il y a aussi le mot-clé `as`, qui est utilisé pour le typage d'un supertype vers un sous-type, comme la conversion de `num` en `int`.

Le mot-clé `as` est également utilisé pour spécifier un préfixe pour les bibliothèques, en utilisant importations (vous pouvez en savoir plus à ce sujet dans [Chapitre 2, Fléchette intermédiaire Programmation](#)).

Opérateurs logiques

Les opérateurs logiques dans Dart sont les opérateurs courants appliqués aux opérandes booléens; ils peuvent être variables, expressions ou conditions. De plus, ils peuvent être combinés avec des expressions en combinant les résultats des expressions. Les opérateurs logiques fournis sont comme suit:

`!`: expression: pour nier le résultat d'une expression, c'est-à-dire vrai à faux et faux à vrai

`||`: Pour appliquer un OU logique entre deux expressions

`&&`: pour appliquer un ET logique entre deux expressions

Manipulation des bits

Dart fournit des opérateurs de bits et de décalage pour manipuler des bits individuels de nombres, généralement avec le type `num`. Ils sont les suivants:

`&`: Pour appliquer un ET logique aux opérandes, en vérifiant si les bits correspondants sont les deux 1

`|`: Pour appliquer un OU logique aux opérandes, en vérifiant si au moins un des bits correspondants sont 1

`^`: Pour appliquer un XOR logique aux opérandes, en vérifiant si un seul mais pas les deux des bits correspondants est 1

`~` opérande: pour inverser les bits de l'opérande, par exemple 1 , devenant 0 , et 0 , devenant 1 .

[22]

Piste 41

Une introduction à Dart

Chapitre 1

`<<`: Pour décaler l'opérande gauche de x bits vers la gauche (cela décale de 0 . de la droite)

`>>`: Pour décaler l'opérande gauche de x bits vers la droite (en supprimant les bits du la gauche)

Comme les opérateurs arithmétiques, les opérateurs au niveau du bit ont également des opérateurs d'affectation de raccourcis, et ils fonctionnent exactement de la même manière que ceux présentés précédemment; ils sont `<<=`, `>>=`, `&=`,

`^ =` et `| =`.

Opérateurs à sécurité nulle et prenant en charge les valeurs nulles

Suivant la tendance des langages OOP modernes, Dart fournit une syntaxe null-safe qui évalue et renvoie une expression en fonction de sa valeur nulle / non nulle.

L'évaluation fonctionne de la manière suivante: si `expression1` n'est pas nulle, elle renvoie sa valeur; sinon, il évalue et renvoie la valeur de `expression2`: `expression1 ?? expression2`.

En plus de l'opérateur d'affectation commun, `=` et ceux répertoriés dans le

opérateurs, Dart fournit également une combinaison entre l'affectation et la expression; autrement dit, l'opérateur ?? =, qui affecte une valeur à une variable uniquement si son la valeur est nulle.

Dart fournit également un opérateur d'accès prenant en charge les valeurs nulles,?. Qui empêche d'accéder à un objet nul membres.

Types et variables de fléchettes

Vous savez probablement déjà comment déclarer une variable simple, c'est-à-dire en utilisant le var mot-clé suivi du nom. Une chose à noter est que lorsque nous n'avons pas spécifié la valeur initiale de la variable, elle a supposé null quel que soit son type.

final et const

Une variable n'aura jamais l'intention de changer sa valeur après son affectation, et vous pouvez utiliser les moyens finaux et const pour déclarer ceci:

```
valeur finale = 1;
```

[23]

Piste 42

Une introduction à Dart

Chapitre 1

La variable value ne peut pas être modifiée une fois initialisée:

```
valeur const = 1;
```

Tout comme le mot-clé final, la variable value ne peut pas être modifiée une fois initialisée, et son initialisation doit avoir lieu avec une déclaration.

En plus de cela, le mot clé const définit une constante de compilation. En temps de compilation constante, les valeurs const sont connues au moment de la compilation. Ils peuvent également être utilisés pour faire instances d'objets ou listes immuables, comme suit:

```
liste const = const [1, 2, 3]
// et
point const = point const (1,2)
```

Cela définira la valeur des deux variables pendant la compilation, les transformant complètement en variables immuables.

Types intégrés

Dart est un langage de programmation de type sécurisé, les types sont donc obligatoires pour les variables. Bien que les types sont obligatoires, les annotations de type sont facultatives, ce qui signifie que vous n'avez pas besoin de spécifier le type d'une variable lors de sa déclaration. Dart effectue l'inférence de type, et nous allons examiner davantage dans l'*inférence de type - apportant du dynamisme à la section show*.

Voici les types de données intégrés dans Dart:

- Nombres (tels que num, int et double)
- Booléens (tels que bool)
- Collections (telles que listes, tableaux et cartes)
- Chaînes et runes (pour exprimer des caractères Unicode dans une chaîne)

Nombres

Dart représente les nombres de deux manières:

Int : valeurs entières non fractionnaires signées 64 bits telles que -2⁶³ à 2⁶³-1.

Double : Dart représente des valeurs numériques fractionnaires avec une double précision de 64 bits

nombre à virgule flottante.

Les deux étendent le type num. De plus, nous avons de nombreuses fonctions pratiques dans la fléchette: bibliothèque mathématique pour aider aux calculs.

[24]

Piste 43

Une introduction à Dart

Chapitre 1

En JavaScript, les nombres sont compilés en nombres JavaScript et permettent valeurs -2₃₃ à 2₃₃-1.

De plus, notez que les types num, double et int ne peuvent pas être étendus ou mis en œuvre.

BigInt

Dart a également le type BigInt pour représenter des entiers de précision arbitraires, ce qui signifie que la limite de taille est la RAM de l'ordinateur en cours d'exécution. Ce type peut être très utile selon sur le contexte; cependant, il n'a pas les mêmes performances que les types num et vous devrait en tenir compte au moment de décider de l'utiliser.

JavaScript a le concept d'entiers sûrs, que Dart suit lors de la transpilation. Cependant, comme JavaScript utilise la double précision pour représenter des entiers pairs, nous n'avons pas débordement lors de l'exécution (maxInt * 2).

Maintenant, vous pourriez envisager de placer BigInt partout où vous utiliseriez des entiers pour être libre de déborde, mais rappelez-vous que BigInt n'a pas les mêmes performances que les types int, le rendant inadapté à tous les contextes.

De plus, si vous voulez savoir comment Dart VM gère les nombres en interne, jetez un œil à la section *Lectures complémentaires* à la fin de ce chapitre.

Booléens

Dart fournit les deux valeurs littérales bien connues pour le type booléen: true et false.

Les types booléens sont des valeurs de vérité simples qui peuvent être utiles pour n'importe quelle logique. Une chose que tu peux ont remarqué, mais que je veux renforcer, concerne les expressions.

Nous savons déjà que les opérateurs, tels que > ou ==, par exemple, ne sont rien de plus que méthodes avec une syntaxe spéciale définie dans les classes et, bien sûr, elles ont une valeur de retour qui peut être évalué dans des conditions. Donc, le type de retour de toutes ces expressions est bool et, comme vous le savez déjà, les expressions booléennes sont importantes dans tout langage de programmation.

Les collections

Dans Dart, les listes sont considérées comme identiques aux tableaux dans d'autres langages de programmation avec quelques méthodes pratiques pour manipuler des éléments.

[25]

Piste 44

Une introduction à Dart

Chapitre 1

Les listes ont l'opérateur [index] pour accéder aux éléments à l'index donné et, en plus, le + l'opérateur peut être utilisé pour concaténer deux listes en retournant une nouvelle liste avec l'opérande de gauche suivi du bon.

Une autre chose importante à propos des listes Dart est la contrainte de *longueur*. C'est de la façon dont nous définir les listes précédentes, en les faisant grossir selon les besoins en utilisant la méthode add, qui grandira pour ajouter l'élément.

Une autre façon de définir la liste consiste à définir sa longueur lors de sa création. Listes avec une taille fixe ne peut pas être étendu, il est donc de la responsabilité du développeur de savoir où et quand utiliser listes de taille fixe, car il peut lever des exceptions si vous essayez d'ajouter ou d'accéder à des éléments non valides.

Les **cartes** de fléchettes sont des collections dynamiques pour stocker des valeurs sur une base clé, où la récupération et la modification d'une valeur est toujours effectuée en utilisant sa clé associée. À la fois la clé et la valeur peut avoir n'importe quel type; si nous ne spécifions pas les types clé-valeur, ils seront déduits par Dart as Map <dynamic, dynamic>, avec ses clés et ses valeurs de type dynamique. Bien expliquer plus sur les types dynamiques plus tard.

Cordes

Dans Dart, les chaînes sont une séquence de caractères (code UTF-16) qui sont principalement utilisées pour représenter texte. Les chaînes de fléchettes peuvent être des lignes simples ou multiples. Vous pouvez faire correspondre des guillemets simples ou doubles (généralement pour les lignes simples) et les chaînes multilignes en faisant correspondre les guillemets triples.

Nous pouvons utiliser l'opérateur + pour concaténer des chaînes. Le type chaîne implémente opérateurs utiles autres que le plus (+). Il implémente l'opérateur multiplicateur (*) où la chaîne est répétée un nombre spécifié de fois, et l'opérateur [index] récupère le caractère à la position d'index spécifiée.

Interpolation de chaîne

Dart a une syntaxe utile pour interpoler la valeur des expressions Dart dans les chaînes: \$ {}, qui fonctionne comme suit:

```
principale() {
    String someString = "Ceci est une chaîne";
    print ("La valeur de la chaîne est: $ someString");
    // imprime La valeur de la chaîne est: Ceci est une chaîne

    print ("La longueur de la chaîne est: $ {someString.length}");
    // imprime La longueur de la chaîne est: 16
}
```

[26]

Piste 45

Une introduction à Dart

Chapitre 1

Comme vous l'avez peut-être remarqué, lorsque nous insérons juste une variable et non une expression value dans la chaîne, nous pouvons omettre les accolades et simplement ajouter directement \$ identifier.

Dart a également le concept de **runes** pour représenter les bits UTF-32. Pour plus de détails, consultez la visite linéistique de Dart: <https://. / fléchette . anglais- tournée>

Littéraux

Vous pouvez utiliser les syntaxes [] et {} pour initialiser des variables telles que des listes et des cartes, respectivement. Voici quelques exemples de littéraux fournis par le langage Dart pour création d'objets des types intégrés fournis:

| Type | Exemple littéral |
|---------|---|
| int | 10, 1, -1, 5 et 0 |
| double | 10.1, 1.2, 3.123 et -1.2 |
| booléen | vrai et faux |
| Chaîne | "Dart", "Dash" et "" "chaîne multiligne" "" |
| liste | [1, 2, 3] et ["un", "deux", "trois"] |
| Carte | {"key1": "val1", "b": 2} |

Un littéral est une notation pour représenter une valeur fixe en programmation langues. Vous en avez probablement déjà utilisé certains auparavant.

Inférence de type - apportant du dynamisme au spectacle

Dans les exemples précédents, nous avons montré deux façons de déclarer des variables: en utilisant le type de la variable, comme int et String, ou en utilisant le mot-clé var.

Alors, maintenant vous vous demandez peut-être comment Dart sait de quel type de variable il s'agit si vous ne le faites pas spécifiez-le dans une déclaration.

[27]

Piste 46

Une introduction à Dart

Chapitre 1

De la documentation Dart (<https://dart.dev/guides/language/documentation>), tenez compte de la déclaration suivante.

"L'analyseur peut déduire des types pour les champs, les méthodes, les variables locales et le type le plus générique arguments. Lorsque l'analyseur n'a pas assez d'informations pour déduire un type spécifique, il utilise le type dynamique."

Cela signifie que, lorsque vous déclarez une variable, l'analyseur Dart déduit le type en fonction de le littéral ou le constructeur d'objet.

Voici un exemple:

```
import 'dart: mirrors';

principale() {
    var someInt = 1;
    print(refléter(someInt).type.reflectedType.toString()); // imprime: int
}
```

Comme vous pouvez le voir, dans cet exemple, nous n'avons que le mot-clé var. Nous n'avons spécifié aucun type, mais comme nous avons utilisé un littéral int (1), l'outil d'analyse pouvait déduire le type avec succès.

Les variables locales obtiennent le type déduit par l'analyseur lors de l'initialisation. Dans le précédent exemple, essayer d'attribuer une valeur de chaîne à someInt échouerait.

Alors, considérons le code suivant:

```
principale() {
    var a; // ici nous n'avons pas initialisé var donc c'est
           // type est la dynamique spéciale
    a = 1; // maintenant a est un int
    a = "a"; // et maintenant une chaîne

    print(a est un entier); // imprime faux
    print(a est une chaîne); // imprime vrai
    print(a est dynamique); // imprime vrai
    print(a.runtimeType); // imprime la chaîne
}
```

Comme vous l'avez peut-être remarqué, a est un type String et un type dynamique. La dynamique est une spéciale type et il peut prendre n'importe quel type au moment de l'exécution; par conséquent, toute valeur peut être convertie en dynamique aussi.

Dart peut déduire des types pour les champs, les retours de méthode et les arguments de type générique; nous explorerons chacun plus en détail dans leurs sections respectives de ce livre.

[28]

Piste 47*Une introduction à Dart**Chapitre 1*

L'analyseur Dart fonctionne également sur les collections et les génériques; pour la carte et lister des exemples dans ce chapitre, nous avons utilisé l'initialiseur littéral pour les deux, donc leurs types ont été déduits.

Contrôle des flux et des boucles

Nous avons examiné comment utiliser les variables et les opérateurs Dart pour créer des expressions conditionnelles. Pour travailler avec des variables et des opérateurs, nous devons généralement implémenter un flux de contrôle pour faire en sorte que notre code Dart prenne la direction appropriée dans notre logique.

Dart fournit une syntaxe de flux de contrôle qui est très similaire à d'autres programmes langues; c'est comme suit:

- sinon
- interrupteur / boîtier
- Boucle avec for, while et do-while
- pause et continue
- affirme
- Exceptions avec try / catch et throw

La syntaxe Dart de ces flux de contrôle n'a pas de particularité importante doit être revu en détail. Veuillez vous référer à la visite des langues officielles sur les flux de contrôle pour Détails: <https:///. / fléch>

Les fonctions

Dans Dart, Function est un type, comme String ou num. Cela signifie qu'ils peuvent également être affectés aux champs ou aux variables locales ou transmis en tant que paramètres à d'autres fonctions; Prendre en compte exemple suivant:

```
String sayHello () {
    retourne "Hello world!";
}

void main () {
    var sayHelloFunction = sayHello; // affectation de la fonction
                                    // à la variable
    print (sayHelloFunction()); // imprime Hello world!
}
```

[29]

Piste 48*Une introduction à Dart**Chapitre 1*

Dans cet exemple, la variable sayHelloFunction stocke la fonction sayHello elle-même et ne l'invoque pas. Plus tard, nous pouvons l'invoquer en ajoutant () au nom de la variable tout comme même si c'était une fonction.

Essayer d'appeler une variable sans fonction peut entraîner une erreur du compilateur.

Le type de retour de fonction peut également être omis, de sorte que l'analyseur Dart déduit le type de l'instruction retour. Si aucun retour n'est fourni, une supposition return null. Si vous voulez lui dire qu'il n'a pas de retour, vous devez le marquer comme nul:

```
sayHello () { // Le type de retour est toujours String
    retourne "Hello world!";
}
```

Une autre façon d'écrire cette fonction consiste à utiliser la syntaxe abrégée, () => expression ;, qui est également appelée fonction Flèche ou fonction Lambda:

```
sayHello () => "Bonjour tout le monde!";
```

Vous ne pouvez pas écrire d'instructions à la place de l'expression, mais vous pouvez déjà utiliser les expressions conditionnelles connues (c'est-à-dire?: ou ??).

Dans cet exemple, la fonction sayHello est une fonction de niveau supérieur. En d'autre mots, il n'a pas besoin d'une classe pour exister. Bien que Dart soit un objet-orienté langage, il n'est pas nécessaire d'écrire des classes pour encapsuler les fonctions.

Paramètres de fonction

Une fonction peut avoir deux types de paramètres: **facultatifs** et **obligatoires**. De plus, comme avec la plupart des langages de programmation modernes, ces paramètres peuvent être nommés sur appel pour rendre le code plus lisible.

[30]

Piste 49

Une introduction à Dart

Chapitre 1

Le type de paramètre n'a pas besoin d'être spécifié; dans ce cas, le paramètre suppose le type **dynamique**:

Paramètres obligatoires: cette simple définition de fonction avec paramètres est en les définissant simplement de la même manière que la plupart des autres langues. dans le fonction suivante, le nom et le message supplémentaire sont requis paramètres, l'appelant *doit donc les* transmettre lors de son appel:

```
sayHello (String name, String additionalMessage) => "Bonjour $ name.
$ additionalMessage ";
```

Paramètres de position facultatifs: parfois, tous les paramètres ne doivent pas être obligatoire pour une fonction, elle peut donc également définir des paramètres facultatifs. le la définition facultative des paramètres de position s'effectue à l'aide de la syntaxe []. Les paramètres de position facultatifs doivent aller après tous les paramètres requis, comme suit:

```
sayHello (String name, [String additionalMessage]) => "Bonjour $ name.
$ additionalMessage ";
```

Si vous exécutez le code précédent sans passer une valeur pour additionalMessage, vous verrez null à la fin de la chaîne renvoyée. Lorsque le paramètre facultatif n'est pas spécifié, la valeur par défaut est nulle sauf si vous spécifiez des valeurs par défaut pour leur:

```
void main () {
    print (sayHello ('mon ami')); // Bonjour, mon ami. nul
    print (sayHello ('mon ami', "Comment vas-tu?"));
    // imprime Bonjour mon ami. Comment ça va?
}
```

Pour définir une valeur par défaut pour un paramètre, vous l'ajoutez après le signe = juste après le définition des paramètres:

```
sayHello (String name, [String additionalMessage = "Bienvenue dans Dart
Fonctions! "]) => "Bonjour $ name. $ additionalMessage";
```

Ne pas spécifier le paramètre entraîne l'impression du message par défaut, comme suit:

```
void main () {
    var bonjour = dire bonjour ('mon ami');
    imprimer (bonjour);
}
```

[31]

Piste 50

Une introduction à Dart

Chapitre 1

Paramètres nommés facultatifs : la définition du paramètre nommé facultatif est terminée en utilisant la syntaxe {}. Ils doivent également aller après tous les paramètres requis:

```
sayHello (String name, {String additionalMessage}) => "Bonjour $ name.
$ additionalMessage";
```

L'appelant doit spécifier le nom du paramètre nommé facultatif, comme suit:

```
void main () {
    print (sayHello ('mon ami'));
    // il reste facultatif, imprime: Bonjour mon ami. nul

    print (sayHello ('mon ami', additionalMessage: "Comment vas-tu?"));
    // imprime: Bonjour mon ami. Comment ça va?
}
```

Les paramètres nommés ne sont pas exclusifs aux paramètres facultatifs; faire un nommé paramètre un paramètre obligatoire, vous pouvez le marquer avec @required:

```
sayHello (String name, {@required String additionalMessage}) =>
"Bonjour $ name. $ AdditionalMessage";
```

Là encore, l'appelant doit spécifier le nom du paramètre nommé requis:

```
void main () {
    var hello = sayHello ('mon ami', additionalMessage: "Comment
tu?");
    // ne pas spécifier le nom du paramètre entraînera un indice sur
    // l'éditeur, ou en exécutant dartanalyzer manuellement sur la console

    imprimer (bonjour); // imprime "Bonjour mon ami. Comment vas-tu?"
}
```

Fonctions anonymes : les **fonctions** Dart sont des objets et elles peuvent être transmises comme paramètres à d'autres fonctions. Nous l'avons déjà vu lors de l'utilisation de forEach () fonction de l'itérable.

Une fonction anonyme est une fonction qui n'a pas de nom; on l'appelle aussi *lambda* ou *fermeture*. La fonction forEach () en est un bon exemple; nous devons le faire lui passer une fonction qui sera exécutée avec chacune des collections de listes éléments:

```
void main () {
    var list = [1, 2, 3, 4];
    list.forEach ((number) => print ('bonjour $ nombre'));
}
```

[32]

Piste 51*Une introduction à Dart**Chapitre 1*

Notre fonction anonyme reçoit un élément mais ne spécifie pas de type; alors, c'est juste imprime la valeur reçue par le paramètre.

Portée lexicale : la portée de Dart est déterminée par la disposition du code en utilisant curly accolades comme de nombreux langages de programmation; les fonctions internes peuvent accéder variables jusqu'au niveau global:

```
globalFunction () {
    print ("fonction globale / de niveau supérieur");
}

simpleFunction () {
    print ("fonction simple");
    globalFunction () {
        print ("Pas vraiment global");
    }
}

globalFunction ();

principale() {
    simpleFunction ();
    globalFunction ();
}
```

Si vous examinez le code précédent, la fonction globalFunction de simpleFunction être utilisé à la place de la version globale, car il est défini localement sur sa portée.

Dans la fonction principale, en revanche, la version globale de la fonction globalFunction est utilisée, car, dans ce cadre, la fonction interne globalFunction de simpleFunction n'est pas défini.

Structures de données, collections et génériques

Dart fournit plusieurs types de structures pour manipuler des valeurs. Les listes de fléchettes sont largement utilisé même dans les cas d'utilisation les plus simples. Les génériques sont un concept lorsque l'on travaille avec collections de données liées à un type spécifique, tel que List ou Map, par exemple. Ils assurent un la collection aura des valeurs homogènes en spécifiant le type de données qu'elle peut contenir.

[33]

Piste 52*Une introduction à Dart**Chapitre 1*

Génériques

La syntaxe `<..>` est utilisée pour spécifier le type pris en charge par une collection. Si vous regardez le exemples précédents de listes et de cartes, vous remarquerez que nous n'avons spécifié aucun type. En effet, ils sont facultatifs et Dart peut déduire le type en fonction des éléments lors de la initialisation de la collection.

Consultez le code source de ce chapitre sur GitHub pour des exemples sur les collections et génériques. N'oubliez pas que si l'outil d'analyse Dart ne peut pas déduire le type, il suppose le type dynamique.

Quand et pourquoi utiliser des génériques

L'utilisation de génériques peut aider un développeur à maintenir et maintenir le comportement de la collection sous

contrôle. Lorsque nous utilisons une collection sans spécifier les types d'élément autorisés, c'est notre responsabilité d'insérer correctement les éléments. Ceci, dans un contexte plus large, peut devenir cher, car nous devons implémenter des validations pour éviter les mauvaises insertions et le documenter pour une équipe.

Considérez l'exemple de code suivant; comme nous avons nommé la variable `avengerNames`, nous attendez-vous à ce que ce soit une liste de noms et rien d'autre. Malheureusement, sous forme codée, nous pouvons insérer également un numéro dans la liste, provoquant une désorganisation ou une confusion:

```
principale() {
    Liste avengerNames = ["Hulk", "Captain America"];
    avengerNames.add (1);
    print ("Noms des vengeurs: $ avengerNames");
    // imprime les noms des vengeurs: [Hulk, Captain America, 1]
}
```

Cependant, si nous spécifions le type de chaîne pour la liste, ce code ne se compilerait pas, éviter cette confusion:

```
principale() {
    Liste <String> avengerNames = ["Hulk", "Captain America"];
    avengerNames.add (1);
    // Maintenant, la fonction add () attend un 'int' donc cela ne compile pas
    print ("Noms des vengeurs: $ avengerNames");
}
```

[34]

Piste 53

Une introduction à Dart

Chapitre 1

Génériques et littéraux Dart

Si vous consultez la liste de ce chapitre et les exemples de cartes, vous verrez que nous avons utilisé les littéraux `[]` et `{}` pour les initialiser. Avec les génériques, nous pouvons spécifier un type pendant la initialisation, ajout d'un préfixe `<elementType> []` pour listes et `<keyType, elementType> {}` pour les cartes.

Jetez un œil à l'exemple suivant:

```
principale() {
    var avengerNames = <String> ["Hulk", "Captain America"];
    var avengerQuotes = <Chaine, Chaine> {
        "Captain America": "Je peux faire ça toute la journée!",
        "Spider Man": "Suis-je un vengeur?",
        "Hulk": "Smaaaaaash!"
    };
}
```

Spécifier le type de liste, dans ce cas, semble être redondant car l'analyseur Dart en déduit le type de chaîne des littéraux que nous avons fournis. Cependant, dans certains cas, c'est important, comme lorsque nous initialisons une collection vide, comme dans ce qui suit exemple:

```
var emptyStringArray = <String> [];
```

Si nous n'avons pas spécifié le type de la collection vide, elle peut contenir n'importe quel type de données car il n'inférera pas le type générique à adopter.

Pour apprendre comment Dart joue avec le concept générique et les données supplémentaires structures fournies par la [langue](#), vous pouvez vous référer à la visite des langues officielles pour Détails: <https://./fléch>

Introduction à la POO dans Dart

Dans Dart, tout est un objet, y compris les types intégrés. Lors de la définition d'une nouvelle classe, même si vous n'étendez rien, ce sera un *descendant d'un objet*. Dart

fait implicitement cela pour vous.

[35]

Piste 54

Une introduction à Dart

Chapitre 1

Dart est appelé un **véritable langage orienté objet**. Même les fonctions sont des objets, ce qui signifie que vous pouvez faire ce qui suit:

- Attribuez une fonction en tant que valeur d'une variable.
- Passez-le comme argument à une autre fonction.
- Renvoyez-le comme résultat d'une fonction comme vous le feriez avec tout autre type, comme String et int.

Ceci est connu comme ayant **des fonctions de première classe** car elles sont traitées de la même manière que les autres les types.

Un autre point important à noter est que Dart prend en charge l'*héritage unique* sur une classe, similaire à Java et la plupart des autres langages, ce qui signifie qu'une classe peut hériter directement de une seule classe à la fois.

Une classe peut implémenter plusieurs interfaces et étendre plusieurs classes utilisant des mixins, que nous aborderons plus loin dans ce chapitre.

Voici les principaux artefacts POO qui sont présentés dans le langage Dart (nous allons explorer plus profondément dans chacun tout au long de ce chapitre):

- Classe** : il s'agit d'un plan pour créer un objet.
- Interface** : il s'agit d'une définition de contrat avec un ensemble de méthodes disponibles sur un objet. Bien qu'il n'y ait pas de type d'interface explicite dans Dart, nous pouvons atteindre le but de l'interface avec des classes abstraites.
- Classe énumérée** : il s'agit d'un type spécial de classe qui définit un ensemble de valeurs constantes.
- Mixin** : c'est une façon de réutiliser le code d'une classe dans plusieurs hiérarchies de classes.

Fonctionnalités de Dart OOP

Chaque langage de programmation peut fournir le paradigme POO à sa manière, avec des ou un soutien complet, en appliquant tout ou partie des principes suivants:

[36]

Piste 55

Une introduction à Dart

Chapitre 1

Dart applique de nombreux principes avec de nombreuses particularités. Alors, renforçons le techniques et structures de POO disponibles pour utiliser ce paradigme dans le langage Dart.

Les sujets indiqués ici peuvent vous sembler nouveaux. Ils sont couverts en plus en profondeur dans les sections suivantes de ce chapitre. N'hésitez pas à revoir ceci section plus tard si vous le trouvez utile.

Objets et classes

Le point de départ de la POO, les objets, sont des instances de classes définies. Dans Dart, comme déjà été souligné, tout est un objet, c'est-à-dire que chaque valeur que nous pouvons stocker dans une variable est un instance d'une classe. En outre, tous les objets étendent également la classe Object, directement ou indirectement:

Les classes Dart peuvent avoir à la fois des membres d'instance (méthodes et champs) et une classe membres (méthodes et champs statiques).

Les classes Dart ne prennent pas en charge la surcharge du constructeur, mais vous pouvez utiliser le flexible spécifications des arguments de fonction à partir du langage (facultatif, positionnel et named) pour fournir différentes manières d'instancier une classe. Aussi, vous pouvez avoir constructeurs nommés pour définir des alternatives.

[37]

Piste 56

Une introduction à Dart

Chapitre 1

Encapsulation

Dart ne contient pas explicitement de restrictions d'accès, comme les célèbres mots-clés utilisés dans Java: protégé, privé et public. Dans Dart, l'encapsulation se produit à la bibliothèque niveau plutôt qu'au niveau de la classe (ceci sera discuté plus en détail dans le chapitre suivant). Ce qui suit s'applique également:

Dart crée des getters et des setters implicites pour tous les champs d'une classe, afin que vous puissiez définir comment les données sont accessibles aux consommateurs et comment elles évoluent.

Dans Dart, si un identificateur (classe, membre de classe, fonction de niveau supérieur ou variable) démarre avec un trait de soulignement (_), il est privé de sa bibliothèque.

Nous allons vérifier la définition des *bibliothèques* dans [Chapitre 2, Fléchette intermédiaire Programmation](#). Ici, nous aborderons également plus en détail le fonctionnement de la confidentialité dans Dart.

Héritage et composition

L'héritage nous permet d'étendre un objet à des versions spécialisées d'un type abstrait.. Dans Dart, en déclarant simplement une classe, nous étendons déjà implicitement le type Object. le ce qui suit s'applique également:

Dart permet l'héritage direct unique.

Dart a un support spécial pour les mixins, qui peut être utilisé pour étendre la classe fonctionnalités sans héritage direct, simulation d'héritages multiples, et réutiliser le code.

Dart ne contient pas de directive de classe finale comme les autres langages, c'est-à-dire un la classe peut toujours être étendue (avoir des enfants).

Abstraction

Après l'héritage, l' **abstraction** est le processus par lequel nous définissons un type et son essentiel caractéristiques, passant aux types spécialisés des parents. Ce qui suit s'applique également:

Dart contient des classes abstraites qui permettent une définition de *ce que* quelque chose fait / fournit, sans se soucier de la *façon dont* cela est mis en œuvre.

Dart a le puissant concept d'**interface implicite**, qui rend également chaque classe une interface, lui permettant d'être implémentée par d'autres sans l'étendre.

[38]

Psaumes 57

Une introduction à Dart

Chapitre 1

Polymorphisme

Le **polymorphisme** est obtenu par héritage et peut être considéré comme la capacité d'un objet à se comporter comme un autre; par exemple, le type int est également un type num. Le suivant s'applique également:

Dart permet de remplacer les méthodes parentes pour modifier leur comportement d'origine.

Dart n'autorise pas la **surcharge** comme vous le savez peut-être. Tu ne peut pas définir la même méthode deux fois avec des arguments différents. Vous pouvez simuler surcharge en utilisant des définitions d'arguments flexibles (c'est-à-dire facultatives et positionnel, comme vu dans la section *Fonctions précédente*) ou ne pas l'utiliser du tout.

Sommaire

Nous avons terminé notre introduction au langage Dart, et j'espère que vous avez aimé ce que vous ont lu jusqu'à présent. Dans ce premier chapitre, nous avons présenté les outils disponibles pour démarrer votre Dart études de langues, découvert à quoi ressemble un programme de base Dart et structure de base du code Dart.

Nous avons démontré le fonctionnement du SDK Dart et les outils qu'il fournit pour aider Flutter développement d'applications et faire en sorte que le framework Flutter atteigne ses objectifs.

Nous avons passé en revue certains concepts importants du langage Dart avec des liens utiles vers le guides linguistiques officiels pour soutenir le développeur. De plus, nous avons examiné les fonctions et spécifications de paramètres, telles que nommé / positionnel et facultatif / requis, et introduites Dart OOP.

Dans le chapitre suivant, nous avancerons vers le concept de programmation POO dans le Dart langue et ses particularités. Nous examinerons également plusieurs Dart avancés importants fonctionnalités de développement, en particulier lorsque l'on parle de développement Flutter, comme comme programmation asynchrone avec Futures, tests unitaires et concept de packages & bibliothèques, qui est peut-être le plus important pour servir de base au développement de l'application Flutter.

Alors, consultez le chapitre suivant pour des sujets plus avancés sur Dart.

[39]

Psaumes 58

Une introduction à Dart

Chapitre 1

Lectures complémentaires

En plus du contenu de ce chapitre, vous pouvez consulter les documents suivants pour plus de référence:

Pour plus d'informations sur les représentations de nombres entiers dans Dart, vous pouvez lire l'article suivant, qui peut vous aider à comprendre comment la langue traite les numéros en interne: <https://.. / www.daii.calcul/numérique>. Vous pouvez en savoir plus sur la syntaxe des génériques ici: <https://.. / GitHub/sdk/bl>

[40]

Piste 59

Programmation de fléchettes intermédiaire



Dans ce chapitre, vous apprendrez le concept de base des objets dans Dart, par exemple, comment créer code orienté objet dans Dart en utilisant ses concepts, tels que les interfaces, les interfaces implicites et des classes abstraites, ainsi que des mixins, pour ajouter un comportement à une classe.

Si vous êtes un programmeur expérimenté ou déjà familiarisé avec Java ou des langages similaires, vous pouvez sauter certaines parties de ce chapitre, car il présente de nombreuses similitudes avec la POO typique concepts, tels que l'héritage et l'encapsulation. Certaines idées, en particulier, sont importantes pour vérifier, même si vous connaissez déjà la majorité des fonctionnalités de POO, comme implicite interfaces et mixins, car ils peuvent vous présenter de nouveaux concepts.

Vous apprendrez également à utiliser des bibliothèques tierces pour accélérer le développement d'un projet, acquérir une compréhension des fonctionnalités avancées du langage Dart pour commencer à développer des applications multithreading en utilisant des rappels et des futurs, et apprenez à tester les unités de votre Dart code.

Ce chapitre couvre les sujets suivants:

- Syntaxe de la définition de classe Dart
- Classes abstraites, interfaces et mixins
- Comprendre les bibliothèques et les packages Dart
- Ajouter des dépendances avec pubspec.yaml
- Présentation de la programmation asynchrone avec Futures et Isolates
- Présentation des tests unitaires

Piste 60

Programmation de fléchettes intermédiaire

Chapitre 2

Classes et constructeurs de fléchettes

Les classes Dart sont déclarées à l'aide du mot-clé class, suivi du nom de la classe, ancêtre classes et interfaces implémentées. Ensuite, le corps de la classe est entouré par une paire de accolades, dans lesquelles vous pouvez ajouter des membres de classe, qui incluent les éléments suivants:

- Champs** : ce sont des variables utilisées pour définir les données qu'un objet peut contenir.
- Accesseurs** : les getters et les setters, comme son nom l'indique, sont utilisés pour accéder aux champs d'une classe, où get est utilisé pour récupérer une valeur, et l'accesseur set est utilisé pour modifier la valeur correspondante.
- Constructeur** : il s'agit de la méthode de création d'une classe où les champs d'instance d'objet sont initialisés.
- Méthodes** : Le comportement d'un objet est défini par les actions qu'il peut entreprendre. Celles-ci sont les fonctions de l'objet.

Reportez-vous à l'exemple de définition de petite classe suivant:

```
Classe Personne {
    String firstName;
    String lastName;

    String getFullName () => "$ firstName $ lastName";
}

principale() {
    Person somePerson = new Person();
```

```

somePerson.firstName = "Clark";
somePerson.lastName = "Kent";
print(somePerson.getFullName()); // imprime Clark Kent
}

```

Maintenant, jetons un œil à la classe Person déclarée dans le code précédent et faisons quelques observations:

Pour instancier une classe, nous utilisons le mot-clé new (*facultatif*) suivi du appel du constructeur. Au fur et à mesure que nous avançons dans ce livre, vous remarquerez que Le mot-clé est moins utilisé.

Il n'a pas de classe ancêtre explicitement déclarée, mais il en a une, le type d'objet, comme déjà mentionné, et cet héritage se produit implicitement dans Dart.

Il a deux champs, firstName et lastName, et une méthode, getFullName(), qui concatène les deux en utilisant une interpolation de chaîne, puis retourne.

[42]

Piste 61

Programmation de fléchettes intermédiaire

Chapitre 2

Aucun accesseur get ou set n'est déclaré, alors comment avons-nous accéder à firstName et lastName pour le faire muter? Un accesseur get / set par défaut est défini pour chaque champ d'une classe.

La notation *dot class.member* est utilisée pour accéder à un membre de classe, quel qu'il soit — une méthode ou un champ (get / set).

Nous n'avons pas défini de constructeur pour la classe, mais, comme vous le pensez peut-être, il y a un constructeur vide par défaut (sans arguments) déjà fourni pour nous.

Le type enum

Le type enum est un type courant utilisé par la plupart des langages pour représenter un ensemble fini valeurs constantes. Dans Dart, ce n'est pas différent. En utilisant le mot-clé enum, suivi de valeurs constantes, vous pouvez définir un type enum:

```

enum PersonType {
    étudiant, employé
}

```

Notez que vous ne définissez que les noms de valeur. les types enum sont des types spéciaux avec un ensemble fini valeurs qui ont une propriété d'index représentant sa valeur. Voyons maintenant comment cela fonctionne.

Tout d'abord, nous ajoutons un champ à notre classe Person précédemment définie pour stocker son type:

```

Classe Personne {
    ...
    Type PersonType;
    ...
}

```

Ensuite, nous pouvons l'utiliser comme n'importe quel autre champ:

```

principale() {
    print(PersonType.values); // imprime [PersonType.student,
                            // PersonType.employee]
    Person somePerson = new Person();
    somePerson.type = PersonType.employee;
    print(somePerson.type); // imprime PersonType.employee
    print(somePerson.type.index); // imprime 1
}

```

Vous pouvez voir que la propriété index est égale à zéro, en fonction de la position de déclaration de la valeur.

[43]

Piste 62*Programmation de fléchettes intermédiaire**Chapitre 2*

En outre, vous pouvez voir que nous appelons directement le getter de valeurs sur l'énumération `PersonType`. Il s'agit d'un membre statique du type enum qui renvoie simplement une liste avec toutes ses valeurs. nous examinerons cela plus en détail bientôt.

La notation en cascade

Nous avons vu que Dart fournit la notation par points pour accéder à un membre de classe. En plus de ça, nous pouvons également utiliser la notation double point / cascade, **sûre syntaxique**, qui nous permet d'enchaîner une séquence d'opérations sur le même objet:

```
principale() {
    Person somePerson = new Person()
        ..firstName = "Clark"
        ..lastName = "Kent";

    print(somePerson.getFullName()); // imprime Clark Kent
}
```

Le résultat est le même que lors de l'utilisation de l'approche typique. C'est juste une bonne façon d'écrire du code succinct et lisible.

La syntaxe en cascade fonctionne en obtenant la première valeur de retour d'expression (nouveau `Person()`, dans ce cas) et opère toujours dans cette valeur, en ignorant la prochaine expression qui renvoie des valeurs.

Ensuite, nous allons approfondir chacun des composants de classe mentionnés auparavant pour comprendre comment ils peuvent être utilisés pour étendre une classe à tous nos besoins.

Constructeurs

Pour instancier une classe, nous utilisons le nouveau mot-clé, suivi du constructeur correspondant avec des paramètres, si nécessaire. Maintenant, changeons la classe `Person` et définissons un constructeur avec des paramètres dessus:

```
Classe Personne {
    String firstName;
    String lastName;

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

[44]

Piste 63*Programmation de fléchettes intermédiaire**Chapitre 2*

```
String getFullName() => "$firstName $lastName";
}

principale() {
    // Person somePerson = new Person(); cela ne compilerait pas car nous
    // définissons des paramètres obligatoires sur le constructeur
    Person somePerson = new Person("Clark", "Kent");
    print(somePerson.getFullName());
}
```

Le constructeur est aussi une fonction dans Dart et son rôle est d'initialiser l'instance de la classe correctement. En tant que fonction, il peut avoir de nombreuses caractéristiques d'une fonction Dart commune,

tels que les arguments - obligatoires ou facultatifs et nommés ou positionnels. Dans le précédent exemple, le constructeur a deux arguments obligatoires.

Si vous regardez dans notre corps de constructeur, il utilise le mot-clé `this`. De plus, le constructeur les noms de paramètres sont les mêmes que ceux des champs, ce qui peut provoquer une ambiguïté. Donc, pour éviter ceci, nous préfixons les champs d'instance d'objet avec le mot clé `this` lors de l'attribution de valeur étape.

Dart fournit une autre façon d'écrire un constructeur comme celui fourni dans l'exemple, par en utilisant une syntaxe de raccourci:

```
// ... définition des champs de classe
// syntaxe d'initialisation du raccourci
Personne (this.firstName, this.lastName);
```

Nous pouvons omettre le corps du constructeur car il ne définit que les valeurs du champ de classe sans aucune configuration supplémentaire qui lui est appliquée.

Constructeurs nommés

Contrairement à Java et à de nombreux autres langages, Dart n'a pas de surcharge par redéfinition, donc, pour définir des constructeurs alternatifs pour une classe, vous devez utiliser les constructeurs nommés:

```
// ... définition des champs de classe
// autres constructeurs
Person.anonymous () {}
```

[45]

Piste 64

Programmation de fléchettes intermédiaire

Chapitre 2

Un constructeur nommé permet de définir des constructeurs alternatifs pour une classe. Dans le précédent Par exemple, nous avons défini un constructeur alternatif pour une classe Person sans nom.

La seule différence par rapport à une méthode simple est que les constructeurs n'ont pas de déclaration de retour, car la seule chose à faire est de initialisez correctement l'instance d'objet.

Nous verrons les constructeurs nommés en action dans les chapitres sur Flutter, comme le framework les utilise beaucoup pour initialiser les définitions de widgets.

Constructeurs d'usine

Une autre syntaxe utile dans Dart est le constructeur de fabrique, qui permet d'appliquer la fabrique pattern, une technique de création qui permet aux classes d'être instanciées sans spécifier le type d'objet résultant exact. Supposons que nous ayons les descendants suivants de la classe Person:

```
classe L'élève étend la personne {
    Étudiant (prénom, nom): super (prénom, nom);
}

L'employé de classe Person {
    Employé (prénom, nom): super (prénom, nom);
}
```

Comme vous pouvez le constater, les classes descendantes sont toujours presque les mêmes que la classe Person, car ils n'ajoutent pas encore de fonctionnalités spécifiques.

Nous pouvons définir un constructeur d'usine sur la classe Person pour instancier la classe basée sur l'argument de type requis:

```
Classe Personne {
    String firstName;
```

```

String lastName;

Personne ([this.firstName, this.lastName]);

Factory Person.fromType ([type PersonType]) {
    commutateur (type) {
        case PersonType.employee:
            retourner un nouvel employé ();
        case PersonType.student:
            retourne un nouvel étudiant ();
    }
    retour Personne ();
}

```

[46]

Piste 65*Programmation de fléchettes intermédiaire**Chapitre 2*

```

String getFullName () => "$ firstName $ lastName";
}

enum PersonType {étudiant, employé}

```

Le constructeur de fabrique est spécifié en ajoutant le mot-clé `factory`, suivi du définition du constructeur, généralement dans une classe de base ou un type de classe abstraite. Dans notre cas, le La classe `Person` définit un constructeur nommé `en usine` basé sur `PersonType` spécifié dans le argument. Si aucun type n'est passé, il crée une classe `Person` simple en utilisant sa valeur par défaut constructeur.

Une autre chose importante à noter est que le constructeur d'usine *ne remplace pas* la valeur par défaut constructeur de classe. Par conséquent, lui et ses descendants peuvent toujours être instanciés directement par le votre interlocuteur.

Accesseurs de terrain - getters et setters

Comme mentionné précédemment, les getters et les setters nous permettent d'accéder à un champ sur une classe, et chaque field a ces accesseurs, même si nous ne les définissons pas. Dans la personne précédente exemple, lorsque nous exécutons `somePerson.firstName = "Peter"`, nous appelons le L'accesseur set du champ `firstName` et l'envoi de "Peter" comme paramètre à celui-ci. Aussi dans le exemple, l'accesseur `get` est utilisé lorsque nous appelons la méthode `getFullName ()` sur la personne, et il concatène les deux noms.

Nous pouvons modifier notre classe `Person` pour remplacer l'ancienne méthode `getFullName ()` et l'ajouter en tant que getter, comme illustré dans le bloc de code suivant, par exemple:

```

Classe Personne {
    String firstName;
    String lastName;

    Personne (this.firstName, this.lastName);

    Person.anonymous () {}

    String get fullName => "$ firstName $ lastName";
    String get initials => "$ {firstName [0]} . ${LastName [0]} .";
}

principale() {
    Person somePerson = new Person ("clark", "kent");

    print (somePerson.fullName); // imprime Clark Kent
    print (somePerson.initials); // imprime ck
}

```

[47]

Piste 66

```
somePerson.fullName = "peter parker";
// nous n'avons pas défini de setter fullName donc il ne compile pas
}
```

Les observations importantes suivantes peuvent être faites à propos de l'exemple précédent:

Nous n'aurions pas pu définir un getter ou un setter avec les mêmes noms de champ: prénom et nom. Cela nous donnerait une erreur de compilation, car la classe les noms de membres ne peuvent pas être répétés.

Le getter des initiales lèverait une erreur pour une personne instanciée par le constructeur nommé anonyme, car il n'aurait pas firstName et valeurs lastName (équivaut à null).

Nous n'avons pas besoin de toujours définir la paire, obtenir et définir, ensemble, comme vous pouvez le voir que nous n'avons défini qu'un getter fullName et non un setter, donc nous ne pouvons pas modifier le nom complet. (Cela entraîne une erreur de compilation, comme indiqué précédemment.)

Nous aurions pu également écrire un setter pour fullName et définir la logique derrière celui-ci pour définir firstName et lastName basés sur cela:

```
Classe Personne {
    // ... définition des champs de classe
    set fullName (String fullName) {
        var parts = fullName.split ("");
        this.firstName = parts.first;
        this.lastName = parts.last;
    }
}
```

De cette façon, quelqu'un pourrait initialiser le nom d'une personne en définissant fullName et le résultat serait pareil. (Bien entendu, nous n'avons effectué aucun contrôle pour déterminer si le La valeur transmise en tant que fullName est valide, c'est-à-dire non vide, avec deux valeurs ou plus, et ainsi de suite.)

Champs et méthodes statiques

Comme vous le savez déjà, les champs ne sont rien de plus que des variables contenant des valeurs d'objet, et Les méthodes sont des fonctions simples qui représentent des actions d'objets. Dans certains cas, vous voudrez peut-être partager une valeur ou une méthode entre toutes les instances d'objet d'une classe. Pour ce cas d'utilisation, vous peut leur ajouter le modificateur statique, comme suit:

```
Classe Personne {
    // ... définition des champs de classe

    static String personLabel = "Nom de la personne:";
```

[48]

Piste 67

```
String get fullName => "$ personLabel $ firstName $ lastName";
// modifié pour imprimer le nouveau champ statique "personLabel"
}
```

Par conséquent, nous pouvons modifier la valeur du champ statique directement sur la classe:

```
principale() {
    Person somePerson = Person ("clark", "kent");
    Person anotherPerson = Person ("peter", "parker");

    print (somePerson.fullName); // imprime le nom de la personne: clark kent
    print (anotherPerson.fullName); // imprime le nom de la personne: peter park

    Person.personLabel = "nom:";

    print (somePerson.fullName); // affiche le nom: clark kent
    print (anotherPerson.fullName); // affiche le nom: peter park
}
```

Les champs statiques sont associés à la classe, plutôt qu'à toute instance d'objet. La même chose pour les définitions de méthode statique. Nous pouvons ajouter une méthode statique pour encapsuler le nom impression, comme illustré dans le bloc de code suivant, par exemple:

```
Classe Personne {
    // ... définition des champs de classe
    static String personLabel = "Nom de la personne:";

    static void printsPerson (Personne personne) {
        print ("$ personLabel $ {person.firstName} $ {person.lastName}");
    }
}
```

Ensuite, nous pouvons utiliser cette méthode pour imprimer une instance de Person, comme nous l'avons fait auparavant:

```
principale() {
    Person somePerson = Person ("clark", "kent");
    Person anotherPerson = Person ("peter", "parker");

    Person.personLabel = "nom:";

    Person.printsPerson (somePerson); // affiche le nom: clark kent
    Person.printsPerson (anotherPerson); // affiche le nom: peter park
}
```

[49]

Piste 68

Programmation de fléchettes intermédiaire

Chapitre 2

Nous pourrions modifier le getter fullName sur la classe Person pour ne pas utiliser le personLabel champ statique, pour avoir plus de sens et obtenir des résultats distincts selon notre exigences:

```
Classe Personne {
    // ... définition des champs de classe

    String get fullName => "$ firstName $ lastName";

}

principale() {
    Person somePerson = Person ("clark", "kent");
    Person anotherPerson = Person ("peter", "parker");

    print (somePerson.fullName); // imprime Clark Kent
    print (anotherPerson.fullName); // imprime Peter Parker

    Person.printsPerson (somePerson); // imprime le nom de la personne: clark kent
    Person.printsPerson (anotherPerson); // imprime le nom de la personne: peter park
}
```

Comme vous pouvez le voir, les champs et méthodes statiques nous permettent d'ajouter des comportements spécifiques aux classes dans général.

Héritage de classe

En plus de l'héritage implicite du type Object, Dart nous permet d'étendre défini classes en utilisant le mot-clé extend, où tous les membres de la classe parent sont hérité, à l'exception des constructeurs.

Maintenant, regardons l'exemple suivant, où nous créons une classe enfant pour l'existant

Classe de personne:

```
classe L'élève étend la personne {
    String nickName;

    Student (String firstName, String lastName, this.nickName)
        : super (prénom, nom);
```

```

@passer outre
String toString () => "$ fullName, également connu sous le nom de $ nickName";
}

principale() {
    Élève étudiant = nouvel étudiant ("Clark", "Kent", "Kal-El");
}

```

[50]

Piste 69*Programmation de fléchettes intermédiaire**Chapitre 2*

```

imprimer (étudiant); // identique à l'appel de student.toString ()
// imprime Clark Kent, également connu sous le nom de Kal-El
}

```

Les observations suivantes peuvent être faites à propos de l'exemple précédent:

Student: la classe Student définit son propre constructeur. Cependant, il appelle le Constructeur de classe Person, en passant les paramètres requis. Ceci est fait avec le super mot-clé.

@override: Il existe une méthode `toString ()` remplacée sur la classe Student. C'est là que l'héritage prend tout son sens: nous modifions le comportement d'une classe parente (Object, dans ce cas) sur la classe enfant.

`print (étudiant):` comme vous pouvez le voir dans la déclaration `print (étudiant)`, nous ne sommes pas appeler n'importe quelle méthode; la méthode `toString ()` est appelée pour nous implicitement.

La méthode `toString ()`

La méthode `toString ()` est un excellent exemple courant de remplacement du comportement des parents. Le l'objectif de cette méthode est de renvoyer une représentation String de l'objet:

```

classe L'élève étend la personne {
    // ... fullName (de la classe Person) et autres champs
    @passer outre
    String toString () => "$ fullName, également connu sous le nom de $ nickName";
}

principale() {
    Élève étudiant = nouvel étudiant ("Clark", "Kent", "Kal-El");

    print ("Ceci est un étudiant: $ étudiant");
    // imprime: Il s'agit d'un étudiant: Clark Kent, également connu sous le nom de Kal-El
    // appellera également le toString () de student implicitement
}

```

Comme vous pouvez le voir, cela rend le code plus propre et nous fournissons une bonne représentation textuelle de l'objet qui peut aider à comprendre les journaux, la mise en forme du texte, etc.

Interfaces, classes abstraites et mixins

Dans Dart, les classes abstraites et les interfaces sont étroitement liées les unes aux autres. Ceci est dû au fait Dart implémente les interfaces d'une manière subtilement différente de la plupart des langages classiques.

[51]

Piste 70*Programmation de fléchettes intermédiaire**Chapitre 2*

Jetons un coup d'œil aux classes abstraites avant de les lier au sujet de l'implicite les interfaces.

Classes abstraites

En POO, les classes abstraites sont des classes qui ne peuvent pas être instanciées, ce qui a beaucoup de sens, selon le contexte et le niveau d'abstraction d'un programme.

Par exemple, notre classe Person pourrait être abstraite si nous voulons nous assurer qu'elle n'existe que dans le contexte du programme s'il s'agit d'une instance Student ou d'un autre sous-type:

```
classe abstraite Personne {
    // ... le corps a été caché par souci de concision
}
```

La seule chose que nous devons changer ici est le début de la définition de classe, en la marquant comme abstrait:

```
principale() {
    Personne étudiant = nouvel étudiant ("Clark", "Kent", "Kal-El"); // travailler en tant que
    // nous instancions le sous-type
    // Personne p = nouvelle personne ();
    // les classes abstraites ne peuvent pas être instanciées

    imprimer (étudiant);
}
```

Comme vous pouvez le voir, nous ne pouvons plus instancier une classe Person, juste son sous-type, Student.

Une classe abstraite peut avoir des membres abstraits sans implémentation, ce qui lui permet d'être implémenté par les types enfants qui les étendent:

```
classe abstraite Personne {
    String firstName;
    String lastName;

    Personne (this.firstName, this.lastName);

    String get fullName;
}
```

[52]

Piste 71

Programmation de fléchettes intermédiaire

Chapitre 2

Le getter fullName de la classe Person précédente est maintenant abstrait, car il n'a pas une mise en œuvre. Il est de la responsabilité de l'enfant de mettre en œuvre ce membre:

```
classe L'élève étend la personne {
    // ... autres membres de la classe

    @passer outre
    String get fullName => "$ firstName $ lastName";
}
```

La classe Student implémente le getter fullName car, si ce n'était pas le cas, nous ne serions pas capable de compiler le code.

Interfaces

Dart n'a pas le mot-clé interface mais nous permet d'utiliser des interfaces de manière subtile manière différente de ce à quoi vous êtes habitué. Toutes les déclarations de classe sont elles-mêmes les interfaces. Cela signifie que, lorsque vous définissez une classe dans Dart, vous définissez également un interface qui peut être *implémentée* et pas seulement *étendue* par d'autres classes. C'est appelé **interfaces implicites** dans le monde de Dart.

Sur cette base, notre précédente classe Person est également une interface Person qui pourrait être

implémenté, au lieu d'être étendu, par la classe Student:

```
class Student implements Person {
    String nickName;

    @passer outre
    String firstName;

    @passer outre
    String lastName;

    Étudiant (this.firstName, this.lastName, this.nickName);

    @passer outre
    String get fullName => "$ firstName $ lastName";

    @passer outre
    String toString () => "$ fullName, également connu sous le nom de $ nickName";
}
```

[53]

Psaumes 72

Programmation de fléchettes intermédiaire

Chapitre 2

Notez que, en général, le code ne change pas trop, sauf dans la mesure où les membres sont maintenant définis dans la classe Student. La classe Person n'est qu'un *contrat* que l'étudiant classe adoptée et doit mettre en œuvre.

Si vous souhaitez déclarer une interface explicite, il vous suffit de créer un classe abstraite sans aucune implémentation dessus, juste des définitions de membres, et ce sera une interface pure, prête à être implémentée.

Mixins - ajouter un comportement à une classe

En POO, les mixins sont un moyen d'inclure des fonctionnalités sur une classe sans avoir besoin de associations entre les parties, telles que l'héritage.

Les contextes les plus courants où les mixins peuvent être utilisés sont dans les endroits où plusieurs des héritages peuvent être nécessaires, car c'est un moyen facile pour les classes d'utiliser des fonctionnalités communes.

Dans Dart, il existe plusieurs façons de déclarer un mixin:

- En déclarant une *classe* et en l'utilisant comme mixin, ce qui lui permet également d'être utilisé comme un objet
- En déclarant une *classe abstraite*, en lui permettant d'être utilisée comme mixin ou d'être héritée, mais pas instancié
- En le déclarant comme *mixin*, lui permettant d'être utilisé uniquement comme mixin

Peu importe comment vous déclarez un mixin, il peut également être utilisé comme interface, car il expose les membres, et c'est la prémissse derrière tout cela.

Voyons maintenant un exemple de déclaration d'une fonctionnalité que notre classe Person précédente Pourrais avoir.

Par exemple, pensons aux professions qu'une personne pourrait avoir. Certaines personnes peuvent avoir des compétences spécifiques et communes, et les mixins peuvent être idéaux pour ce cas d'utilisation car nous pouvons ajouter les compétences à une profession sans avoir besoin de les faire étendre une classe générique ou implémentez une interface dans chacun d'eux. Comme la mise en œuvre le ferait probablement la même chose, cela provoquerait des duplications de code:

```
// Définition de la classe Person
```

```
class ProgrammingSkills {
```

[54]

Piste 73

Programmation de fléchettes intermédiaire

Chapitre 2

```
codage () {
    print ("code d'écriture ...");
}
}

class ManagementSkills {
    gérer() {
        print ("gestion du projet ...");
    }
}
```

Dans l'exemple précédent, nous avons créé deux classes de compétences professionnelles, ProgrammingSkills et ManagementSkills. Maintenant, nous pouvons les utiliser en ajoutant le mot-clé with à la classe définition, par exemple:

```
La classe SeniorDeveloper étend la personne avec des compétences en programmation,
Compétences de gestion {
    SeniorDeveloper (String firstName, String lastName): super (firstName,
    nom de famille);
}
```

```
La classe JuniorDeveloper étend Person avec ProgrammingSkills {
    JuniorDeveloper (String firstName, String lastName): super (firstName,
    nom de famille);
}
```

Les deux classes auront la méthode coding () sans avoir besoin de l'implémenter dans chaque classe, car il est déjà implémenté dans le mixin ProgrammingSkills.

Comme mentionné précédemment, il existe plusieurs façons de déclarer un mixin. Dans le précédent exemple, nous avons utilisé une définition de classe simple. De cette façon, la classe ProgrammingSkills peut être étendue comme une classe normale ou même implémenté en tant qu'interface (perdre le mixin propriété):

```
La classe AdvancedProgrammingSkills étend ProgrammingSkills {
    faire du café() {
        imprimer ("faire du café ...");
    }
}
```

L'écriture de AdvancedProgrammingSkills de cette manière n'en fait pas un mixin. Les classes Mixin doivent étendre la classe Object et déclarer non constructeur.

[55]

Piste 74

Programmation de fléchettes intermédiaire

Chapitre 2

Une autre façon d'écrire un mixin est d'utiliser le mot-clé mixin:

```
mixin ProgrammingSkills {
    codage () {
        print ("code d'écriture ...");
    }
}
```

```
 mixin ManagementSkills {
    gérer() {
        print ("gestion du projet ...");
    }
}
```

L'écriture de mixins de cette manière empêche les comportements indésirables car les mixins ne peuvent pas être étendu et sont destinés à être utilisés correctement. Les classes de métiers qui utilisent des mixins reste le même.

Une autre chose que nous pouvons faire est de limiter les classes qui peuvent utiliser un certain mixin. Pour ce faire, nous besoin de spécifier la superclasse requise en utilisant le mot-clé on:

```
 mixin ProgrammingSkills on Developer {
    codage () {
        print ("code d'écriture ...");
    }
}
```

Les mixins limités par le mot-clé on exigent que la classe cible ait un no-constructeur d'arguments.

Classes appelables, fonctions de niveau supérieur et variables

Dart est très flexible pour laisser le développeur prendre le contrôle de tous les morceaux de code et, contrairement à de nombreuses langues, il n'y a pas de manière unique de faire quelque chose.

Comme Dart propose de combiner les avantages des concepts POO modernes avec les concepts traditionnels, vous pouvez toujours choisir quand et où appliquer les différentes approches.

[56]

Piste 75

Programmation de fléchettes intermédiaire

Chapitre 2

Classes appelables

De la même manière que les fonctions Dart ne sont rien de plus que des objets, les classes Dart peuvent se comporter comme les fonctions aussi, c'est-à-dire qu'elles peuvent être invoquées, prendre des arguments et renvoyer quelque chose. Par conséquent. La syntaxe pour émuler une fonction dans une classe est la suivante:

```
 class ShouldWriteAProgram { // c'est une classe simple
    Langue des cordes;
    Plateforme String;

    ShouldWriteAProgram (this.language, this.platform);

    // cette méthode spéciale nommée 'call' fait que la classe se comporte comme une fonction
    appel booléen (catégorie String) {
        if (language == "Dart" && platform == "Flutter") {
            return category! = "à faire";
        }
        retourner faux;
    }

    principale() {
        var shouldWrite = ShouldWriteAProgram ("Dart", "Flutter");

        print (shouldWrite ("todo")); // affiche faux.
        // cette fonction appelle la classe appelable ShouldWriteAProgram
        // résultant en un appel implicite à sa méthode "call"
    }
}
```

Comme vous pouvez le voir, la variable shouldWrite est un objet, une instance de

la classe `ShouldWriteAProgram`, mais peut également être appelée comme une fonction normale en passant un paramètre et en utilisant sa valeur de retour. Ceci est possible en raison de l'existence de la méthode `call()` définie dans la classe.

La méthode `call()` est une méthode spéciale de Dart. Chaque classe qui la définit peut se comporter comme un fonction Dart normale.

Si vous affectez une classe appellable à une variable de type de fonction, elle sera implicitement converti en un type de fonction et se comportent comme un normal fonction.

[57]

Psautres 76

Programmation de fléchettes intermédiaire

Chapitre 2

Fonctions et variables de niveau supérieur

Dans ce chapitre, nous avons vu que les fonctions et les variables de Dart peuvent être liées à des classes comme membres: champs de classe et méthodes.

La méthode de haut niveau d'écriture des fonctions est également déjà connue à partir du [chapitre 1](#), *Un Introduction à Dart*, où nous avons écrit la fonction Dart la plus célèbre: le point d'entrée de chaque application, `main()`. Pour les variables, la manière de déclarer est la même. Nous le laissons juste hors de toute portée de fonction, afin qu'il soit accessible globalement sur l'application / le package:

```
var globalNumber = 100;
globalFinalNumber final = 1000;

void printHello () {
    print ("Dart de la portée globale.
    Il s'agit d'un nombre de premier niveau: \$ globalNumber
    Il s'agit d'un nombre final de premier niveau: \$ globalFinalNumber
    ");
}

principale() {
    // la fonction de niveau supérieur Dart la plus célèbre
    printHello (); // imprime la valeur par défaut

    globalNumber = 0;
    // globalFinalNumber = 0; // ne compile pas car il s'agit d'une variable finale

    printHello (); // imprime la nouvelle valeur
}
```

Comme vous pouvez le voir, les variables et les fonctions n'ont pas besoin d'être liées à une classe pour exister. C'est la flexibilité proposée par le langage Dart, apportant au développeur la possibilité d'écrire code simple et cohérent, sans oublier les modèles et les fonctionnalités de la langues.

Comprendre les bibliothèques et les packages Dart

Les bibliothèques sont un moyen de structurer un projet basé sur la modularité, ce qui permet au développeur pour diviser le code sur plusieurs fichiers et pour partager un *morceau de code* ou un *module* avec d'autres développeurs.

[58]

Piste 77*Programmation de fléchettes intermédiaire**Chapitre 2*

De nombreux langages de programmation utilisent des bibliothèques pour fournir cette modularité au développeur, et Dart n'est pas différent. Dans Dart, ces bibliothèques ont également un autre rôle important en plus structuration du code. Autrement dit, ils déterminent ce qui est visible ou non pour les autres bibliothèques.

Avant d'entrer dans le package Dart, nous devons comprendre la plus petite unité que le bibliothèque se compose de. Tout d'abord, explorons comment utiliser une bibliothèque dans notre package et, en suivant ceci, apprenez à définir une bibliothèque dans Dart.

Importer et utiliser une bibliothèque

Dans [Chapitre 1](#), *Une introduction à Dart*, dans la section *Fonctions*, nous avons importé la métabibliothèque pour utiliser l'annotation `@required` sur certains paramètres. Maintenant, explorons la déclaration d'importation plus en détail.

Pour définir une bibliothèque, nous créons simplement un fichier Dart contenant du code.

Jetez un œil à l'exemple `example_1_importing` pour plus de clarté visualisation des librairies et instructions d'import. Vous pouvez trouver le code source de ce chapitre sur GitHub.

Dans cet exemple, nous avons défini une bibliothèque simple avec la personne, l'étudiant, et les classes `Employee` avec l'énumération `PersonType`:

```
// bibliothèque person_lib - le contenu des classes a été tronqué par souci de concision

Classe Personne {
    String firstName;
    String lastName;
    Type PersonType;

    Personne ([this.firstName, this.lastName]);

    String get fullName => "$ firstName $ lastName";
}

enum PersonType {étudiant, employé}

classe L'élève étend la personne {
    Étudiant ([firstName, lastName]): super (firstName, lastName) {
        type = PersonType.student;
    }
}
```

[59]

Piste 78*Programmation de fléchettes intermédiaire**Chapitre 2*

```
L'employé de classe étend Person {
    Employé ([firstName, lastName]): super (firstName, lastName) {
        type = PersonType.employee;
    }
}
```

Pour l'importer, nous pouvons simplement ajouter l'`import library_path;` déclaration au début de le fichier et avant tout code:

```
import 'person_lib.dart';

void main () {
    Personne personne = Personne ("Clark", "Kent");
}
```

```
// a omis le mot-clé optionnel 'new'

Personne étudiant = étudiant ("Clark", "Kent");

print ("Personne: $ {person.fullName}, tapez: $ {person.type}");
print ("Student: $ {student.fullName}, tapez: $ {student.type}");
}
```

Comme les fichiers sont dans le même répertoire, le chemin d'importation est simplement le nom du fichier. Après avoir ajouté l'instruction import, nous pouvons utiliser n'importe quel code disponible à partir de celle-ci - de la même manière que nous l'avons fait avec les classes Personne et Etudiant.

Importer afficher et masquer

Si vous regardez l'exemple précédent, vous remarquerez que nous n'avons pas utilisé tous les classes disponibles dans la bibliothèque person_lib. Pour rendre le code plus propre et moins susceptibles d'erreurs et de conflits de noms, nous pouvons utiliser le mot-clé show, qui nous permet pour importer uniquement les identifiants que nous souhaitons utiliser efficacement dans notre code:

```
// import 'person_lib.dart' show Person, Student;
```

Nous pouvons également spécifier les identifiants que nous ne souhaitons pas importer explicitement en utilisant le mot-clé hide. Dans ce cas, nous importerons tous les identifiants de la bibliothèque sauf ceux après le mot-clé hide:

```
// import 'person_lib.dart' hide Employee;
```

[60]

Piste 79

Programmation de fléchettes intermédiaire

Chapitre 2

Importer des préfixes dans des bibliothèques

Dans Dart, il n'y a pas de définition d'espace de noms ou quelque chose qui identifie de manière unique une bibliothèque dans le contexte dans lequel il est utilisé, donc des conflits peuvent survenir lors de la création de noms d'identifiant, c'est-à-dire Les bibliothèques peuvent définir une fonction de premier niveau ou même une classe avec le même nom. Bien que nous peut utiliser les modificateurs afficher et masquer pour définir explicitement les membres que nous voulons importer d'une bibliothèque, ce n'est pas suffisant pour résoudre le problème car, parfois, nous pouvons être intéressé par une classe ou une fonction de premier niveau avec le même nom dans différentes bibliothèques:

une instruction d'importation pour définir un préfixe pour tous les identificateurs de la bibliothèque importée:

```
importer 'a.dart' comme bibliothèqueA;
importer 'b.dart' en tant que bibliothèqueB;

void main () {

libraryA.Person personA = libraryA.Person ("Clark", "Kent");

print ("Personne A: $ {personA.fullName}");
}
```

[61]

Piste 80

Programmation de fléchettes intermédiaire

Chapitre 2

```
libraryB.Person personB = libraryB.Person (); // La personne 'b' ne
                                              // avoir n'importe quel champ
print ("Personne B: $ {personneB}");
}
```

Comme vous pouvez le voir, sans ce préfixe, nous n'avons pas de moyen d'identifier la classe Person à utilisation. Il en va de même pour tout identifiant de bibliothèque publique, tel qu'une fonction ou une variable. Après en spécifiant le préfixe, nous devons l'ajouter à chaque appel à un membre de cette bibliothèque, pas seulement les contradictoires.

Vous pouvez trouver le code source de ce chapitre sur GitHub.

Si vous vous souvenez du [chapitre 1](#), *Une introduction à Dart*, le mot-clé as est également utilisé pour le typage d'un supertype vers un sous-type.

Importer des variantes de chemin

Dans les exemples précédents, nous avons importé une bibliothèque de fichiers locale qui se trouve dans le même répertoire que le client de la bibliothèque, nous venons donc de spécifier le nom du fichier.

Cependant, ce n'est pas le cas lorsque vous utilisez des packages Dart tiers. Dans ce cas, les fichiers n'existeront pas dans le même répertoire, alors voyons comment nous pouvons importer un bibliothèque Dart du paquet externe.

Il existe plusieurs façons de spécifier les chemins de bibliothèque dans l'instruction d'importation, et nous avons déjà utilisé deux d'entre eux: l'importation de fichiers relatifs et l'importation à partir d'un *package*. Maintenant, prenons un regard sur chacun d'eux plus en détail.

Supposons que nous ayons un répertoire de package d'un petit package foo contenant deux fichiers: a.dart et b.dart. Pour les importer, nous pouvons utiliser plusieurs approches:

Un chemin de fichier relatif : Ceci est similaire à la méthode que nous avons utilisée dans le précédent exemple, car les bibliothèques étaient dans le même dossier. On peut juste mettre le parent chemin vers le fichier de bibliothèque que nous voulons importer, comme suit:

```
import 'foo / a.dart';
import 'foo / b.dart';
```

[62]

Piste 81

Un chemin de fichier absolu : nous pouvons ajouter le chemin absolu sur l'ordinateur à une bibliothèque fichier en ajoutant le préfixe URI file: // au chemin d'importation:

```
import "fichier: // c:/dart_package/foo/a.dart";
import "fichier: // c:/dart_package/foo/b.dart";
```

Bien que possible, l'importation absolue n'est pas recommandée et c'est une mauvaise moyen d'importer des bibliothèques car, dans les environnements de développement distribués, il causera probablement des problèmes lors de la localisation des fichiers.

Une URL sur le Web : de la même manière que l'utilisation d'un chemin de fichier absolu, nous pouvons ajouter l'URL d'un site Web contenant le code source d'une bibliothèque directement sur le protocole http://:

```
import "http://dartpackage.com/dart_package/foo/a.dart";
```

Un package : c'est la manière la plus courante d'importer une bibliothèque. Ici, on précise le chemin de la bibliothèque à partir de la racine du package. Nous explorerons la définition des packages plus loin dans ce chapitre; dans le cas de l'importation d'une bibliothèque locale, elle part *de la racine de le package, dans l'arborescence des sources jusqu'au fichier de bibliothèque*:

```
import 'package: mon_package / foo / a.dart';
import 'package: mon_package / foo / b.dart';
```

La méthode de package est la méthode recommandée pour importer des bibliothèques, car elle fonctionne bien avec *les bibliothèques locales* (c'est-à-dire, les fichiers locaux et les bibliothèques de votre projet) et est le moyen d'utiliser le bibliothèques fournies à partir de packages tiers.

N'hésitez pas à revoir l'exemple du package après avoir appris ce qu'est un package est dans le contexte Dart. Vous pouvez trouver le code source de ce chapitre sur GitHub.

Création de bibliothèques Dart

Une bibliothèque Dart peut être composée d'un seul fichier ou de plusieurs fichiers. Dans les plus courants et méthode recommandée, lorsque vous créez un fichier, vous créez une petite bibliothèque. Mais, si vous préférez, vous pouvez diviser une définition de bibliothèque en plusieurs fichiers. Bien que moins courant, il peut être utile selon le contexte, en particulier lorsque l'on travaille avec des classes, par exemple.

[63]

Psaumes 82

La décision de fractionner est importante, non seulement pour l'encapsulation mais aussi pour la façon dont la bibliothèque les clients les importeront et les utiliseront. Disons, par exemple, que nous avons deux étroitement classes couplées qui doivent vivre ensemble pour qu'elles puissent travailler. Les diviser en différents les bibliothèques obligeraient les clients à importer les deux bibliothèques. Ce n'est pas le moyen le plus pratique, alors il est très important de faire attention au fractionnement des bibliothèques lors de la création de bibliothèques open source.

Avant d'entrer dans d'autres manières de définir une bibliothèque, nous devons jeter un coup d'œil à la bibliothèque intimité; cela aide à l'encapsulation, ce qui permet de comprendre plus facilement pourquoi nous devons correctement diviser une bibliothèque en plusieurs fichiers ou non.

Confidentialité des membres de la bibliothèque

Le moyen le plus courant de contrôler la confidentialité (*encapsulation de code*), dans la plupart des langues, se produit au niveau de la classe. C'est en ajoutant un mot clé spécifique qui identifie le niveau de membre d'accès, comme protégé et privé en langage Java, par exemple; Prendre en compte schéma suivant:

Dans Dart, chaque identifiant, par défaut, est accessible depuis n'importe quel endroit, à l'intérieur et à l'extérieur du bibliothèque, sauf si elle est précédée d'un caractère _ (trait de soulignement). Cela signifie qu'il devient private aux bibliothèques déclarantes, l'empêchant d'être accessible de l'extérieur. Prendre un coup d'oeil à l'exemple suivant, où nous avons utilisé le préfixe _.

Le méta-package Dart fournit l'annotation @protected. Quand ajouté à un membre de classe, il indique que le membre doit être utilisé uniquement à l'intérieur de la classe ou de ses sous-types.

[64]

Piste 83

Programmation de fléchettes intermédiaire

Chapitre 2

De plus, notez que cette partie de Dart est très susceptible de changer dans les versions futures, en tant que partie de la communauté Dart a été influencée par Java et d'autres langues, où le contrôle de la confidentialité a lieu au niveau de la classe.

La définition de la bibliothèque

Dart a un mot-clé pour définir une bibliothèque - bibliothèque, comme vous pouvez vous y attendre. Bien que facultatif, ce mot-clé est très utile lors de la création de plusieurs bibliothèques de fichiers ou pour créer de la documentation pour les bibliothèques avant de les publier en tant qu'API.

Dart a l'outil dartdoc pour générer de la documentation HTML pour Dart paquets. Pour utiliser cet outil, nous devons rédiger des commentaires de manière spécifique, et nous explorerons cela plus en détail dans les exemples suivants.

Voyons comment définir une bibliothèque à l'aide de ce mot-clé et les multiples approches qui peuvent être prises lors de la création de bibliothèques pour faire l'encapsulation correcte et faire utilisation de la bibliothèque plus concise.

Une bibliothèque à un seul fichier

La façon la plus simple de définir une bibliothèque consiste à ajouter tout le code interdépendant, c'est-à-dire classes, fonctions de niveau supérieur et variables dans un seul fichier. Par exemple, notre La bibliothèque de personnes précédente est la suivante:

```
Classe Personne {
    String firstName;
    String lastName;
    PersonType _type;

    Personne ({this.firstName, this.lastName});

    String toString () => "($_type): $ firstName $ lastName";
}

enum PersonType {étudiant, employé}

classe L'élève étend la personne {
    Étudiant ({firstName, lastName})
```

```

        : super (firstName: firstName, lastName: lastName) {
        _type = PersonType.student;
    }
}

```

[65]

Psaumes 84

*Programmation de fléchettes intermédiaire**Chapitre 2*

```

Le programmeur de classe étend la personne {
    Programmeur ({firstName, lastName})
        : super (firstName: firstName, lastName: lastName) {
        _type = PersonType.employee;
    }
}

```

Il n'y a rien de nouveau à noter ici dans la définition du fichier, juste les deux observations suivantes:

Le fichier, en lui-même, est une bibliothèque, nous n'avons donc pas besoin de déclarer quoi que ce soit explicitement.
Le champ `_type` est privé pour la bibliothèque, c'est-à-dire qu'il n'est accessible que par le code de cette même bibliothèque.

Disons que nous essayons d'utiliser ces classes d'une autre bibliothèque, comme suit:

```

principale() {
    Programmer programmer = Programmer (firstName: "Dean", lastName: "Pugh");

    // nous ne pouvons pas accéder à la propriété _type car elle est privée pour le
    // programmeur de bibliothèque de fichier _unique._type = PersonType.employee;

    imprimer (programmeur);
}

```

Comme vous pouvez le voir, nous avons accès à tous les identifiants publics de la définition précédemment définie bibliothèque. Nous ne pouvons pas accéder à la propriété `_type` pour définir la valeur, bien que, dans la méthode `toString ()` de la classe `Person`, sa valeur est exposée.

Bien qu'il soit tentant de définir tout le code associé dans un seul fichier, cela peut devenir plus difficile à maintenir, car le code et sa complexité augmentent avec le temps. Utilisez plutôt ceci pour types simples de définitions qui ne changeront probablement pas avec le temps.

Diviser les bibliothèques en plusieurs fichiers

Nous avons vu l'approche à fichier unique pour définir une bibliothèque, alors explorons maintenant comment diviser la définition de la bibliothèque en plusieurs fichiers pour nous permettre d'organiser le projet en petit, réutilisable pièces (ce qui est le véritable objectif de l'utilisation des bibliothèques).

[66]

Piste 85

*Programmation de fléchettes intermédiaire**Chapitre 2*

Pour définir une bibliothèque multi-fichiers, nous pouvons utiliser la partie combinée, partie de, et déclarations de la bibliothèque:

part: Cela permet à une bibliothèque de spécifier qu'elle est composée de petites parties de bibliothèque.

partie de: La petite partie bibliothèque spécifie la bibliothèque qu'elle aide à composer.

bibliothèque: Ceci est pour utiliser les instructions de partie précédentes, car nous devons reliez les fichiers pièce à la partie principale de la bibliothèque.

Examinons à quoi ressemble l'exemple précédent en utilisant les instructions part:

```
// la partie 'principale' de la bibliothèque, person_library.dart
// défini à l'aide du mot-clé de la bibliothèque et listant les parties ci-dessous

bibliothécaire;

partie 'person_types.dart';
partie 'student.dart';
partie 'programmer.dart';

Classe Personne {
    String firstName;
    String lastName;
    PersonType _type;

    Personne ({this.firstName, this.lastName});

    String toString () => "($_type): $firstName $lastName";
}
```

Faisons quelques observations sur le code précédent, comme suit:

Le mot-clé de la bibliothèque est suivi de l'identifiant de la bibliothèque, personne, dans ce Cas. Il est recommandé de nommer l'identifiant en utilisant uniquement des caractères minuscules et le caractère de soulignement comme séparateur. Notre exemple pourrait s'appeler quelque chose comme person_lib ou person_library.

Les composants de la bibliothèque sont répertoriés juste en dessous de la définition de la bibliothèque.

Le code lui-même ne change rien.

[67]

Psaumes 86

Programmation de fléchettes intermédiaire

Chapitre 2

La syntaxe de la pièce est définie comme suit:

La partie PersonType est définie dans le fichier person_types.dart:

partie de la personne;

enum PersonType {étudiant, employé}

La partie Student est définie dans le fichier student.dart:

partie de la personne;

```
classe L'élève étend la personne {
    Étudiant ({firstName, lastName})
        : super (firstName: firstName, lastName: lastName) {
            _type = PersonType.student;
        }
}
```

La partie Programmer est définie dans le fichier programmer.dart:

partie de la personne;

```
Le programmeur de classe étend la personne {
    Programmeur ({firstName, lastName})
        : super (firstName: firstName, lastName: lastName) {
```

```

        } _type = PersonType.employee;
    }
}

```

La mise en œuvre en elle-même ne change rien; le seul
la différence est la partie de l'instruction au début du fichier.

De plus, comme vous pouvez le voir, la propriété `_type` est également accessible dans les fichiers pièce, car elle
privé à la bibliothèque personnelle et tous les fichiers sont dans la même bibliothèque.

Si les fichiers pièce contenaient des champs, des classes ou des fonctions de niveau supérieur et
variables préfixées par `_`, elles seraient accessibles au fichier de bibliothèque principal
et d'autres pièces également; rappelez-vous, ils sont tous dans la même bibliothèque.

[68]

Psaumes 87

Programmation de fléchettes intermédiaire

Chapitre 2

Jetons un coup d'œil au code suivant, qui utilise la bibliothèque de personnes:

```

import 'person_lib / person_library.dart';

principale() {
    // l'accès à la classe Programmer est autorisé, faisant partie de la person_library
    Programmer programmer = Programmer(firstName: "Dean", lastName: "Pugh");

    // ne peut pas accéder à la propriété _type, c'est une bibliothèque privée pour personne
    // programmer._type = PersonType.employee;

    imprimer(programmeur);
}

```

Jetez un œil au code précédent; le client de la bibliothèque personne n'a pas besoin de changer
quoi que ce soit, car les modifications que nous avons apportées sont dans la structure interne de la bibliothèque.

La syntaxe de la pièce est en train de changer et est un candidat à abandonner dans le
prochaine version de Dart. Si cela se produit, le changement le plus probable sera la création
de nouvelle syntaxe pour le remplacer.

Une bibliothèque multi-fichiers - l'instruction d'exportation

L'approche précédente n'est pas la manière idéale de fractionner une bibliothèque Dart, comme déjà
mentionné. En effet, la syntaxe de l'instruction part est susceptible de changer à l'avenir
versions. De plus, vous l'avez peut-être trouvé un peu exagéré et difficile à utiliser si vous
veulent juste contrôler la visibilité des membres de la bibliothèque.

Nous pouvons choisir de ne pas créer les parties de la bibliothèque et de simplement diviser la bibliothèque en petits
bibliothèques individuelles. Pour les exemples précédents, cela entraînerait des
changements pendant la mise en œuvre.

Nous avons les parties précédentes sous forme de trois bibliothèques individuelles: `person_library`, `programmer`,
et `étudiant`. Bien que liés, ils se comportent comme des bibliothèques individuelles et ne savent pas
tout sauf les membres publics les uns des autres:

```

// bibliothèque de personnes définie dans person_library.dart
Classe Personne {
    String firstName;
    String lastName;
    type de PersonType final;

    Personne ({this.firstName, this.lastName, this.type});
}

```

[69]

Psaumes 88

*Programmation de fléchettes intermédiaire**Chapitre 2*

```
String toString () => "($ type): $ firstName $ lastName";
}

enum PersonType {étudiant, employé}
```

La bibliothèque personnelle n'a pas besoin de l'identificateur de bibliothèque dans ce cas.

La bibliothèque de programmeur importe la bibliothèque de personnes pour accéder à sa classe Person:

```
// bibliothèque de programmeur définie dans programmer.dart

import 'person_library.dart';

Le programmeur de classe étend la personne {
    Programmeur ({firstName, lastName})
    : super (firstName: firstName, lastName: lastName, tapez:
        PersonType.employee);
}
```

De la même manière, la bibliothèque étudiante importe la bibliothèque personne:

```
// bibliothèque étudiante définie dans student.dart

import 'person_library.dart';

classe L'élève étend la personne {
    Étudiant ({firstName, lastName})
    : super (
        Prénom,
        Nom de familleNom de famille,
        type: PersonType.student,
    );
}
```

Vous pouvez voir ce qui suit à partir du code précédent:

Les bibliothèques du programmeur et des étudiants doivent importer la bibliothèque personnelle dans le prolonger.
En outre, la propriété type de la classe Person a été rendue publique par suppression du préfixe `_`. Cela signifie qu'il est accessible par les autres bibliothèques.
Comme la propriété type, dans ce cas, n'est pas destinée à changer et elle est initialisée dans le constructeur, nous l'avons également rendu définitif.

Jetons un coup d'œil au client de la bibliothèque, comme suit:

```
import 'person_lib / programmer.dart';
import 'person_lib / student.dart';
```

[70]

Psaumes 89

*Programmation de fléchettes intermédiaire**Chapitre 2*

```
principale() {
    // nous pouvons accéder à la classe Programmer car elle fait partie de la person_library
    Programmer programmer = Programmer (firstName: "Dean", lastName: "Pugh");
    Etudiant étudiant = Etudiant (prénom: "Dilo", nom: "Pugh");

    imprimer (programmeur);
    imprimer (étudiant);
}
```

Le client de la bibliothèque personnelle aura un petit changement, car maintenant la bibliothèque est divisée en

plusieurs parties, nous devrons donc importer chaque bibliothèque que nous voulons utiliser individuellement.

Ce n'est pas un gros problème quand on parle de petites bibliothèques, mais essayez de penser à un structure de bibliothèque plus complexe, où l'importation de toutes les bibliothèques interdépendantes individuellement ajouterait des difficultés à son utilisation.

C'est là que l'instruction d'exportation entre en jeu. Ici, nous pouvons sélectionner le fichier de bibliothèque principal et, à partir de là, exportez toutes les petites bibliothèques qui lui sont associées. De cette manière, le client uniquement doit importer une seule bibliothèque et toutes les plus petites bibliothèques seront disponibles à côté.

Dans notre exemple, le meilleur choix pour utiliser ceci pourrait être la bibliothèque de personnes:

```
export 'programmer.dart';
export 'student.dart';
```

```
Classe Personne {...}
```

```
énumération PersonType {...}
```

De cette façon, le client de la bibliothèque serait comme suit:

```
import 'person_lib / person_library.dart';

principale() {
    // nous pouvons accéder à la classe Programmer et Student lors de leur exportation
    // de la bibliothèque_personne
    Programmer programmer = Programmer(firstName: "Dean", lastName: "Pugh");
    Etudiant étudiant = Etudiant(prénom: "Dilo", nom: "Pugh");

    imprimer (programmeur);
    imprimer (étudiant);
}
```

Notez que seule l'instruction d'importation change. Nous pouvons utiliser les classes du petit bibliothèques normalement telles qu'elles sont exportées depuis person_library.

[71]

Piste 90

Programmation de fléchettes intermédiaire

Chapitre 2

Après avoir acquis une compréhension du concept de bibliothèque Dart, nous pouvons maintenant examiner comment combinez ces morceaux de code en quelque chose de partageable et réutilisable: le package Dart.

Paquets de fléchettes

Un package Dart est le point de départ de tout projet Dart. Dans les exemples précédents, nous n'avons pas dérangez-vous car nous utilisions des exemples de syntaxe de fichier unique; cependant, dans le monde réel, nous fonctionnera toujours avec des *packages*:

Le principal avantage de l'utilisation et de la création de packages est que le code peut être réutilisé et partagé. Dans l'écosystème Dart, cela se fait par l'outil pub, qui nous permet d'extraire et d'envoyer dépendances au site Web et au référentiel pub.dartlang.org.

L'utilisation d'un package de bibliothèque dans un projet en fait une dépendance *immédiate*, et le La bibliothèque utilisée peut avoir ses propres dépendances, appelées dépendances **transitives**.

Si vous jouez avec DartPad, il est temps de changer; maintenant, vous aurez besoin d'un environnement de développement Dart approprié configuré, comme nous allons commencer à travailler avec des packages.

En général, il existe deux types de packages Dart: **les packages d'application et la bibliothèque paquets**.

Packages d'application et packages de bibliothèque

Tous les packages ne sont pas censés être partageables; une application elle-même est également un package. Celles-ci les packages peuvent avoir des dépendances sur les packages de bibliothèque normalement, mais ils ne sont pas destinés à être utilisé comme une dépendance dans d'autres projets.

[72]

Piste 91

Programmation de fléchettes intermédiaire

Chapitre 2

D'un autre côté, les packages de bibliothèque sont ceux qui contiennent du code utile qui peut être utile dans de nombreux projets. Ces types peuvent être utilisés comme dépendance et avoir des dépendances sur d'autres aussi.

En termes simples, la structure recommandée d'un package Dart ne diffère pas trop entre une application et un package de bibliothèque - leur objectif et leur utilisation sont différents de L'une et l'autre.

Structures de package

La première chose importante à souligner à propos d'une structure de projet de package Dart est que son la validité est affirmée par la présence d'un fichier pubspec.yaml; c'est-à-dire s'il y a un fichier pubspec.yaml dans votre structure, puis il y a un package et c'est là que vous Décrivez-le correctement - sans lui, il n'y a pas du tout de paquet. C'est ce qu'un package typique ressemble à:

Cet exemple de package a été généré à l'aide de l'outil *Stagehand*. Vous pouvez reportez-vous à la section suivante pour plus de détails.

Pour les packages d'application, aucune mise en page de projet n'est requise (car elle n'est pas destinée à être publié dans le répertoire de publication); cependant, au fur et à mesure de son évolution, il existe déjà plusieurs les méthodes et conventions recommandées à suivre. Jetons un coup d'œil à la structure commune de un package Dart général. La plupart de la structure est conventionnelle et dépend de votre projet complexité et si vous souhaitez partager son code d'une manière ou d'une autre.

[73]

Psaumes 92

Jetons un coup d'œil au rôle de chaque fichier et répertoire dans une structure de package Dart typique:

`pubspec.yaml`: Comme déjà indiqué, il s'agit du fichier de package fondamental et il décrit le package dans le référentiel pub. Nous allons examiner le plein structure de ce fichier plus en détail ultérieurement.

Les répertoires `lib` / `lib` / `src` /: ce sont les endroits où le code source de la bibliothèque de paquets vit. Comme vous le savez déjà, un simple fichier `.dart` est une *petite* bibliothèque, donc tout ce que vous mettez dans le répertoire `lib` est accessible au public autres forfaits. C'est ce qu'on appelle l'API publique du package. Le sous-répertoire `src` contient, par convention, tout le code interne du package, c'est-à-dire le code source privé du package qui n'est pas censé être directement importés par d'autres.

Bien qu'il soit possible d'importer une bibliothèque placée dans le sous-répertoire `src`, ce n'est pas recommandé, car il est destiné à être une bibliothèque interne implémentation et ne fait pas partie de l'API publique de la bibliothèque. Ça peut changer et casser le code client.

`lib / simple_package_structure.dart`: Une pratique courante consiste à ajouter un seul, ou quelques fichiers de premier niveau qui exportent (souvenez-vous de l'instruction d'exportation) `src / bibliothèques locales`. Le nom de ce fichier est généralement le même que celui du package. Si il y a plus d'une bibliothèque, alors le nom doit être suffisamment simple pour être identifié l'objectif général des bibliothèques exportées.

`test /`: les tests unitaires et les analyses de référence sont classiquement placés à l'intérieur les répertoires de test et de référence, respectivement. De plus, le code source à l'intérieur du dossier de test est généralement postfixé avec l'identificateur `_test`.

Vous pouvez consulter la section *Une introduction aux tests unitaires avec Dart* pour comprendre comment écrire des tests unitaires.

`README.md`, `CHANGELOG.md` et `LICENSE`: ce sont généralement des fichiers de démarque présents dans des packages destinés à être publiés dans un référentiel public, comme le pub Dart. Ces fichiers sont également très courants en open source projets. Le fichier `LICENSE`, qui spécifie le copyright du code source information, est également parfois présente.

exemple /: Ceci est important dans les packages publiés et peut montrer comment le package peut être utilisé.

`analysis_options.yaml`: C'est un fichier utile pour personnaliser les contrôles de peluches, le style analyse et autres vérifications de précompilation.

[74]

Piste 93

Vous pouvez consulter le didacticiel de personnalisation de l'analyse sur le site Web de Dart à <https://www.dartlang.org/guides/language/error-analysis.html>

Certains fichiers supplémentaires dépendent de l'objectif du projet, notamment les suivants:

`tools /`: c'est un répertoire contenant des scripts qui peuvent être utilisés pendant développement, y compris des utilitaires pour manipuler des images, des fichiers bruts et tout type de script qui est privé du package et utile au développeur.

`doc / et doc / api`: c'est ici que vous pouvez ajouter des informations utiles sur le projet. `api` / sous-répertoire est l'endroit où l'outil `dartdoc` (présenté au [chapitre 1](#), *An Introduction to Dart*) génère la documentation de l'API basée sur commentaires de code.

Dans les packages Web, de nouveaux fichiers et répertoires sont inclus; ils sont les suivants:

Le dossier lib / est la destination typique des fichiers de ressources Web statiques, tels que images ou fichiers .css.

web / est un répertoire utilisé dans les projets d'application Web. Contrairement au dossier lib /, qui est censé être un code de bibliothèque, ce code est destiné à avoir l'application Web code source et points d'entrée (c'est-à-dire la fonction main ()).

Dans les packages de ligne de commande, le répertoire bin est inclus:

Le répertoire bin / est censé avoir un script qui peut s'exécuter directement à partir du ligne de commande; l'outil Stagehand décrit ci-après est un exemple de la commande-outil de bibliothèque de lignes.

La structure du projet Flutter présente certaines similitudes avec les packages Dart et nous en apprendrons davantage sur cette structure dans le chapitre suivant.

Stagehand - le générateur de projet Dart

Démarrer un nouveau projet Dart nécessite quelques étapes simples: créer un dossier vide, ajouter un fichier pubspec.yaml et décrivez le package avec un nom, une version, etc. Ensuite, vous ajoutez les fichiers nécessaires progressivement.

[75]

Épisode 94

Programmation de fléchettes intermédiaire

Chapitre 2

En général, la plupart des fichiers et leurs structures ne changent pas d'un package à l'autre, donc créer à chaque fois toute la structure du package Dart peut être fastidieux. C'est pourquoi le Outil Stagehand a été créé pour générer des projets d'échafaudage Dart.

Pour exécuter l'outil Stagehand, nous devons d'abord l'installer sur notre système. Dans un environnement correctement configuré Environnement Dart, exécutez la commande pub suivante dans un terminal pour l'installer:

pub global activer stagehand

L'outil pub est présent dans le SDK Dart. Si vous avez une fléchette ou un scintillement environnement prêt, vous pouvez utiliser cet outil. Sinon, regardez à nouveau à [Chapitre 1, Une introduction à Dart](#).

Cette commande télécharge un package (Stagehand, dans ce cas) à partir du référentiel pub et l'installe dans le répertoire de cache des packages Dart de votre système. Cela varie en fonction de votre système d'exploitation: \$ HOME / .pub-cache / bin sur les systèmes Linux et AppData \ Roaming \ Pub \ Cache \ bin sous Windows.

Pour exécuter Stagehand et tout autre outil de package activé global à partir de la ligne de commande, vous peut utiliser l'une des deux méthodes suivantes:

La première consiste à faire précéder la commande d'outil de ce qui suit:

pub géré dans le monde

La seconde consiste à ajouter le répertoire de cache des packages globaux de Dart dans le chemin du système d'exploitation.

Après avoir correctement installé et configuré l'outil Stagehand, vous pouvez commencer à générer Dart projets:

1. Tout d'abord, créez un dossier vide avec le nom de package souhaité.

Jetez un œil à la description du champ de nom dans la section *Fichier pubspec* pour comprendre comment nommer correctement votre colis.

[76]

Psaumes 95

Programmation de fléchettes intermédiaire

Chapitre 2

2. Ensuite, dans le dossier créé, générez la structure du package à l'aide du commande suivante:

```
pub run global stagehand <modèle>
```

Alternativement, si votre chemin est correctement configuré, vous pouvez utiliser stagehand <template>, où <template> est le modèle Stagehand à utiliser.

Vous pouvez vérifier les modèles de projet disponibles sur la page projet du Site Web du pub Dart à <https://pub.dev/packages/>

Le fichier pubspec

Le fichier pubspec est au cœur d'un package Dart, et pour comprendre comment correctement décrire le package, nous devons comprendre comment ce fichier est structuré. Ce fichier est basé sur la syntaxe yaml, un format commun utilisé pour les fichiers de configuration, avec une structure qui est facile à lire et à suivre. Le fichier pubspec est le suivant:

```
nom: simple_package_structure
description: un exemple de package simple
version: 1.0.0
page d'accueil: https://www.example.com
auteur: Alessandro Biessek <alessandrobiessek@gmail.com>

environnement:
  sdk: '>= 2.0.0 <3.0.0' # vérifier la section des dépendances
                           # ci-dessous pour comprendre la gestion des versions de deps

dépendances:
  json_serializable: ^2.0.1

dev_dependencies:
  test: ^1.0.0
```

Les projets Flutter contiennent également un fichier pubspec avec des champs. Pour plus d'informations, vous pouvez vous référer à [Chapitre 3, Une introduction à Flutter](#).

[77]

Psaumes 96

Programmation de fléchettes intermédiaire

Chapitre 2

Le fichier spécifie les informations de métadonnées du package, ce qui est utile si vous souhaitez publier le paquet. Il définit également les dépendances tierces du package et le SDK Dart version. Examinons les champs pubspec plus en détail:

name: Il s'agit de l'identifiant du package. Il est *obligatoire* et devrait contenir *uniquement des* lettres minuscules et des chiffres, plus le caractère `_`; en plus, il doit être un identifiant Dart valide (c'est-à-dire qu'il ne peut pas commencer par des chiffres et ne peut pas un mot réservé). C'est une propriété très importante si vous souhaitez publier le package dans le référentiel pub, et il est bon de vérifier les noms de packages existants pour éviter les doubles emplois.

description: bien qu'il s'agisse d'un champ facultatif, il est obligatoire si vous avez l'intention de publier le package, en décrivant en termes simples le but du package.

version: Ceci est également facultatif pour les packages personnels, mais il est obligatoire pour publication dans le référentiel pub. Il est important de maintenir la cohérence dans le versioning d'un package qui sera utilisable par la communauté.

page d'accueil: pour les packages pub, cela sera lié à la page du package sur le pub site Internet. Il est très important d'en fournir un lorsque vous prévoyez de le publier.

auteurs: Bien que ce ne soit pas obligatoire, il est important de fournir les coordonnées sur le ou les créateurs de la bibliothèque. De plus, une bibliothèque peut avoir plus d'un auteur; dans ce cas, la syntaxe de la liste YAML peut être utilisée en définissant le champ auteurs à la place (notez les coordonnées facultatives):

auteurs:
 - Alessandro Biessek <alessandrobiessek@gmail.com>
 - Alessandro Biessek

dépendances et dev_dependencies: elles font référence à l'objectif réel de le fichier pubspec. Une liste des packages tiers est requise pour l'utilisation du bibliothèque et le développement de la bibliothèque, respectivement.

environnement: Outre les dépendances tierces, il y en a une autre, disons, la *principale* dépendance d'un package, qui est le SDK Dart lui-même. Dans ce champ, vous devez spécifier la cible et les versions prises en charge du SDK Dart.

Le champ d'environnement est la dépendance du SDK; il est recommandé que vous spécifiez la version cible du SDK Dart à l'aide de la syntaxe de plage, la plage séquentielle n'est pas compatible avec les anciennes versions (c'est-à-dire `<1.8.3`).

La structure pubspec typique contient les champs spécifiés précédemment. Pour une explication complète du fichier pubspec et des autres champs spécifiques à un usage, jetez un œil à le site Web de Dart: <https://.. / www>

[78]

Épisode 97

Programmation de fléchettes intermédiaire

Chapitre 2

Vous pouvez utiliser le caractère `#` pour commencer un commentaire dans yaml.

Dépendances de paquet - pub

Maintenant que vous comprenez le rôle le plus important du fichier pubspec dans le package lorsque développant des applications Dart, vous pouvez ajouter des dépendances de packages tiers à votre projet. Il existe d'importantes commandes de pub avec lesquelles vous pouvez travailler lors de l'ajout ou mettre à jour les dépendances de package dans votre projet. Nous devons également montrer comment spécifiez correctement la version de dépendance que nous devons utiliser.

Après avoir démarré un nouveau projet Dart, soit manuellement, soit à l'aide d'un outil générateur tel que Stagehand, la première chose à faire est d'exécuter la commande suivante:

`pub obtenir`

Par exemple, le package suivant contient uniquement le fichier pubspec suivant:

De plus, il contient le contenu de pubspec, comme suit:

```
nom: add_dependencies
```

Il s'agit d'une description de package minimale et il n'a aucune dépendance spécifiée, pas même la version cible du SDK Dart. Cependant, exécutons la commande pub get à l'intérieur le dossier du package, car il fonctionnera de la même manière:

```
pub obtenir
```

Nous obtenons la sortie réussie suivante:

```
Résolution des dépendances ...
Vous avez des dépendances!
```

[79]

Psaumes 98

Programmation de fléchettes intermédiaire

Chapitre 2

Nous obtiendrons une structure de fichier comme dans la capture d'écran suivante:

Notez les nouveaux fichiers générés par la commande dans le dossier .packages; ces fichiers sont importants pour que l'outil pub fonctionne avec les packages de dépendances:

.packages: cela mappe les dépendances dans le cache pub du système (précédemment mentionné dans le *Stagehand - la section du générateur de projet Dart*). Au lieu de faire copies dans tous vos packages, l'outil pub stocke simplement le mappage entre l'emballage et son emplacement respectif dans le système. Une fois le colis mappé ici, il sera disponible pour que vous puissiez l'importer dans votre code Dart. Ce fichier ne doit pas être inclus dans le système de gestion du code source; c'est car il est généré et géré par l'outil pub.

pubspec.lock: il s'agit du fichier auxiliaire de l'outil pub qui contient tous les graphes de dépendance du package, c'est-à-dire qu'il répertorie tous les dépendances et les transitifs. Il contient également les versions exactes et d'autres informations de métadonnées sur toutes les dépendances. C'est recommandé que vous incluez ce fichier dans le système de gestion source uniquement s'il s'agit d'un package de l'application; cela aide une équipe de développement, par exemple, à travailler avec l'exact même configuration de dépendance. Si vous utilisez un package de bibliothèque, c'est généralement non inclus, car il devrait fonctionner avec une large gamme de dépendances, c'est-à-dire qu'il ne doit pas être verrouillé sur des versions spécifiques.

N'oubliez pas que tout cela est fait par l'outil pub, vous ne devez donc pas les toucher des dossiers.

Spécifier les dépendances

Maintenant que vous savez comment l'outil pub résout les packages à l'intérieur du projet, jetons un œil

comment y ajouter des dépendances.

[80]

Psautres 99

Programmation de fléchettes intermédiaire

Chapitre 2

Les dépendances sont spécifiées dans le champ dependencies du fichier pubspec. C'est une liste YAML champ, vous pouvez donc en spécifier autant que nécessaire dans le champ. Supposons que nous ayons besoin le package json_serializable dans notre projet. Nous pouvons le spécifier en ajoutant simplement au liste, comme suit:

```
nom: add_dependencies
dépendances:
  json_serializable:
    # un autre forfait ci-dessous
```

La syntaxe pour spécifier une dépendance est la suivante:

```
<paquet>:
  <contraintes>
```

Ici, vous ajoutez son nom (<package>) suivi des champs <constraints>: version et la source. Dans ce cas, nous n'avons spécifié aucune contrainte, il suppose donc version disponible pour la contrainte de version et la source par défaut (pub.dartlang.org).

Notez que les deux points,: après le nom du package *ne sont pas* facultatifs; la La liste de dépendances s'attend à ce que chaque dépendance soit une valeur de mappage YAML. Pour plus d'informations vous pouvez consulter la documentation YAML à <https://docs.dartlang.org/en/latest/dart/pub/specification.html#yaml-syntax>.

La contrainte de version

La contrainte de version peut être un numéro de version concret, une plage ou un minimum ou contrainte maximale. Explorons à quoi cela ressemble dans chaque situation:

Any / empty : Comme dans l'exemple précédent, nous pouvons l'utiliser sans version contrainte, par exemple, json_serializable: ou json_serializable: any.

Version concrète : nous pouvons ajouter le numéro de version spécifique que nous voulons travailler avec, par exemple, json_serializable: 2.0.1.

Limite minimale : Ici, nous pouvons ajouter une version minimale acceptable du package nous voulons de deux manières: json_serializable: '> 1.0.0', où nous acceptons tout version ultérieure à la version spécifiée (à l'exclusion de celle spécifiée), ou json_serializable: '>= 1.0.0', où nous acceptons n'importe quelle version supérieur ou égal à la version spécifiée.

[81]

Piste 100

Programmation de fléchettes intermédiaire

Chapitre 2

Limite maximale : comme l'exemple minimum précédent mais dans la limite supérieure, nous pouvons ajouter une version maximale acceptable du package que nous voulons en deux manières: json_serializable: '<2.0.1', où nous acceptons tout version inférieure à celle spécifiée, ou json_serializable: '<= 2.0.1', où nous acceptons toute version inférieure ou égale à celle spécifiée.

Plage : en combinant des limites minimales et maximales, nous pouvons spécifier un intervalle de versions acceptable: json_serializable: '> 1.0.0 <= 2.0.1', json_serializable: '> 1.0.0 <2.0.1', ou json_serializable: '> = 1.0.0 <= 2.0.1'.

Plage sémantique : Ceci est similaire à plage mais, en utilisant le caractère caret, nous peut spécifier la plage entre une version minimale acceptable et la prochaine rupture changement. Par exemple, json_serializable: ^ 1.0.0 est le même comme json_serializable: '> = 1.0.0 <2.0.0', et json_serializable: ^ 0.1.0 est égal à json_serializable: '> = 0.1.0 <0.2.0'.

Le versionnage sémantique aide la communauté à utiliser les bibliothèques et il est largement adoptée. Pour l'examiner plus en détail, vous pouvez visiter les outils de pub page par page <https://>.

La contrainte source

L'outil pub ne recherche pas uniquement les packages dans le référentiel pub; si tu as déjà utilisé un autre système de gestion de paquets, vous savez qu'il peut être utile, parfois, de hébergez vos packages dans d'autres endroits que le référentiel public, tel que l'entreprise privée packages ou ceux de votre utilisation personnelle. Pour la partie source de la spécification du package, nous ont quatre alternatives pour changer où l'outil de publication doit rechercher le package:

La source hébergée : il s'agit du dépôt de publication par défaut ou d'une autre alternative http serveur qui implémente l'API pub. Par exemple, considérez le code suivant bloquer:

```
dépendances:
  json_serializable :
    hébergé :
      nom : json_serializable
      url : http://pub-packages-private-server.com # changement de serveur
```

[82]

Épisode 101

Programmation de fléchettes intermédiaire

Chapitre 2

Comme vous pouvez le voir, nous n'avons besoin de spécifier le champ hébergé que si nous n'utilisons pas le référentiel pub, c'est-à-dire la source par défaut.

La source du chemin : Ici, vous pouvez ajouter une dépendance à un package dans votre propre système:

```
dépendances:
  json_serializable :
    chemin: / Users / biessek / json_serializable
```

Bien que vous ne soyez pas autorisé à partager un package avec ce type de dépendance, il peut être utile dans les étapes de développement.

La source Git : Ici, vous pouvez spécifier un package à partir d'un dépôt git:

```
dépendances:
  json_serializable :
    git :
      url : git: //github.com/dart-lang/json_serializable.git
      path : path / to / json_serializable # si la racine du package est
                                                # pas la racine du
                                                # référentiel
      ref : master # pour dépendre d'un commit, d'une balise, d'une branche spécifiques
```

Cela peut être utile dans les étapes de développement ou si une source de package publiée le code n'est pas encore présent dans le dépôt pub.

La source du SDK : un SDK peut avoir ses propres packages qui peuvent être utilisés comme

dépendances:

```
 dépendances:
 flutter_localizations : # une dépendance disponible dans le sdk flutter
 sdk : scintillement
```

Jusqu'à présent, cette façon de spécifier les contraintes source n'est utilisée que pour le SDK Flutter dépendances.

Les dépendances de package sont un sujet fondamental dans le développement de Dart; avec ces concepts dans l'esprit, vous pouvez ajouter des dépendances tierces utiles à vos projets et augmenter votre productivité.

[83]

Épisode 102

Programmation de fléchettes intermédiaire

Chapitre 2

Présentation de la programmation asynchrone avec Futures et isolats

Dart est un langage de programmation monothread, c'est-à-dire que tout le code de l'application s'exécute dans le même fil. En termes simples, cela signifie que tout code peut bloquer l'exécution du thread en exécuter des opérations de longue durée telles que des demandes d'E / S ou http.

Bien que Dart soit monothread, il peut effectuer des opérations asynchrones via l'utilisation des **Futures**. De plus, pour représenter le résultat de ces opérations asynchrones, Dart utilise l'objet Future combiné avec les mots-clés `async` et `await`. Comprendons ces concepts importants pour développer une application réactive.

Futures de fléchettes

L'objet Future <T> dans Dart représente une valeur qui sera fournie à un moment donné dans le futur. Il peut être utilisé pour marquer une méthode, par exemple, avec un résultat futur; c'est-à-dire une méthode renvoyer un objet Future <T> n'aura pas la valeur de résultat appropriée immédiatement mais, au lieu de cela, après quelques calculs à un moment ultérieur.

Considérez le code suivant, où nous avons la fonction principale qui appelle un opération:

```
import 'fléchette: io';

void longRunningOperation () {
    pour (int i = 0; i <5; i++) {
        sommeil (Durée (secondes: 1));
        print ("index: $ i");
    }
}

principale() {
    print ("début de l'opération longue");

    longRunningOperation ();

    impression ("corps principal continu");

    pour (int i = 10; i <15; i++) {
        sommeil (Durée (secondes: 1));
        print ("index de main: $ i");
    }
}
```

[84]

Épisode 103

Programmation de fléchettes intermédiaire

Chapitre 2

```
    print ("fin du principal");
}
```

Si vous exécutez le code précédent, vous remarquerez qu'il arrête l'exécution de la fonction principale pendant que la fonction longRunningOperation () est en cours d'exécution. C'est une exécution synchrone de tout le code et il ne s'intégrera probablement pas bien dans tous les cas d'utilisation.

Maintenant, disons que la fonction longRunningOperation () est une fonction asynchrone et main () peut continuer à s'exécuter sans attendre la fin pour continuer:

```
import 'dart:async';

Future longRunningOperation () async {
    pour (int i = 0; i < 5; i++) {
        attendre Future.delayed (Durée (secondes: 1));
        print ("index: $ i");
    }
}

main () {...} // la fonction principale est la même
```

Nous avons apporté quelques modifications pour montrer comment Future fonctionne correctement:

La fonction longRunningOperation () a maintenant le modificateur `async` pour indiquer que cela renverra une fonction Future et que la fonction Future sera terminé à la fin de l'exécution de la fonction. Notez que le type de retour est aussi Future.

Nous avons remplacé l'appel `sleep()` par l'appel `Future.delayed`. Ça aussi démontrer l'utilisation du mot clé `await`. Le mot clé `await` fonctionne avec des fonctions asynchrones. Lors de l'appel d'une fonction Future, nous pouvons avoir besoin du résultat de la fonction Future pour continuer l'exécution. Dans ce cas, nous voulons procéder à l'impression uniquement après le délai spécifié.

Si vous exécutez le code précédent, vous remarquerez peut-être quelque chose d'étrange; la sortie est comme suit:

```
début d'une opération longue durée
corps principal continu
index du principal: 10
index du principal: 11
index du principal: 12
indice principal: 13
index du principal: 14
fin de main
indice: 0
```

[85]

Épisode 104

Programmation de fléchettes intermédiaire

Chapitre 2

```
indice: 1
indice: 2
indice: 3
indice: 4
```

Ce n'est pas un code simultané où un code s'exécute après un autre comme avant; ici, quoi les changements est l'ordre. Dans l'exemple précédent, le changement se produit lorsque les appels de fonction longRunningOperation () attendent dans une autre fonction asynchrone. Ici le fonction est suspendue et ne reprendra qu'après un délai de 1 seconde. Après le retard, cependant, la fonction principale est déjà en cours d'exécution car elle n'attend plus longtemps l'opération à terminer, de sorte que le code longRunningOperation () ne sera exécuté qu'après

la fonction principale est terminée.

Une chose que nous pouvons faire est de transformer la fonction main () en une fonction asynchrone et d'attendre le exécution de longRunningOperation (). De cette façon, la fonction main () sera suspendu à droite lorsque nous appelons wait longRunningOperation () et ne sera repris après son exécution. Cela se comporte comme un code synchrone normal, comme suit:

```
main () async {
    print ("début de l'opération longue");

    attendre longRunningOperation ();

    impression ("corps principal continu");

    pour (int i = 10; i < 15; i++) {
        sommeil (Durée (secondes: 1));
        print ("index de main: $ i");
    }

    print ("fin du principal");
}
```

Comme vous l'avez peut-être remarqué, les fonctions précédentes ne s'exécutent jamais vraiment de manière asynchrone; c'est car nous attendons l'exécution de la méthode longRunningOperation () avant exécuter le reste de son code. Pour les faire fonctionner de manière asynchrone, nous devons omettre le mot-clé await, comme suit:

```
main () async {
    print ("début de l'opération longue");

    longRunningOperation ();

    impression ("corps principal continu");

    pour (int i = 10; i < 15; i++) {
```

[86]

Épisode 105

Programmation de fléchettes intermédiaire

Chapitre 2

```
sommeil (Durée (secondes: 1));
print ("index de main: $ i");
}
print ("fin du principal");
}
```

Cela fera que la méthode main () continue simplement son exécution, où nous obtenons le sortie suivante:

```
début d'une opération longue durée
corps principal continu
indice: 0
index du principal: 10
indice: 1
index du principal: 11
indice: 2
index du principal: 12
indice: 3
indice principal: 13
indice: 4
index du principal: 14
fin de main
```

Dart exécute les deux méthodes asynchrones dans le même thread. Les deux fonctions s'exécutent de manière asynchrone dans ce cas, mais cela ne signifie pas qu'ils sont exécutés en parallèle.

Dart exécute une opération à la fois; tant qu'une opération est en cours d'exécution, il ne peut être interrompu par aucun autre code Dart.

Cette exécution est contrôlée par la boucle Dart Event, qui agit comme un gestionnaire pour Dart Futures et code asynchrone.

Vous pouvez vous référer à la documentation officielle de Dart sur la boucle de l'événement pour comprendre comment cela fonctionne: https://. / fléchette_boucle.html

[event-boucle.html](#)

Pour exécuter du code Dart en parallèle (c'est-à-dire en même temps), nous utilisons Dart Isolates.

[87]

Épisode 106

Programmation de fléchettes intermédiaire

Chapitre 2

Isolats de fléchettes

Alors, vous vous demandez peut-être comment exécuter du code vraiment parallèle et améliorer la performance et la réactivité? Les isolats de fléchettes sont là pour cela. Chaque application Dart est composé d'au moins une instance Isolate, l'instance Isolate principale, où toutes les parties du code d'application s'exécutent. Donc, pour créer un code d'exécution parallèle, nous devons créer une nouvelle instance Isolate qui peut s'exécuter en parallèle avec l'instance Isolate principale:

Les isolats peuvent être considérés comme une sorte de thread, mais ils ne partagent rien entre eux, comme son nom l'indique. Cela signifie qu'ils ne partagent pas la mémoire, donc nous n'avons pas besoin d'utiliser des verrous et d'autres techniques de synchronisation de thread ici.

Pour communiquer entre les isolats, c'est-à-dire pour envoyer et recevoir des données entre eux, nous avons besoin d'échanger des messages. Dart fournit un moyen d'accomplir cela.

Modifions l'implémentation précédente pour utiliser une instance Isolate à la place:

```
import 'fléchette: io';
import 'fléchette: isoler';

Future<void> longRunningOperation ( String message ) async {
    pour (int i = 0; i < 5; i++) {
        attendre Future.delayed (Durée (secondes: 1));
        print ("index: $ i");
    }
}

principale() {
    print ("début de l'opération longue");

    Isolate.spawn (longRunningOperation, "Bonjour");

    impression ("corps principal continu");

    pour (int i = 10; i < 15; i++) {
```

[88]

Épisode 107

*Programmation de fléchettes intermédiaire**Chapitre 2*

```

        sommeil (Durée (secondes: 1));
        print ("index de main: $ i");
    }

    print ("fin du principal");
}

```

Comme vous pouvez le voir, le code affiche de petites modifications:

La fonction longRunningOperation () devient une instance Isolate, c'est-à-dire cela reste comme une simple fonction.
Pour envoyer le processus Isolate à l'exécution, nous utilisons la méthode spawn () de la classe Isolate. Il faut deux arguments: la fonction à générer et un paramètre à passer à la fonction.

En exécutant le code précédent, vous noterez une sortie différente, comme suit:

```

début d'une opération longue durée
corps principal continu
Bonjour d'isoler
index du principal: 10
indice: 0
index du principal: 11
indice: 1
index du principal: 12
indice: 2
indice principal: 13
indice: 3
index du principal: 14
fin de main

```

Désormais, le code de ces deux fonctions s'exécute indépendamment après la génération d'Isolate.

Lors de la compilation en JavaScript, les isolats sont convertis en travailleurs Web.
Vous pouvez en savoir plus sur les travailleurs Web dans l'article W3Schools à l'adresse <https://w3schools.com>

Présentation des tests unitaires avec Dart

Dans n'importe quel langage, nous pouvons écrire du code qui accomplit un but; cependant, pour écrire code performant et sans bogue, nous devons utiliser toutes les ressources disponibles que nous pouvons.

[89]

Épisode 108

*Programmation de fléchettes intermédiaire**Chapitre 2*

Les tests unitaires sont l'une des choses qui peuvent nous aider à écrire de manière modulaire, efficace et sans bogue code. Le test unitaire n'est pas le seul moyen de tester le code, bien sûr, mais c'est une partie cruciale de tester de petits logiciels de manière à les isoler des autres composants, ce qui nous aide à se concentrer sur des choses spécifiques.

Couvrir tout le code de l'application avec des tests unitaires ne garantit pas qu'il s'agit d'un bug à 100%. libre; cependant, cela nous aide à atteindre progressivement un code mature, et c'est l'une des étapes pour assurer un bon cycle de développement, avec des versions stables de temps en temps.

Dart fournit également des outils utiles pour travailler avec des tests; jetons un coup d'œil au départ point de test unitaire Code Dart: le package de test Dart.

Le package de test Dart

Le package de test Dart ne fait pas partie du SDK lui-même, il doit donc être installé normalement

dépendance à des tiers. Vous devriez déjà savoir comment faire cela.

Pour référence, consultez l'exemple, `4_unit_tests`, dans la source de ce chapitre code sur GitHub. Le code de test se trouve dans le dossier `test/`.

Dans cet exemple (généré avec l'outil Stagehand), il existe une dépendance de développement; c'est une dépendance qui n'est requise que pendant le développement et non au moment de l'exécution:

```
dev_dependencies:  
  test: ^1.0.0
```

Cela nous permet d'utiliser les bibliothèques fournies par le package de test pour écrire des tests unitaires.

Rédaction de tests unitaires

Maintenant, supposons que nous voulions créer une fonction qui additionne deux nombres:

```
Calculatrice de classe {  
  num sumTwoNumbers (num a, num b) {  
    // FAIRE  
  }  
}
```

[90]

Épisode 109

Programmation de fléchettes intermédiaire

Chapitre 2

Nous pouvons écrire un test unitaire pour évaluer cette implémentation de méthode en utilisant le package de test:

```
import 'package: test / test.dart';  
import 'package: unit_tests / calculator.dart';  
  
void main () {  
  Calculatrice calculatrice;  
  
  installer();  
  calculatrice = calculatrice ();  
}  
  
test ('calculatrice sumTwoNumbers () additionne les deux nombres', () {  
  expect ( calculator.sumTwoNumbers (1, 2) , 3);  
});  
}
```

Dans l'exemple précédent, nous avons commencé par importer la bibliothèque principale du package de test qui expose des fonctions, par exemple: `setUp()`, `test()` et `expect()`. Chacune des fonctions a rôles spécifiques, comme suit:

`setUp()` exécutera le callback que nous lui passons avant chacun des tests du test suite.

`test()` est le test en lui-même; il reçoit une description et un rappel avec le test la mise en oeuvre.

`expect()` est utilisé pour faire les assertions sur le test. Dans le précédent Par exemple, nous affirmons simplement une somme de $1 + 2$, ce qui devrait entraîner le numéro 3.

Pour exécuter un test, nous utilisons la commande suivante:

```
test d'exécution de pub <test_file>
```

Dans l'exemple précédent, la commande serait (à partir de la racine du projet) comme suit:

```
pub exécuter test test / calculator _tests.dart
```

Avant d'implémenter efficacement la méthode sumTwoNumbers (), la sortie du test est comme suit:

00:01 +0 -1: la calculatrice sumTwoNumbers () additionne les deux nombres [E]

Attendu: <3>

Réel: <null>

package: test_api expect

[91]

Épisode 110

Programmation de fléchettes intermédiaire

Chapitre 2

test \ calculator_tests.dart 12: 7 main. <fn>

00:01 +0 -1: certains tests ont échoué.

De plus, après avoir correctement implémenté la méthode sumTwoNumbers (), nous verrons le Suivant:

00:01 +1: Tous les tests ont réussi!

Vous pouvez également créer des groupes de tests, car vous pensez peut-être qu'un seul cas de test peut ne pas être suffisant pour tester efficacement une unité de code. Supposons que nous changions notre suite de tests en ont un groupe de tests de somme, comme suit:

```
void main () {
    Calculatrice calculatrice;

    installer();
    calculatrice = calculatrice ();
};

groupe ("somme des tests", () {
    test ('calculatrice sumTwoNumbers () additionne les deux nombres', () {
        expect (calculatrice.sumTwoNumbers (1, 2), 3);
    });
    test ('calculatrice sumTwoNumbers () sum null comme il était 0', () {
        expect (calculatrice.sumTwoNumbers (1, null), 1);
    });
});
}
```

Notez la sortie du test précédent:

00:01 +1 -1: la somme teste la calculatrice sumTwoNumbers () sum null telle qu'elle était 0 [E]

NoSuchMethodError: La méthode '_addFromInteger' a été appelée sur null.

Récepteur: nul

Appel essayé: '_addFromInteger ()'

fléchette: noyau int. +

package: unit_tests / src / calculator_base.dart 3:14 Calculator.sumTwoNumbers

test \ calculator_tests.dart 15:25 main. <fn>. <fn>

00:01 +1 -1: certains tests ont échoué.

Il y a eu un test réussi (+1) et un échec (-1) - avec l'exception décrite à droite sous la description du test qui a échoué. Dans cet esprit, nous pouvons changer le sumTwoNumbers () implémentation pour accepter une valeur nulle, car elle était 0, et réexécuter le test:

00:01 +2: Tous les tests ont réussi!

[92]

Épisode 111

Programmation de fléchettes intermédiaire

Chapitre 2

Comme vous pouvez le voir, les tests peuvent nous aider à éviter que des erreurs logiques ne se produisent en production; de

Bien sûr, il se peut que nous ayons toujours des erreurs, mais les tests peuvent nous aider à éviter autant possible.

C'était une introduction aux tests unitaires avec Dart. Vous pouvez en apprendre davantage sur tous les possibilités en lisant la page du package de test sur le site Web du pub à <https://pub.dev/packages/test>.

dartlang.org/pubs/

Sommaire

Dans ce chapitre, nous avons vu comment le langage Dart est structuré en termes de Paradigme de la POO. Nous avons vu que le langage propose de fournir toutes les fonctionnalités pour le développeur lors de l'utilisation du paradigme OOP, mais aussi certaines particularités qui sont significées pour étendre les possibilités de développement, telles que les mixins pour explorer les avantages du multi-héritage et des interfaces implicites qui permettent à toute classe d'être implémentée par toute autre classe, appelable classes pour ajouter un comportement de fonction à des objets simples et des fonctions et variables de niveau supérieur qui n'ont pas besoin d'être liés à une classe. Il est très utile pour les fonctions utilitaires qui ne dépendent du contexte.

Nous avons exploré comment les packages Dart sont structurés et comment utiliser l'outil pub pour ajouter dépendances au projet et utilisez des packages tiers. Nous avons vérifié le multiple comment structurer une bibliothèque et comment elle compose un package Dart. De plus, nous avons appris comment décrire correctement un package dans le fichier pubspec pour créer des packages partageables. Enfin, nous avons examiné la programmation asynchrone à l'aide de futurs et d'isolats. Aussi, nous avons appris sur les installations de test unitaire Dart pour écrire un meilleur code.

Dans le chapitre suivant, nous commencerons à comprendre et à travailler avec le framework Flutter. De plus, vous continuerez avec les connaissances Dart que vous avez acquises jusqu'à présent.

[93]

Psaumes 112

3

Une introduction à Flutter

Dans ce chapitre, vous apprendrez l'histoire du framework Flutter, comment et pourquoi il était créé, et son évolution jusqu'à présent. Vous apprendrez comment sa communauté y contribue, et comment et pourquoi il s'est développé rapidement ces derniers mois. Vous serez présenté à la principales caractéristiques de Flutter, avec de courtes comparaisons avec d'autres frameworks. Aussi, vous verrez comment créer un projet Flutter de base. Pour ce faire, nous aurons besoin d'une machine appropriée configuré avec Flutter et ses différents prérequis.

Suivez les instructions de configuration de l'environnement du framework Flutter
ici: <https://flutter.dev/docs/get-started/install/macos>

Les sujets suivants seront traités dans ce chapitre:

- Comparaisons avec d'autres frameworks de développement d'applications mobiles
- Compilation de flutter
- Rendu flottant
- Présentation des widgets
- Structure de projet de base Flutter

Épisode 113

Une introduction à Flutter

chapitre 3

Comparaisons avec d'autres applications mobiles cadres de développement

Bien qu'il soit relativement nouveau, Flutter a connu beaucoup d'expérimentation et évolution au fil des ans. Il s'appelait Sky, lors de sa première apparition chez *Dart Developer Summit 2015* présenté par *Eric Seidel*. Il a été présenté comme l'évolution de certains Expérimentations Google pour créer quelque chose de mieux pour mobile en termes de développement et d'utilisateur expérience, avec pour objectif principal un rendu de haute performance. Présenté comme *Flutter* en 2016, et avec sa première version alpha en mai 2017, il se construisait déjà pour Systèmes iOS et Android. Ensuite, il a commencé à mûrir et l'adoption communautaire a commencé grandir. Il a évolué depuis les commentaires de la communauté vers sa première version stable fin 2018.

Il existe de nombreux frameworks de développement mobile qui visent un objectif commun: créer des applications mobiles natives pour Android et iOS avec une base de code unique. Certains de ces cadres sont largement adoptés par la communauté et fournissent des solutions similaires problèmes qu'ils prétendent résoudre. Sachant cela, nous pourrions demander ce qui suit:

- Pourquoi Flutter a-t-il été créé?
- En avons-nous vraiment besoin?
- En quoi est-ce mieux que les cadres concurrents?

Voyons comment fonctionne Flutter et répondons à certaines de ces questions avant d'obtenir notre les mains dessus.

Les problèmes que Flutter veut résoudre

Depuis le début du framework Flutter, il était destiné à offrir une meilleure expérience à l'utilisateur grâce à une exécution haute performance, mais ce n'est pas la seule promesse de Flutter. L'expérience du développement a également été axée sur la résolution de certains des problèmes de développement mobile multi-plateforme:

Cycles de développement longs / plus chers : pour pouvoir faire face au marché demande, vous devez choisir de créer pour une seule plate-forme ou de créer plusieurs équipes. Cela a des conséquences en termes de coût, de délais multiples, et différentes capacités des frameworks natifs.

[95]

Psaumes 114

Une introduction à Flutter

chapitre 3

Plusieurs langues à apprendre : si un développeur souhaite développer pour plusieurs plates-formes, ils doivent apprendre à faire quelque chose dans un système d'exploitation et à programmer une autre langue, et plus tard, la même chose sur un autre OS et langage de programmation.

Cela a certainement un impact sur le temps de production du développeur.

Temps de construction / compilation long : certains développeurs ont peut-être déjà expérimenté comment le temps de construction peut avoir un impact sur la productivité. Sous Android, par exemple, certains développeurs connaissent plusieurs longs temps de construction après quelques minutes de codage (ceci évolue, et c'est beaucoup mieux maintenant, mais cela causait déjà beaucoup de douleur).

Effets secondaires des solutions multiplateformes existantes : vous adoptez une framework de plateforme (c'est-à-dire, React Native, Xamarin, Ionic, Cordova) dans un essayer de contourner les problèmes précédents, mais avec cela vient un certain côté des effets, tels qu'un impact sur les performances, un impact sur la conception ou une expérience utilisateur impact.

Voyons maintenant comment Flutter résout ces problèmes.

Différences entre les cadres existants

Il existe un grand nombre de cadres et de technologies de haute qualité et bien acceptés.

Certains d'entre eux sont les suivants:

- Xamarin
- Réagir natif
- Ionique
- Cordoue

Ainsi, vous pourriez penser qu'il est difficile pour un nouveau framework de trouver sa place sur un terrain déjà plein, mais ce n'est pas. Flutter a des avantages qui se font de la place, pas nécessairement en surmontant le d'autres frameworks, mais en étant déjà au moins au même niveau que les frameworks natifs:

- Haute performance
- Contrôle total de l'interface utilisateur
- Langue de fléchettes
- Être soutenu par Google
- Framework open source
- Ressources et outils pour les développeurs

Examinons chacun de ces éléments plus en détail.

[96]

Épisode 115

Une introduction à Flutter

chapitre 3

Haute performance

À l'heure actuelle, il est difficile de dire que les performances de Flutter sont toujours meilleures que toutes les autres cadres dans la pratique, mais il est sûr de dire que son objectif est d'être. Par exemple, son rendu La couche a été développée avec une fréquence d'images élevée à l'esprit. Comme nous le verrons dans le *Flutter* section de *rendu*, certains des frameworks existants reposent sur le rendu JavaScript et HTML, ce qui peut entraîner une surcharge de performances car tout est dessiné dans une vue Web (un

composant visuel comme un navigateur Web). Certains utilisent l' **équipement d'origine Widgets du fabricant (OEM)** mais reposent sur un pont pour demander à l'API du système d'exploitation de rendre le composants, ce qui crée un goulot d'étranglement dans l'application car elle nécessite une étape supplémentaire pour rendre l' **interface utilisateur (UI)**.

Voir la section "*Rendu Flutter*" pour plus de détails sur le rendu Flutter approche par rapport aux autres.

Quelques points qui rendent les performances de Flutter excellentes:

Flutter possède les pixels : Flutter rend l'application pixel par pixel (voir section), interagissant directement avec le moteur graphique **Skia**.

Pas de couches supplémentaires ou d'appels d'API OS supplémentaires: comme Flutter possède le rendu de l'application, il n'a pas besoin d'appels supplémentaires pour utiliser les widgets OEM, donc pas de goulot d'étranglement.

Flutter est compilé en code natif : Flutter utilise le compilateur Dart AOT pour produire du code natif. Cela signifie qu'il n'y a pas de frais généraux lors de la mise en place d'un environnement pour interpréter le code Dart à la volée, et il fonctionne comme une application native, démarrer plus rapidement que les frameworks qui nécessitent une sorte d'interprète.

Contrôle total de l'interface utilisateur

Le framework Flutter choisit de faire toute l'interface utilisateur en rendant le visuel composants directement sur le canevas, comme nous l'avons vu précédemment, ne nécessitant rien de plus que le canevas de la plate-forme, donc ce n'est pas limité par des règles et des conventions. La plupart temps, les frameworks ne font que reproduire ce que propose la plateforme d'une autre manière. Par exemple, d'autres frameworks multiplateformes basés sur la vue Web reproduisent des composants visuels à l'aide Éléments HTML avec style CSS. D'autres frameworks émulent la création du visuel composants et les transmettre à la plate-forme de l'appareil, qui rendra les widgets OEM comme une application développée en natif. Nous ne parlons pas de performances ici, alors que fait d'autre Flutter offre en n'utilisant pas les widgets OEM et en faisant le travail tout seul?

[97]

Épisode 116

Une introduction à Flutter

chapitre 3

Voyons voir:

Régler tous les pixels de l'appareil : les cadres limités par les widgets OEM reproduisent tout au plus ce qu'une application développée native ferait, car ils n'utilisent que le composants disponibles de la plateforme. D'autre part, les frameworks basés sur le web les technologies peuvent reproduire plus que des composants spécifiques à la plate-forme, mais être également limité par le moteur Web mobile disponible sur l'appareil. En obtenant le contrôle du rendu de l'interface utilisateur, Flutter permet au développeur de créer l'interface utilisateur dans leur à votre façon en exposant une API de widgets extensible et riche, qui fournit des outils qui peut être utilisé pour créer une interface utilisateur unique sans inconvénient en termes de performances et sans limites de la conception.

Kits d'interface utilisateur de plate-forme : en n'utilisant pas de widgets OEM, Flutter peut casser la plate-forme conception, mais ce n'est pas le cas. Flutter est équipé de packages qui fournissent une plate-forme widgets de conception, le jeu de matériaux dans Android et **Cupertino** dans iOS.

Nous en verrons plus sur les kits d'interface utilisateur de la plate-forme au [chapitre 4, Widgets: Construction Dispositions dans Flutter](#).

Exigences de conception d'interface utilisateur réalisables : Flutter fournit une API propre et robuste avec la possibilité de reproduire des mises en page fidèles aux exigences de conception. Contrairement aux frameworks basés sur le Web qui reposent sur des règles de mise en page CSS qui peuvent être volumineuses et compliqué et même conflictuel, Flutter simplifie cela en ajoutant des règles sémantiques qui peut être utilisé pour créer des mises en page complexes mais efficaces et belles.

Une apparence et une sensation plus fluides : en plus des kits de widgets natifs, Flutter cherche à

fournir une expérience de plateforme native où l'application s'exécute, donc les polices, les gestes et les interactions sont mis en œuvre d'une manière spécifique à la plateforme, apportant une sensation naturelle de l'utilisateur, comme une application native.

Nous appelons les composants visuels des widgets. C'est aussi la façon dont Flutter les appelle. Nous en discuterons plus à ce sujet dans l'*introduction des widgets* section de ce chapitre.

Maintenant, approfondissons Dart.

[98]

Épisode 117

Une introduction à Flutter

chapitre 3

Dart

Depuis sa création, l'un des principaux objectifs de Flutter était d'être une alternative performante aux cadres multiplateformes existants. Mais pas seulement cela; pour améliorer considérablement le mobile l'expérience du développeur a été l'un des points cruciaux du projet.

Dans cet esprit, Flutter avait besoin d'un langage de programmation qui lui permette d'accomplir ces objectifs, et Dart semble correspondre parfaitement au cadre pour ce qui suit les raisons:

Compilation Dart AOT et JIT : Dart est suffisamment flexible pour fournir différentes méthodes d'exécuter le code, donc Flutter utilise Dart AOT avec les performances à l'esprit lorsque compiler une *version finale* de l'application, et il utilise JIT avec sous-seconde compilation de code au moment du développement, visant des flux de travail et du code rapides changements.

Dart Just in time compilation (JIT) et Ahead of time (AOT) fait référence à lorsque la phase de compilation a lieu. Dans AOT, le code est compilé avant fonctionnement. Dans JIT, le code est compilé lors de l'exécution. (Découvrez la *Dart* section *introduction* dans le premier chapitre).

Haute performance: en raison du support de Dart pour la compilation AOT, Flutter ne nécessite un pont lent entre les royaumes (par exemple, non natif à natif), qui permet aux applications Flutter de démarrer beaucoup plus rapidement. De plus, Flutter utilise un flux de style fonctionnel avec des objets de courte durée, ce qui signifie beaucoup de allocations. Le ramassage des ordures de Dart fonctionne sans verrou, ce qui facilite allocation.

Apprentissage facile: Dart est un langage flexible, robuste, moderne et avancé. Bien qu'il soit encore en évolution, le langage a une orientation bien définie orientée objet framework avec des fonctionnalités familières aux langages dynamiques et statiques, un communauté active et documentation bien structurée.

Interface utilisateur déclarative : dans Flutter, nous utilisons un style **déclaratif** pour mettre en page les widgets, signifie que les widgets sont immuables et ne sont que des «plans» légers. Pour changer l'interface utilisateur, un widget déclenche une reconstruction sur lui-même et sur son sous-arbre. dans le style **impératif** opposé (le plus courant), nous pouvons changer de composant spécifique propriétés après leur création.

[99]

Épisode 118

Une introduction à Flutter

chapitre 3

Remarque : jetez un œil à l'introduction officielle de l'interface utilisateur déclarative de Flutter: <https://flutter.dev/> / Flutter déclaratif

Syntaxe Dart à la mise en page: Différent de nombreux frameworks qui ont un syntaxe de mise en page, dans Flutter, la mise en page est créée en écrivant du code Dart, visant à plus de flexibilité et de facilité pour créer un environnement de développement, avec des outils pour débogage des performances de rendu de disposition, par exemple.

Dart et Flutter sont développés par Google, et c'est important, comme nous le verrons.

Être soutenu par Google

Flutter est un tout nouveau framework, ce qui signifie qu'il n'a pas une grande partie du marché du développement mobile encore, mais cela est en train de changer, et les perspectives pour les prochaines années est très positif.

Soutenu par Google, le framework dispose de tous les outils nécessaires pour réussir communauté, avec le soutien de l'équipe Google, présence à de grands événements comme Google IO, et des investissements dans l'amélioration continue de la base de code. Dès le lancement du troisième version bêta de Google IO 2018 à la première version stable lancée pendant le *Flutter Live Event* fin 2018, sa croissance est évidente:

- Plus de 200 millions d'utilisateurs des applications Flutter.
- Plus de 3000 applications Flutter sur le Play Store.
- Plus de 250 000 nouveaux développeurs.

Le 34^e référentiel de logiciels le plus populaire sur GitHub - il était dans le **Top 15 au début 2019**.

OS Fuchsia et Flutter

Ce n'est plus un secret que Google travaille sur son nouveau système d'exploitation Fuchsia en remplacement de le système d'exploitation Android. Une chose à laquelle il faut faire attention est que Fuchsia OS peut être un Google universel système pour fonctionner sur plus que les téléphones mobiles, et cela affecte directement l'adoption de Flutter. En effet, Flutter sera la première méthode de développement d'applications mobiles pour le nouveau Fuchsia OS, et non seulement cela, l'interface utilisateur du système est en cours de développement avec lui. Avec le système ciblant plus d'appareils que de simples smartphones, comme cela semble être le cas, Flutter ont certainement beaucoup d'améliorations.

[100]

Épisode 119

Une introduction à Flutter

chapitre 3

La croissance de l'adoption du framework est directement liée au nouveau système d'exploitation Fuchsia. Tel quel se rapproche du lancement, il est important pour Google d'avoir des applications mobiles ciblant le nouveau système. Par exemple, Google a annoncé que les applications Android seront compatibles avec le nouveau système d'exploitation, facilitant considérablement la transition et l'adoption de Flutter.

Framework open source

Avoir une grande entreprise comme Google derrière elle est fondamental pour un cadre tel que Flutter (voir React, par exemple, qui est maintenu par Facebook). De plus, la communauté le soutien devient encore plus important à mesure qu'il devient plus populaire.

En étant open source, la communauté et Google peuvent travailler ensemble pour:

- Aide à la résolution de bogues et à la documentation via la collaboration de code
- Créer un nouveau contenu éducatif sur le cadre
- Documentation de support et utilisation
- Prendre des décisions d'amélioration basées sur de vrais retours

L'amélioration de l'expérience des développeurs est l'un des principaux objectifs du framework. Donc, en plus d'être proche de la communauté, le framework fournit d'excellents outils et ressources pour les développeurs. Voyons-les.

Ressources et outils pour les développeurs

L'accent mis sur les développeurs dans le framework Flutter va de la documentation et de l'apprentissage ressources pour fournir des outils pour aider à la productivité:

Documentation et ressources d'apprentissage : les sites Web Flutter sont riches pour les développeurs provenant d'autres plates-formes, y compris de nombreux exemples et cas d'utilisation pour exemple. les fameux Google Codelabs (<https://.. / codelabs develop> com/ ?

[101]

Épisode 120

Une introduction à Flutter

chapitre 3

Outils de ligne de commande et intégration IDE : outils Dart qui aident à analyser, l'exécution et la gestion des dépendances font également partie de Flutter. Par ailleurs, Flutter a également des commandes pour aider au débogage, au déploiement et à l'inspection de la mise en page rendu et intégration avec les IDE via des plugins Dart. Voici une liste des diverses commandes:

[102]

Épisode 121

Une introduction à Flutter

chapitre 3

Démarrage facile : Flutter est livré avec l'outil Flutter **Doctor**, qui est une ligne de commande outil qui guide le développeur dans la configuration du système en indiquant ce qui est nécessaire pour être prêt à configurer un environnement Flutter. C'est à quoi ça ressemble comme:

La commande Flutter Doctor identifie également les appareils connectés et si il y a des mises à jour disponibles, comme vous pouvez le voir.

Rechargement à chaud : c'est la fonctionnalité qui a été au centre des présentations sur le cadre. En combinant les capacités du langage Dart (comme Compilation JIT) et la puissance de Flutter, il est possible pour le développeur de voir instantanément les modifications de conception apportées au code dans le simulateur ou l'appareil. Dans Flutter, il n'y a pas d'outil spécifique pour l'aperçu de la mise en page. Le rechargement à chaud le rend inutile.

Maintenant que nous avons appris les avantages de Flutters, commençons à regarder les logiciels compilations.

Compilation Flutter (Dart)

La façon dont une application est construite est fondamentale pour la façon dont elle fonctionnera sur la cible Plate-forme. Il s'agit d'une étape importante concernant les performances. Même si tu ne le fais pas besoin nécessairement de le savoir pour chaque type d'application, sachant comment l'application est built vous aide à comprendre et à mesurer les améliorations possibles.

[103]

Épisode 122

Comme nous l'avons déjà souligné, Flutter s'appuie sur la compilation AOT de Dart pour la sortie mode et la compilation JIT de Dart pour le mode développement / débogage. Dart est l'un des très quelques langages capables d'être compilés à la fois en AOT et JIT, et pour Flutter, c'est génial.

Compilation de développement

Pendant le développement, Flutter utilise la compilation JIT en mode développement. Cela permet des fonctionnalités de développement importantes telles que le chargement à chaud, la fonctionnalité mentionnée ci-dessus section. En raison de la puissance du compilateur de Dart, les interactions entre le code et le simulateur / l'appareil sont très rapides et les informations de débogage aident les développeurs à entrer dans la source code.

Compilation de publication

En mode version, les informations de débogage ne sont pas nécessaires et l'accent est mis sur les performances. Flutter utilise une technique commune aux moteurs de jeu. En utilisant le mode AOT, le code Dart est compilé en code natif, et l'application charge la bibliothèque Flutter et délègue le rendu, les entrées et la gestion des événements à l'aide du moteur Skia.

Plateformes prises en charge

À présent, Flutter prend en charge les appareils Android ARM fonctionnant au moins sur la version Jelly Bean 4.1.x, et les appareils iOS de l'iPhone 4S ou plus récent. Bien sûr, les applications Flutter peuvent normalement être exécutées sur simulateurs.

Google a l'intention de porter le moteur d'exécution Flutter sur le Web en utilisant la fonctionnalité Dart de compilation en JavaScript. Initialement appelé **Hummingbird**, ce projet est maintenant connu sous le nom de "Flutter pour le Web".

Nous n'allons pas entrer plus en détail sur les aspects de compilation de Flutter car ils dépassent le cadre de ce livre. Pour plus d'informations, vous pouvez lire <https://flutter.dev/>

[Exécutez](#)
[FAQ #1](#)

[104]

Épisode 123

Rendu flottant

L'un des principaux aspects qui rend Flutter unique est la façon dont il dessine le visuel composants à l'écran. La grande différence réside dans la manière dont l'application communique avec le SDK de la plateforme, ce qu'il demande au SDK de faire et ce qu'il fait par lui-même:

Le SDK de la plate-forme peut être considéré comme l'interface entre les applications et l'opération système et services. Chaque système fournit son propre SDK avec ses propres capacités et est basé sur un langage de programmation (c'est-à-dire Kotlin / Java pour le SDK Android et Swift / Objective C pour le SDK iOS). Nous avons examiné certaines approches de rendu utilisées par différents cadres auparavant; examinons-les maintenant plus en détail.

[105]

Épisode 124

Une introduction à Flutter

chapitre 3

Technologies basées sur le Web

Nous avons déjà vu des frameworks qui utilisent des webviews pour reproduire une interface utilisateur en combinant HTML et CSS. En termes d'utilisation de la plate-forme, cela ressemblerait à ceci:

L'application ne sait pas comment le rendu est effectué par la plateforme; la seule chose c'est besoins est le widget de visualisation Web sur lequel il rendra le code HTML et CSS.

Outre la partie rendu, il y a un petit point à remarquer, à savoir accéder aux API du système, le code JavaScript a besoin d'un pont pour appeler le code natif, provoquant une légère surcharge de performances.

[106]

Épisode 125

Une introduction à Flutter

chapitre 3

Framework et widgets OEM

Une autre façon de rendre les widgets consiste à ajouter une couche au-dessus des widgets de la plate-forme, mais pas changer la façon dont le système rend efficacement les composants visuels:

Dans ce mode de rendu, le travail est effectué par le SDK comme une application native normale, mais avant it, la mise en page est définie par une étape supplémentaire dans le langage du framework. Chaque changement dans l'interface utilisateur provoque la communication entre le code de l'application et le code natif qui chargé d'appeler le SDK de la plateforme, travaillant comme un intermédiaire. Comme le précédent technique, cela peut entraîner une légère surcharge pour l'application, peut-être un peu plus grande que le précédent, car le rendu se produit souvent, et donc le la communication.

[107]

Épisode 126

Une introduction à Flutter

chapitre 3

Flutter - rendu par lui-même

Flutter choisit de faire tout le travail dur par lui-même. La seule chose dont il a besoin de la plate-forme

Le SDK est un accès aux API de services et à un canevas sur lequel dessiner l'interface utilisateur:

Flutter déplace les widgets et le rendu vers l'application, d'où il obtient la personnalisation et extensibilité. À travers un canevas, il peut dessiner n'importe quoi et aussi accéder aux événements à gérer entrées et gestes de l'utilisateur par lui-même. Le pont à Flutter se fait par plateforme canaux, que nous verrons plus en détail dans [Chapitre 13, Amélioration de l'expérience utilisateur](#).

Présentation des widgets

Comprendre les widgets Flutter est essentiel si vous souhaitez travailler avec. Tu connais Flutter prend le contrôle du rendu et le fait en gardant à l'esprit l'extensibilité et la personnalisation, l'intention d'ajouter du pouvoir aux mains du développeur. Voyons comment Flutter applique les widgets idée dans le développement d'applications pour créer des interfaces utilisateur impressionnantes.

Les widgets peuvent être compris comme la représentation visuelle (mais pas seulement) de parties du application. De nombreux widgets sont assemblés pour composer l'interface utilisateur d'une application. Imagine-le comme un puzzle dans lequel vous définissez les pièces.

[108]

Épisode 127

Une introduction à Flutter

chapitre 3

L'intention des widgets est de fournir un moyen pour votre application d'être modulaire, évolutive et expressif avec moins de code et sans imposer de limitations. Les principales caractéristiques du Les widgets UI dans Flutter sont la composabilité et l'immuabilité.

Composabilité

Flutter choisit la composition plutôt que l'héritage, dans le but de garder chaque widget simple et avec un objectif bien défini. Atteindre l'un des objectifs du framework, flexibilité, Flutter permet au développeur de faire de nombreuses combinaisons pour obtenir des résultats incroyables.

Immutabilité

Flutter est basé sur le style réactif de la programmation, où les instances de widget sont courtes vécu et changer leurs descriptions (visuellement ou non) en fonction de la configuration changements, donc il réagit aux changements et propage ces changements à ses widgets de composition, etc.

Un widget Flutter peut avoir un état associé, et lorsque l'état associé modifications, il peut être reconstruit pour correspondre à la représentation.

Les termes *état* et *réactif* sont bien connus dans le style de programmation React,

diffusé par la célèbre bibliothèque React de Facebook.

Tout est un widget

Les widgets Flutter sont partout dans une application. Peut-être que tout n'est pas un widget, mais presque tout est. Même l'application est un widget dans Flutter, et c'est pourquoi ce concept est si important. Un widget représente une partie d'une interface utilisateur, mais cela ne signifie pas que c'est seulement quelque chose qui est visible. Il peut s'agir de l'un des éléments suivants:

- Un élément visuel / structurel qui est un élément structurel de base, tel que le bouton ou widgets de texte
- Un élément spécifique à la mise en page qui peut définir la position, les marges ou le remplissage, comme comme widget Padding
- Un élément de style qui peut aider à coloriser et à thématiser un élément visuel / structurel, comme le widget Thème
- Un élément d'interaction qui permet de répondre aux interactions de différentes manières, comme le widget GestureDetector

[109]

Épisode 128

Une introduction à Flutter

chapitre 3

Nous allons vérifier des exemples d'utilisation de ces widgets dans ce qui suit chapitre.

Les widgets sont les éléments de base d'une interface. Pour créer correctement une interface utilisateur, Flutter organise les widgets dans une arborescence de widgets.

L'arborescence des widgets

C'est un autre concept important dans les mises en page Flutter. C'est là que les widgets prennent vie. le L'arborescence des widgets est la représentation logique de tous les widgets de l'interface utilisateur. Il est calculé pendant *mise en page* (mesures et informations structurelles) et utilisées lors du *rendu* (image à écran) et *test de frappe* (interactions tactiles), et c'est ce que Flutter fait le mieux. En utilisant beaucoup de algorithmes d'optimisation, il essaie de manipuler le moins possible l'arbre, en réduisant volume total de travail consacré au rendu, visant une plus grande efficacité:

[110]

Épisode 129

*Une introduction à Flutter**chapitre 3*

Les widgets sont représentés dans l'arborescence sous forme de nœuds. Il peut être associé à un état; chaque le changement de son état entraîne la reconstruction du widget et de l'enfant impliqué.

Comme vous pouvez le voir, la structure enfant de l'arborescence n'est pas statique et elle est définie par les widgets ' la description. Les relations enfants dans les widgets sont ce qui fait l'arborescence de l'interface utilisateur; il existe par composition, il est donc courant de voir les widgets intégrés de Flutter exposant des enfants ou propriétés des enfants, en fonction de l'objectif du widget.

L'arborescence des widgets ne fonctionne pas seule dans le framework. Il a l'aide de l'arbre des éléments; une arbre qui se rapporte à l'arborescence des widgets en représentant le widget construit à l'écran, de sorte que chaque widget aura un élément correspondant dans l'arborescence des éléments après sa construction.

L'arbre des éléments a une tâche importante dans Flutter. Il permet de mapper les éléments à l'écran sur le arborescence de widgets. En outre, il détermine comment la reconstruction des widgets est effectuée dans les scénarios de mise à jour. Lorsqu'un le widget change et doit être reconstruit, cela entraînera une mise à jour du élément. L'élément stocke le type du widget correspondant et une référence à son éléments enfants. Dans le cas du repositionnement, par exemple, d'un widget, l'élément sera vérifiez le type du nouveau widget correspondant, et dans une correspondance, il se mettra à jour avec la nouvelle description du widget.

L'arborescence des éléments peut être vue comme une arborescence auxiliaire de prendre au widget arbre. Si vous avez besoin de plus d'informations à ce sujet, vous pouvez consulter le site officiel docs: <https://https://. /docs>

Bonjour Flutter

Il est temps de commencer à se salir les mains avec du code. Avec le développement Flutter environnement configuré, nous pouvons commencer à utiliser les commandes Flutter. La façon typique de démarrer un Le projet Flutter consiste à exécuter la commande suivante:

```
flutter create <output_directory>
```

Ici, output_directory sera également le nom du projet Flutter si vous ne le spécifiez pas comme argument.

[111]

Épisode 130

*Une introduction à Flutter**chapitre 3*

En exécutant la commande précédente, le dossier avec le nom fourni sera généré avec un exemple de projet Flutter. Nous analyserons le projet dans quelques instants. Tout d'abord, c'est bon à savoir qu'il existe des options utiles pour manipuler le projet résultant de la commande flutter create. Les principaux sont les suivants:

--org: Ceci peut être utilisé pour changer l'organisation du propriétaire du projet. Si vous connaissez déjà le développement Android ou iOS, c'est le nom de domaine inversé, et est utilisé pour identifier les noms de packages sur Android et comme préfixe dans le bundle iOS

identifiant. La valeur par défaut est com.example.

-s, - sample: la plupart des exemples officiels d'utilisation des widgets ont un identifiant unique que vous pouvez utiliser pour cloner rapidement l'exemple sur votre machine avec ce argument.

Chaque fois que vous explorez le site Web Flutter docs (<https://docs.flutter.dev/>

[docs.flut](#) extraire un échantillon et l'utiliser
with cet

-i, --ios-language et -a, --android-language: Ils sont utilisés pour spécifier la langue du code natif du projet et ne sont utilisés que si vous prévoyez d'écrire du code de plateforme natif. Dans [Chapitre 13](#), *Amélioration de l'expérience utilisateur*, nous verrons comment ajouter du code natif au projet.

--project-name: utilisez ceci pour changer le nom du projet. Ce doit être un Dart valide identifiant de package, comme nous l'avons déjà vu sur la description du format pubspec la page (<https://flutter.dev/docs/development/packages-and-plugins/using-packages>).

"Les noms de package doivent être tous en minuscules, avec des traits de soulignement pour séparer mots, 'juste_comme_ceci'. N'utilisez que des lettres latines de base et des chiffres arabes: [a-zA-Z0-9]. Assurez-vous également que le nom est un identifiant Dart valide - qu'il ne le fait pas commencer par des chiffres et n'est pas un mot réservé."

Si vous ne spécifiez pas ce paramètre, il essaie d'utiliser le même nom comme sortie annuaire. Notez que cet argument doit être le dernier de la liste des arguments à condition de.

[112]

Épisode 131

Une introduction à Flutter

chapitre 3

Voyons une structure de projet Flutter typique créée avec la commande précédente, flutter créer hello_world:

Si vous pensez que cela ressemble aux packages Dart, vous avez peut-être raison. Battement Les projets sont une sorte de package Dart, avec bien sûr quelques particularités. Liste des basiques éléments de structure, nous obtenons ce qui suit:

android / ios: contient les codes spécifiques à la plate-forme. Si vous connaissez déjà le Structure de projet Android d'Android Studio, il n'y a pas de surprise ici. le il en va de même pour les projets XCode iOS.

hello_flutter.iml: il s'agit d'un fichier de projet IntelliJ typique, qui contient

les informations JAVA_MODULE utilisées par l'EDI.
 répertoire lib: il s'agit du dossier principal d'une application Flutter; le générée
 Le projet doit contenir au moins un fichier main.dart sur lequel commencer à travailler. Nous serons
 vérifier ce fichier en détail en quelques étapes.

pubspec.yaml et pubspec.lock: si vous vous en souvenez [Chapitre 2](#), *Intermédiaire*
Programmation Dart, ce fichier pubspec.yaml est ce qui définit un package Dart. C'est
 ce qui se passe ici, et c'est l'un des principaux fichiers du projet, où vous
 listez les dépendances de l'application et dans le cas de Flutter plus que cela. Nous serons
 en regardant cela plus en détail dans [Chapitre 4](#), *Widgets: création de mises en page dans Flutter*.

[113]

Épisode 132

Une introduction à Flutter

chapitre 3

README.md: Ce fichier a généralement une description du projet, et il est très
 commun dans les projets open source.

répertoire de test: il contient tous les fichiers liés aux tests du projet. Ici, nous pouvons
 ajoutez des tests unitaires, comme nous l'avons vu auparavant, et aussi des tests de widgets en utilisant Flutter-
 packages spécifiques.

Dans la plupart de ce livre, nous utilisons des outils de ligne de commande directement depuis le
 Terminal. De plus, à titre informatif, l'EDI utilisé est Visual Studio
 Code. N'oubliez pas que les IDE utilisent ces outils en coulisses pour interagir
 avec le projet.

fichier pubspec

Le fichier pubspec de Flutter est similaire à un simple package Dart. En plus de cela, il contient un
 section supplémentaire pour les configurations spécifiques à Flutter. Voyons le fichier pubspec.yaml
 contenu en détails:

```
nom: hello_flutter
description: Un nouveau projet Flutter.
version: 1.0.0 + 1
```

Le début du fichier est simple. Comme nous le savons déjà, la propriété name est définie lorsque
 nous exécutons la commande pub create, suivie de la description du projet par défaut.

Vous pouvez spécifier la description lors de la commande de création de flutter
 en utilisant l'argument --description.

La propriété version suit les conventions du package Dart: le numéro de version, plus un
 numéro de version de build facultatif séparé par +. En plus de cela, Flutter vous permet de
 remplacer ces valeurs lors de la construction. Nous examinerons cela plus en détail dans le [chapitre](#)
[12](#), *Test, débogage et déploiement*, dans la section *Déploiement d'application*.

Ensuite, nous avons la section des dépendances du fichier pubspec:

```
environnement:
  sdk: ">= 2.0.0-dev.68.0 <3.0.0"

dépendances:
  battement:
    sdk: scintillement
```

[114]

Épisode 133

Une introduction à Flutter

chapitre 3

```
# Ce qui suit ajoute la police Cupertino Icons à votre application.  
# À utiliser avec la classe CupertinoIcons pour les icônes de style iOS.  
cupertino_icons: ^ 0.1.2
```

```
dev_dependencies:  
  flutter_test:  
    sdk: scintillement
```

Maintenant, jetez un œil à l'explication du code précédent:

Nous commençons par la propriété d'environnement avec les contraintes de version du SDK Dart défini. Vous êtes d'accord pour utiliser la version fournie par l'outil car elle est suivie par les mises à jour du SDK Flutter.

Le SDK Dart est intégré au SDK Flutter, vous n'avez donc pas à installez-les séparément.

Ensuite, nous avons la propriété dependencies, qui commence par le main dépendance d'une application Flutter, le SDK Flutter lui-même, qui contient de nombreux des packages principaux de Flutter.

En tant que dépendance supplémentaire, le générateur ajoute les cupertino_icons package, qui contient des éléments d'icône utilisés par le Flutter Cupertino intégré widgets (il y en a plus à ce sujet dans le chapitre suivant).

La propriété dev_dependencies contient uniquement le package flutter_test dépendance fournie par le SDK Flutter lui-même, et contient Flutter-specific extensions du package de test Dart déjà connu.

Dans le dernier bloc du fichier, il y a une section de flottement dédiée:

```
battement:  
  
utilise-matière-conception: vrai  
  
# Pour ajouter des actifs à votre application, ajoutez une section d'actifs, comme ceci:  
# les atouts:  
# - images / a_dot_burr.jpeg  
# - images / a_dot_ham.jpeg  
# ...  
# Pour ajouter des polices personnalisées à votre application, ajoutez une section de polices ici,  
# polices:  
# - famille: Schyler  
# polices:  
# - actif: fonts / Schyler-Regular.ttf
```

[115]

Épisode 134

Une introduction à Flutter

chapitre 3

```
# - actif: fonts / Schyler-Italic.ttf  
#           style: italique  
#
```

Cette section flottante nous permet de configurer les ressources qui sont regroupées dans l'application à utiliser pendant l'exécution, comme des images, des polices et un fichier JSON, généralement, tout non fichier de code source qui aide à la composition de l'application:

uses-material-design: Nous verrons les widgets Material fournis par Flutter dans le chapitre suivant. En plus d'eux, nous pouvons également utiliser la conception matérielle Icônes (<https://?/io>) format de police pe: (définissez-le sur true) pour que les icônes soient incluses dans l'application.

assets: cette propriété est utilisée pour lister les chemins de ressources qui seront regroupés avec

la demande finale. Consultez le code suivant pour plus de détails sur son utilisation.
 Les fichiers d'actifs peuvent être organisés de n'importe quelle manière; ce qui compte pour Flutter est le chemin des fichiers. Vous spécifiez le chemin du fichier par rapport à la racine du projet. Ce est utilisé plus tard dans le code Dart lorsque vous devez faire référence à un fichier d'actif. Voici un exemple:

les atouts:
 - images / home_background.jpeg

Pour ajouter une image à utiliser plus tard, nous ajoutons le chemin dans la liste des actifs, ou si nous voulons ajouter tous les fichiers dans le répertoire, nous spécifions simplement le chemin du répertoire:

les atouts:
 - images/

Cela inclut tous les fichiers à l'intérieur du répertoire. Notez le caractère / à la fin.

polices: Cette propriété nous permet d'ajouter des polices personnalisées à l'application. Il y a plus à ce sujet au [chapitre 6, Thème et style](#), dans la section *Polices personnalisées*.

Nous vérifierons comment charger les différents actifs au cours du livre chaque fois que nous en avons besoin. En outre, vous pouvez en savoir plus sur la spécification des actifs détails sur le site Web Flutter docs: <https://flutter.io/docs/>

[développement / ui / i](#)

[116]

Épisode 135

Une introduction à Flutter

chapitre 3

Exécution du projet généré

Le projet généré utilise le modèle Flutter par défaut pour créer le projet. Ce L'application dispose d'un compteur pour démontrer le style de programmation React dans Flutter. Nous allons vérifier les détails dans le chapitre suivant, lorsque nous parlons des différents widgets qui nous pouvons utiliser pour composer notre application. Dans l'exemple hello_flutter que nous avons créé précédemment en utilisant la commande flutter create, MyApp est le widget racine de l'application.

fichier lib / main.dart

Le fichier principal du projet généré est le point d'entrée de l'application Flutter:

```
void main () => runApp (MyApp ());
```

La fonction principale en elle-même est le point d'entrée Dart d'une application. Qu'est-ce qui rend le Flutter application take the scene est la fonction runApp appelée en passant un widget en paramètre, qui sera le widget racine de l'application (l'application elle-même).

Flutter run

Pour exécuter une application Flutter, nous devons avoir un appareil ou un simulateur connecté. Le cheque se fait en utilisant les outils déjà connus de flutter doctor et de flutter émulateurs. le La commande suivante vous permet de connaître les émulateurs Android et iOS existants qui peuvent être utilisé pour exécuter le projet:

émulateurs de flutter

Vous obtiendrez quelque chose de similaire à la capture d'écran suivante:

[117]

Épisode 136

Une introduction à Flutter

chapitre 3

Vous pouvez vérifier comment exécuter vos émulateurs Android sur <https://developer.android.com/studio/run/index.html>
simulateurs, voir plus d'informations sur le site Web de documentation Apple (https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/iOS_Swift_Guide/)
[GettingStartedwithiOS](#),

Après avoir affirmé que nous avons un appareil connecté qui peut exécuter l'application, nous pouvons utiliser la commande suivante:

flutter run

Reportez-vous à la capture d'écran suivante:

[118]

Épisode 137

Une introduction à Flutter

chapitre 3

Cette commande démarre le débogueur et rend la fonctionnalité de rechargement à chaud disponible, lorsque vous peut voir. La première exécution de l'application peut prendre un peu plus de temps que la suite exécutions:

[119]

Épisode 138

Une introduction à Flutter

chapitre 3

L'application est opérationnelle; vous pouvez voir une marque de débogage dans le coin supérieur droit. Cette signifie que ce n'est pas une version en cours d'exécution, comme vous le savez déjà; c'est le développement version de l'application, avec des fonctions de rechargement à chaud et de débogage.

L'exemple précédent a été exécuté sur un simulateur iPhone 6s. Le même résultat serait obtenu en utilisant un émulateur Android, ou un **Android appareil virtuel (AVD)**.

Sommaire

Dans ce chapitre, nous avons enfin commencé à jouer avec le framework Flutter. Tout d'abord, nous avons appris quelques concepts importants sur Flutter, principalement les concepts de widgets. On a vu ça les widgets sont la partie centrale du monde Flutter, où l'équipe Flutter travaille en permanence pour améliorer les widgets existants et en ajouter de nouveaux. C'est parce que le concept de widget est partout, des performances de rendu au résultat final à l'écran.

Nous avons également vu comment démarrer un projet d'application Flutter avec les outils du framework, les

la structure des fichiers du projet et les particularités du fichier pubspec. À la fin, nous avons vu comment pour exécuter un projet sur un émulateur.

Vous pouvez trouver le code source de ce chapitre sur GitHub.

Dans le chapitre suivant, nous approfondirons les types de widgets, tels que avec état et apatrides, et comment et quand ils peuvent être utilisés. En outre, nous en apprendrons davantage sur les widgets et démarrer un projet d'application Flutter que nous suivrons pour le reste du livre, en quel nous appliquerons cumulativement les connaissances acquises dans chaque chapitre.

[120]

Épisode 139

2

Section 2: L'utilisateur Flutter Interface - Tout est un Widget

Dans cette section, vous découvrirez la manière Flutter de travailler avec l'interface utilisateur, la saisie des données utilisateur, et les ressources disponibles pour créer des interfaces utilisateur riches.

Dans cette section, nous couvrirons les chapitres suivants:

[Chapitre 4, Widgets: création de dispositions dans Flutter](#)

[Chapitre 5, Gestion des entrées et des gestes de l'utilisateur](#)

[Chapitre 6, Thème et Styling](#)

[Chapitre 7, Routage: navigation entre les écrans](#)

Épisode 140

4 Widgets: création de dispositions dans Battement

Dans ce chapitre, vous découvrirez les concepts centraux des widgets, les différences entre les widgets sans état et avec état, les widgets les plus courants dans Flutter et comment les ajouter à votre application et comment créer des interfaces complètes à partir de widgets intégrés ou des widgets personnalisés développés par vous-même.

Les sujets suivants seront traités dans ce chapitre:

- Widgets avec état / sans état
- Widgets intégrés
- Comprendre les widgets de mise en page intégrés
- Créer des widgets personnalisés

Widgets avec état ou sans état

De [Chapitre 3](#), *Une introduction à Flutter*, nous avons vu que les widgets jouent un rôle important rôle dans le développement d'applications Flutter. Ce sont les éléments qui forment l'interface utilisateur; Ils sont les représentation de code de ce qui est visible pour l'utilisateur.

Les interfaces utilisateur ne sont presque jamais statiques; ils changent fréquemment, comme vous le savez. Bien qu'immuable par définition, les widgets ne sont pas censés être définitifs - après tout, nous avons affaire à une interface utilisateur et une interface utilisateur changera certainement au cours du cycle de vie de toute application. C'est pourquoi Flutter nous fournit avec deux types de widgets: sans état et avec état.

La grande différence entre ces derniers réside dans la manière dont le widget est *construit*. C'est le développeur responsabilité de choisir le type de widget à utiliser dans chaque situation pour composer l'interface utilisateur afin de tirer le meilleur parti de la puissance de la couche de rendu de widget de Flutter.

Épisode 141

Widgets: création de dispositions dans Flutter

Chapitre 4

Flutter a également le concept de widgets **hérités** (le type `InheritedWidget`), qui est aussi une sorte de widget mais qui est un peu peu différent des deux autres types que nous avons mentionnés. Nous allons Vérifiez-le après avoir exploré le `hello_flutter`

exemple du [chapitre 3](#), *Une introduction à Flutter*, en détail.

Widgets sans état

Une interface utilisateur typique sera composée de nombreux widgets, et certains d'entre eux ne changeront jamais leur properties après avoir été instancié. Ils n'ont pas d'*État*; c'est-à-dire qu'ils ne changent pas en eux-mêmes par une action ou un comportement interne. Au lieu de cela, ils sont modifiés par des événements sur les widgets parents dans l'arborescence des widgets. Donc, il est prudent de dire que les widgets sans état donnent contrôle de la façon dont ils sont construits sur un widget parent dans l'arborescence. Ce qui suit est une représentation d'un widget sans état:

Ainsi, le widget enfant recevra sa description du widget parent et ne changera pas tout seul. En termes de code, cela signifie que les widgets sans état n'ont que des propriétés finales définies lors de la construction, et c'est la seule chose qui doit être construite sur l'appareil écran.

Nous explorerons le code source en détail dans un instant, lorsque nous prendrons le projet Flutter généré par défaut à partir de l'outil de création Flutter utilisé dans le chapitre précédent.

[123]

Épisode 142

Widgets: création de dispositions dans Flutter

Chapitre 4

Widgets avec état

Contrairement aux widgets sans état, qui reçoivent une description de leurs parents qui persiste pendant la durée de vie des widgets, les widgets avec état sont destinés à modifier leurs descriptions de manière dynamique au cours de leur vie. Par définition, les widgets avec état sont également immuables, mais ils ont une *company State Classe* qui représente l'état actuel du widget. Il est montré dans le schéma suivant:

En conservant l'état du widget dans un objet State séparé, le framework peut le reconstruire chaque fois que nécessaire sans perdre son état associé actuel. L'élément dans les éléments tree contient une référence du widget correspondant ainsi que l'objet State associé avec ça. L'objet State avertira lorsque le widget doit être reconstruit et provoquera une mise à jour dans l'arborescence des éléments également.

Widgets avec état et sans état dans le code

Dans le chapitre précédent, nous avons généré un projet Flutter en utilisant la commande suivante:

flutter créer

[124]

Épisode 143

Widgets: création de dispositions dans Flutter

Chapitre 4

Ce projet a été créé avec les arguments par défaut du modèle Flutter par défaut, donnant une petite application avec un compteur qui montre le nombre de fois (+) le bouton a été touché:

L'application de démonstration Flutter de la capture d'écran précédente est utile pour afficher les deux types de widgets dans la pratique.

[125]

Épisode 144

Widgets: création de dispositions dans Flutter

Chapitre 4

Widget sans état dans le code

Commençons par examiner les widgets sans état dans le code. Le tout premier widget sans état du application est la classe d'application elle-même:

```
class MyApp étend StatelessWidget {
  @passer outre
  Construction du widget (contexte BuildContext) {
    retour MaterialApp (
      titre: 'Flutter Demo',
      thème: ThemeData (
        primarySwatch: Colors.blue,
      ),
      accueil: MyHomePage (titre: 'Flutter Demo Home Page'),
    );
  }
}
```

Comme vous pouvez le voir, la classe MyApp étend StatelessWidget et remplace la méthode build (BuildContext). Cette méthode décrit une partie de l'interface utilisateur; c'est-à-dire qu'il construit le sous - arborescence de widgets en dessous . Dans le cas précédent, MyApp est la racine de l'arborescence des widgets et, par conséquent, il construit tous les widgets dans l'arborescence. Dans ce cas, son enfant direct est MaterialApp. Selon la documentation, cela est défini comme suit:

"Un widget pratique qui englobe un certain nombre de widgets généralement requis pour applications de conception de matériaux."

BuildContext est un argument fourni à la méthode de construction comme moyen utile d'interagir avec l'arborescence des widgets qui permet d'accéder à des informations ancestrales importantes qui aident à décrire le widget en cours de construction. N'oubliez pas que la description ne dépend que de ce contexte informations et les propriétés du widget définies dans le constructeur.

Nous examinerons bientôt les widgets de conception de matériaux en détail, lorsque nous explorerons les widgets intégrés disponibles, ainsi que [Chapitre 6 , Thème et style](#) .

En plus d'autres propriétés, MaterialApp contient la propriété home, qui spécifie le premier widget affiché comme page d'accueil de l'application. Ici, la maison est le Widget MyHomePage, qui est le widget avec état de cet exemple.

[126]

Épisode 145

Widgets: création de dispositions dans Flutter

Chapitre 4

En utilisant la classe Navigator, MaterialApp vous permet de définir widgets à afficher pour des itinéraires spécifiques avec un historique logique de navigation, en gérant la pile arrière (nous vérifierons les itinéraires et navigation dans les pages au [chapitre 7 , Routage: navigation entre les écrans](#)).

Widgets avec état dans le code

MyHomePage est un widget avec état, et donc il est défini avec un objet State, _MyHomePageState, qui contient des propriétés qui affectent l'apparence du widget:

```
class MyHomePage étend StatefulWidget {
    MyHomePage ({Key key, this.title}): super (key: key);
    titre final de la chaîne;

    @passer autre
    _MyHomePageState createState () => _MyHomePageState ();
}
```

En étendant StatefulWidget, MyHomePage doit retourner un objet State valide dans son createState (). Dans notre exemple, il renvoie une instance de _MyHomePageState.

Normalement, les widgets avec état définissent leurs classes d'état correspondantes dans le même fichier. En outre, l'état est généralement privé pour la bibliothèque de widgets, car les clients externes n'ont pas besoin d'interagir directement avec lui.

La classe _MyHomePageState suivante représente l'objet State de MyHomePage widget:

```
la classe _MyHomePageState étend l'état <MyHomePage> {
    int _counter = 0;

    void _incrementCounter () {
        setState (() {
            _counter++;
        });
    }

    @passer autre
    Construction du widget (contexte BuildContext) {
        retour échafaudage (
            appBar: AppBar (
                title: Texte (widget.title),
            ),
            corps: Centre (

```

[127]

Épisode 146

Widgets: création de dispositions dans Flutter

Chapitre 4

```
enfant: Colonne (
    mainAxisAlignment: MainAxisAlignment.center,
    enfants: <Widget> [
        Texte(
            'Vous avez appuyé sur le bouton autant de fois:',
        ),
        Texte(
            '$_counter',
            style: Theme.of (contexte).textTheme.display1,
        ),
    ],
),
),
FloatingActionButton: FloatingActionButton (
    onPressed: _incrementCounter,
    info-bulle: 'Incrémenter',
    enfant: Icône (Icons.add),
),
// Cette virgule de fin rend le formatage automatique plus agréable.
);
}
}
```

Un état de widget valide est une classe qui étend la classe State du framework, qui est définie dans la documentation comme suit:

"La logique et l'état interne d'un StatefulWidget."

L'état du widget MyHomePage est défini par une seule propriété, _counter, le

La propriété _counter conserve le nombre de pressions sur le bouton d'incrémantation en bas-coin droit de l'écran. Cette fois, la classe descendante du widget State est responsable pour construire le widget. Il est composé d'un widget Texte qui affiche la valeur _counter.

Le texte est un widget intégré utilisé pour afficher du texte à l'écran. En savoir plus sur les

dans les widgets seront traités dans la section suivante.

Un widget avec état est censé changer son apparence au cours de sa vie - c'est-à-dire définit qu'il va changer - et doit donc être reconstruit pour refléter de tels changements. Ici le changement se produit dans la méthode `_incrementCounter()`, qui est appelée à chaque fois que le bouton d'incrémentation est touché.

[128]

Épisode 147

Widgets: création de dispositions dans Flutter

Chapitre 4

Notez l'utilisation de la propriété `onPressed` du widget `FloatingActionButton`.

`FloatingActionButton` est le bouton d'action flottant de conception matérielle, et cette propriété reçoit un rappel de fonction qui sera exécuté sur presse:

Page d'accueil Flutter Demo (Ceci est une image de la page d'accueil Flutter Demo. Les autres informations (superposées) ne sont pas importantes ici)

Comment le framework sait-il quand quelque chose dans le widget change et qu'il doit le reconstruire? `setState` est la réponse. Cette méthode reçoit une fonction comme paramètre où vous devez mettre à jour l'état correspondant du widget (c'est-à-dire `_incrementCounter`). En appelant `setState`, le framework est notifié qu'il a besoin de reconstruire le widget. Dans l'exemple précédent, il est appelé pour refléter la nouvelle valeur de la propriété `_counter`.

[129]

Épisode 148

Widgets: création de dispositions dans Flutter

Chapitre 4

Widgets hérités

Outre StatelessWidget et StatefulWidget, il existe un autre type de widget dans le Framework Flutter, InheritedWidget. Parfois, un widget doit avoir accès à données dans l'arbre, et dans un tel cas, nous aurions besoin de répliquer les informations jusqu'à le widget intéressé. Ce processus est illustré dans le diagramme suivant:

Supposons que certains des widgets situés dans l'arborescence aient besoin d'accéder à la propriété title depuis le widget racine. Pour ce faire, avec StatelessWidget ou StatefulWidget, nous aurions besoin pour répliquer la propriété dans les widgets correspondants et la transmettre via le constructeur. Il peut être ennuyeux de répliquer la propriété sur tous les widgets enfants afin que la valeur atteint le widget intéressé.

[130]

Épisode 149

Widgets: création de dispositions dans Flutter

Chapitre 4

Pour résoudre ce problème, Flutter fournit la classe InheritedWidget, une sorte auxiliaire de widget qui aide à propager les informations dans l'arborescence comme indiqué ci-dessous diagramme:

En ajoutant un InheritedWidget à l'arborescence, tout widget en dessous peut accéder aux données exposées à l'aide de la méthode inheritFromWidgetOfExactType (InheritedWidget) de BuildContext classe qui reçoit un type InheritedWidget comme paramètre et utilise le tree pour trouver le premier widget ancestral du type demandé.

Il existe des apparences très courantes de l'utilisation de InheritedWidget dans Flutter. L'une des utilisations les plus courantes est celle du Classe de thème, qui aide à décrire les couleurs pour une application entière. nous va l'examiner au [chapitre 5, Gestion des entrées et des gestes de l'utilisateur](#).

[131]

Épisode 150

Widgets: création de dispositions dans Flutter

Chapitre 4

Propriété de la clé du widget

Si vous regardez à la fois les constructeurs des classes StatelessWidget et StatefulWidget, vous remarquerez un paramètre nommé key. C'est une propriété importante pour les widgets dans Battement. Il aide au rendu de l'arborescence des widgets à l'arborescence des éléments. Outre le type et une référence au widget correspondant, cet élément contient également la clé qui identifie le widget dans l'arborescence. La propriété key permet de préserver l'état d'un widget entre reconstruit. L'utilisation la plus courante de la clé est lorsque nous traitons des collections de widgets qui ont le même type; donc, sans clés, l'arborescence des éléments ne saurait état correspond à quel widget, car ils auraient tous le même type. Par exemple, chaque fois qu'un widget change de position ou de niveau dans l'arborescence des widgets, la correspondance est effectuée dans le arborescence d'éléments pour voir ce qui doit être mis à jour à l'écran pour refléter le nouveau widget structure. Lorsqu'un widget a un état, il a besoin de l'état correspondant pour être déplacé avec ça. En bref, c'est ce qu'une clé aide à faire. En maintenant la valeur clé, le élément en question connaîtra l'état du widget correspondant qui doit être avec lui.

Nous utiliserons les clés de notre application plus loin dans ce livre. Si vous avez besoin de trouver plus de détails sur la façon dont la clé affecte le widget et les types de clés disponibles maintenant, veuillez consulter la documentation officielle sur les clés: <https://flutter.io/doc>

Widgets intégrés

Flutter met l'accent sur l'interface utilisateur et, pour cette raison, il contient un grand catalogue de widgets pour permettre la construction d'interfaces personnalisées selon vos besoins.

Les widgets disponibles de Flutter vont de simples, tels que le widget Texte dans le Flutter exemple d'application de compteur, à des widgets complexes qui aident à concevoir une interface utilisateur dynamique avec animations et gestion de gestes multiples.

Widgets de base

Les widgets de base de Flutter sont un bon point de départ, non seulement pour leur facilité d'utilisation, mais aussi car ils démontrent la puissance et la flexibilité du cadre, même dans des cas simples.

Nous n'étudierons pas tous les widgets disponibles car cela briserait l'objectif de ce livre, nous n'en listerons donc que certains pour votre connaissance et nous utiliserons certains des les mettre en pratique afin que vous puissiez apprendre les bases à explorer davantage.

[132]

Épisode 151

Widgets: création de dispositions dans Flutter

Chapitre 4

Le widget Texte

Le texte affiche une chaîne de texte permettant le style:

```
Texte(  
    "Ceci est un texte",  
)
```

Les propriétés les plus courantes du widget Texte sont les suivantes:

style: une classe qui compose le style d'un texte. Il expose des propriétés qui permet de changer la couleur du texte, l'arrière-plan, la famille de polices (permettant l'utilisation d'un police personnalisée à partir des actifs; voir le [chapitre 3, An Introduction to Flutter](#)), hauteur de ligne, la taille de la police, et ainsi de suite.

textAlign: contrôle l'alignement horizontal du texte, en donnant des options telles que le centre alignés ou justifiés, par exemple.

maxLines: permet de spécifier un nombre maximum de lignes pour le texte qui être tronquée si la limite est dépassée.

débordement: définira comment le texte sera tronqué en cas de débordement, donnant des options telles que la spécification d'une limite de lignes max. Cela peut être en ajoutant des points de suspension au fin, par exemple.

Pour voir toutes les propriétés du widget Texte disponibles, veuillez consulter le site officiel

Page de documentation du widget texte: [http://](https://api.flutter.dev/flutter/widgets/Text-class.html)

[Texte- clé](#)

Le widget Image

L'image affiche une image provenant de différentes sources et formats. À partir de la documentation, les les formats d'image sont JPEG, PNG, GIF, GIF animé, WebP, WebP animé, BMP et WBMP:

```
Image(  
    image: AssetImage (  
        "assets / dart_logo.jpg"  
)  
)
```

[133]

Épisode 152

Widgets: création de dispositions dans Flutter

Chapitre 4

La propriété Image du widget spécifie ImageProvider. L'image à afficher peut

proviennent de différentes sources. La classe Image contient différents constructeurs pour différents moyens de charger des images:

```
Image (https://.. / api flutter / dart / core / class / Image.html)
obtention d'une image

peinture / classe
Image.asset (https://.. / api flutter / dart / core / class / Image.html#asset) : obtient une image à partir d'un asset

actif.html : obtient une image à partir d'un fichier HTML actif
de Assetf : obtient une image à partir d'un fichier Asset
AssetBundle_ classe. l : obtient une image à partir d'un AssetBundle

Image.asset (
  'assets / dart_logo.jpg',
)

Image.network (https://.. / api flutter / dart / core / class / Image.html#network)

Image.network (
  <https://picsum.photos/250?image=9>,
)

Fichier d'image (https://.. / api flutter / dart / core / class / Image.html#file)
html ) : obtient une image à partir d'un fichier HTML
dev / fl : obtient une image à partir d'un fichier dev / fl

Fichier d'image(
  Fichier (chemin_fichier)
)

Image.memory (https://.. / api flutter / dart / core / class / Image.html#memory)
https://.. / assets / dart\_logo.jpg

Image.memory (
  Uint8List (octets_image),
)
```

[134]

Épisode 153

Widgets: création de dispositions dans Flutter

Chapitre 4

Outre la propriété Image, il existe d'autres propriétés couramment utilisées:

- hauteur / largeur: pour spécifier les contraintes de taille d'une image
- répéter: pour répéter l'image pour couvrir l'espace disponible
- alignement: pour aligner l'image dans une position spécifique dans ses limites
- fit: Pour spécifier comment l'image doit être inscrite dans l'espace disponible

Pour voir toutes les propriétés du widget Image disponibles, veuillez vous rendre sur le site officiel image widget docs page: <https://docs.flutter.io>

[Image_ classe](#)

Widgets Material Design et iOS Cupertino

De nombreux widgets de Flutter descendant d'une manière ou d'une autre d'une plate-forme spécifique directive: **Material Design** ou **iOS Cupertino**. Cela aide le développeur à suivre la plate-forme directives spécifiques de la manière la plus simple possible.

Si vous ne connaissez pas les directives de Material Design ou iOS Cupertino, alors

c'est le bon moment pour les connaître:

Matériel de conception: <https:///. />

[l'introduction.html](#).

iOS Cupertino: <https:///. / développement/>

[interface_](#) lignes d'i

Flutter, par exemple, n'a pas de widget Button; au lieu de cela, il fournit un bouton alternatif implémentations pour Google Material Design et iOS Cupertino.

Nous n'allons pas approfondir chaque propriété ou comportement de widget, car ceux-ci peuvent être facilement étudiés en exécutant des exemples ou en visitant la documentation. Aussi, vous pouvez consulter la galerie Flutter application sur Google Play (<https:///. / jeu>).

[google.com](#)

une démon

[135]

Épisode 154

Widgets: création de dispositions dans Flutter

Chapitre 4

Boutons

Du côté de la conception de matériaux, Flutter implémente les composants de bouton suivants:

RaisedButton: bouton en relief Material Design. Un bouton en relief se compose d'un morceau de matériau rectangulaire qui plane sur l'interface.

FloatingActionButton: un bouton d'action flottant est un bouton icône circulaire qui survole le contenu pour promouvoir une action principale dans l'application.

FlatButton: un bouton plat est une section imprimée sur un widget Matériau qui réagit à touche en éclaboussant / ondulant avec la couleur.

IconButton: un bouton icône est une image imprimée sur un widget Matériau qui réagit aux touches par éclaboussures / ondulations.

L'encre, sur le site Web des directives de conception de matériaux, peut être expliquée comme suit:

"Composant qui fournit une action radiale sous la forme d'une ondulation visuelle expansion vers l'extérieur à partir du toucher de l'utilisateur."

DropDownButton: affiche l'élément actuellement sélectionné et une flèche qui ouvre un menu pour sélectionner un autre élément.

PopUpMenuItem: affiche un menu lorsque vous appuyez sur.

Pour le style iOS Cupertino, Flutter fournit la classe CupertinoButton.

En raison des directives de Material Design, de l'élévation, des effets d'encre et de la lumière effets, les widgets Material Design sont un peu plus chers que Cupertino widgets. Pas au point de s'inquiéter, mais c'est intéressant à savoir.

Échafaud

Scaffold implémente la structure de base d'un visuel Material Design ou iOS Cupertino disposition. Pour la conception de matériaux, le widget Scaffold peut contenir plusieurs matériaux de conception Composants:

body: Le contenu principal de l'échafaudage. Il est affiché sous AppBar, le cas échéant.

AppBar: une barre d'applications se compose d'une barre d'outils et potentiellement d'autres widgets.

TabBar: un widget Material Design qui affiche une ligne horizontale d'onglets. C'est généralement utilisé dans le cadre d'AppBar.

TabBarView: une vue de page qui affiche le widget qui correspond au onglet actuellement sélectionné. Généralement utilisé avec TabBar et utilisé comme

un widget de corps.

[136]

Épisode 155

Widgets: création de dispositions dans Flutter

Chapitre 4

BottomNavigationBar: Les barres de navigation inférieures facilitent l'exploration et basculez entre les vues de premier niveau en un seul clic.

Tiroir: panneau Material Design qui se glisse horizontalement à partir du bord d'un échafaudage pour afficher les liens de navigation dans une application.

Dans iOS Cupertino, la structure est différente avec des transitions et des comportements spécifiques.

Les classes iOS Cupertino disponibles sont CupertinoPageScaffold

et CupertinoTabScaffold, qui sont généralement composés des éléments suivants:

CupertinoNavigationBar : une barre de navigation supérieure. Il est généralement utilisé avec CupertinoPageScaffold.

CupertinoTabBar: une barre d'onglets inférieure généralement utilisée avec CupertinoTabScaffold.

Dialogues

Les boîtes de dialogue Material Design et Cupertino sont implémentées par Flutter. Sur le matériel

Côté design, ce sont SimpleDialog et AlertDialog; du côté de Cupertino, ils

sont CupertinoDialog et CupertinoAlertDialog.

Champs de texte

Les champs de texte sont également implémentés dans les deux directives, par le widget TextField dans Material Conception et par le widget CupertinoTextField dans iOS Cupertino. Les deux affichent le clavier pour la saisie utilisateur. Certaines de leurs propriétés communes sont les suivantes:

autofocus: indique si le TextField doit être mis au point automatiquement (si rien sinon est déjà concentré)

enabled: pour définir le champ comme modifiable ou non

keyboardType: pour changer le type de clavier affiché à l'utilisateur lorsque édition

Pour voir tous les widgets TextField et CupertinoTextField disponibles properties, veuillez vous rendre sur la page de documentation officielle des widgets:

[flutter.io / docs / development / ui / text / text-input](https://flutter.io/docs/development/ui/text/text-input)

[137]

Épisode 156

Widgets: création de dispositions dans Flutter

Chapitre 4

Widgets de sélection

Les widgets de contrôle disponibles pour la sélection dans Material Design sont les suivants:

La case à cocher permet la sélection de plusieurs options dans une liste.

La radio permet une seule sélection dans une liste d'options.

L'interrupteur permet de basculer (marche / arrêt) d'une seule option.
Le curseur permet la sélection d'une valeur dans une plage en déplaçant le curseur du curseur.

Du côté d'iOS Cupertino, certaines de ces fonctionnalités de widget n'existent pas; cependant, il existe des alternatives disponibles:

CupertinoActionSheet: Une feuille d'action modale de style iOS pour choisir un option parmi tant d'autres.

CupertinoPicker: également un contrôle de sélection. Il est utilisé pour sélectionner un élément dans une courte liste.

CupertinoSegmentedControl: se comporte comme un bouton radio, où la sélection est un élément unique d'une liste d'options.

CupertinoSlider: similaire à Slider dans la conception matérielle.

CupertinoSwitch: Ceci est également similaire au Switch de Material Design.

Sélecteurs de date et d'heure

Pour la conception de matériaux, Flutter fournit des sélecteurs de date et d'heure via showDatePicker et fonctions showTimePicker, qui crée et affiche la boîte de dialogue Material Design pour le

actions correspondantes. Du côté iOS Cupertino, le CupertinoDatePicker

et les widgets CupertinoTimerPicker sont fournis, suivant le précédent

Style CupertinoPicker .

Autres composants

Il existe également des composants spécifiques à la conception qui sont uniques à chaque plate-forme. Matériel La conception, par exemple, a le concept de **cartes** , qui sont définies comme suit dans le Documentation:

"Une feuille de matériel utilisée pour représenter certaines informations connexes."

De l'autre côté des choses, les widgets spécifiques à Cupertino peuvent avoir des transitions uniques présentes dans le monde iOS.

[138]

Épisode 157

Widgets: création de dispositions dans Flutter

Chapitre 4

Pour plus de détails, n'hésitez pas à consulter le catalogue de widgets Flutter sur le site flutter.io: <https://flutter.io/docs/develop/widgets>

Comprendre les widgets de mise en page intégrés

Certains widgets semblent ne pas apparaître à l'écran pour l'utilisateur, mais s'ils sont dans l'arborescence des widgets, ils seront là d'une manière ou d'une autre, affectant l'apparence d'un widget enfant (comme positionné ou stylisé, par exemple).

Pour positionner un bouton dans le coin inférieur de l'écran, par exemple, nous pourrions spécifier un positionnement lié à l'écran, mais comme vous l'avez peut-être remarqué, les boutons et autres widgets n'ont pas de propriété Position. Alors, vous vous demandez peut-être: " *Comment les widgets organisés sur l'écran?* " La réponse est encore une fois les *widgets* . C'est vrai! Flutter fournit des widgets pour composer la mise en page elle-même, avec positionnement, dimensionnement, style, etc.

Conteneurs

L'affichage d'un seul widget à l'écran n'est pas un bon moyen d'organiser une interface utilisateur. Nous allons généralement jeter une liste de widgets organisés d'une manière spécifique; pour ce faire, nous utilisons des widgets conteneurs.

Les conteneurs les plus courants dans Flutter sont les widgets Row et Column. Ils ont un

children qui permet à ce que une liste de widget affiche dans une certaine direction (c'est-à-dire

Un autre widget largement utilisé est le widget Stack, qui organise les enfants en couches, où un enfant peut chevaucher un autre enfant partiellement ou totalement.

Si vous avez déjà développé une sorte d'application mobile, vous avez peut-être déjà utilisé listes et grilles. Flutter fournit des classes pour les deux: à savoir, ListView et

Widgets GridView. En outre, d'autres widgets de conteneur moins typiques mais néanmoins importants sont disponibles, comme Table, par exemple, qui organise les enfants dans une mise en page tabulaire.

[139]

Épisode 158

Widgets: création de dispositions dans Flutter

Chapitre 4

Stylisme et positionnement

La tâche de positionnement d'un widget enfant dans un conteneur, tel qu'un widget Stack, par exemple, se fait en utilisant d'autres widgets. Flutter fournit des widgets pour des tâches très spécifiques. Centrer un widget à l'intérieur d'un conteneur se fait en l'enveloppant dans un widget Centre. Aligner un enfant widget relatif à un parent peut être fait avec le widget Aligner, où vous spécifiez la position souhaitée grâce à sa propriété d'alignement. Un autre widget utile est Padding, ce qui nous permet de spécifier un espace autour de l'enfant donné. Les fonctionnalités de ces widgets sont agrégés dans le widget Conteneur, qui combine les éléments communs de positionnement et style des widgets pour les appliquer directement à un enfant, ce qui rend le code beaucoup plus propre et plus court.

Autres widgets (gestes, animations et transformations)

Flutter fournit des widgets pour tout ce qui concerne l'interface utilisateur. Par exemple, des gestes tels que le défilement ou les touches seront toutes liées à un widget qui gère les gestes. Animations et les transformations, telles que la mise à l'échelle et la rotation, sont également toutes gérées par des widgets spécifiques. Nous en vérifierons certains en détail dans les chapitres suivants, lorsque nous développerons parties d'une petite application.

Nous ne sommes pas en mesure d'explorer tous les widgets disponibles et toutes les combinaisons possibles de leur. Nous commencerons notre voyage en développant une petite application dans la section suivante, où nous explorerons certains des widgets disponibles dans toutes les catégories afin que vous puissiez visualisez comment utiliser certains d'entre eux. Mais surtout, vous découvrirez les principes de base de la création de mises en page dans Flutter. Une fois que cela est fait, découvrez de nouveaux des widgets spécifiques seront une tâche facile.

Pendant l'écriture de ce livre, Flutter fait évoluer un autre grand fonctionnalité, **Platform View**, qui nous permet d'utiliser toutes les interfaces natives qui sont déjà disponibles sur iOS et Android. En savoir plus dans le [chapitre 11](#), *Vues de la plate-forme et intégration de la carte*, dans la section *Affichage d'une carte*.

[140]

Épisode 159

Widgets: création de dispositions dans Flutter

Chapitre 4

Créer une interface utilisateur avec des widgets (gestionnaire de faveur application)

Maintenant que nous connaissons certains des widgets disponibles de Flutter, il est temps de commencer le petit application que nous construirons au cours du livre.

L'application que nous allons développer sera une application de gestionnaire de faveur. Ce sera un petit réseau où un ami peut demander une faveur à un autre ami, et cet ami peut accepter ou refuser de faire la faveur. En acceptant, la faveur entre dans la liste des choses à faire de l'utilisateur. C'est comme un application à faire où les tâches à effectuer sont proposées par les amis de l'utilisateur et uniquement acceptées ou rejetées par l'utilisateur. Dans cette application, nous explorerons de nombreux concepts qui peuvent aider dans l'application développement.

Dans les chapitres suivants, nous ajouterons des fonctionnalités à l'application, progressivement découvrir toutes les différentes pièces qui composent une application Flutter.

Les écrans de l'application

L'application *Friend Favors* comprendra deux écrans. Dans les deux, nous utiliserons Material Composants de conception fournis par Flutter. Le premier écran sera une liste de faveurs, et le second sera un formulaire pour demander une faveur à un ami. Pour l'instant, nous allons utiliser in listes de mémoire; autrement dit, les informations ne seront stockées nulle part ailleurs que dans l'application.

Le code de l'application

Le code de l'application n'est pas encore entièrement fonctionnel. Il est suffisamment petit pour afficher la mise en page. Il construit une instance de widget MaterialApp qui définit l'écran d'accueil sur la page de liste des favoris, appelé FavorsPage:

```
class MyApp étend StatelessWidget {
    // utilisation des valeurs fictives du fichier de fléchettes mock_favors pour le moment
    @passer outre
    Construction du widget (contexte BuildContext) {
        retour MaterialApp (
            titre: 'Flutter Demo',
            accueil: FavorsPage (
                pendingAnswerFavors: mockPendingFavors,
                completedFavors: mockCompletedFavors,
                refuséFavors: mockRefusedFavors,
                AcceptéFaveurs: mockDoingFavors,
            ),
        );
    }
}
```

[141]

Épisode 160

Widgets: création de dispositions dans Flutter

Chapitre 4

```
        );
    }
}
```

MaterialApp est un widget qui fournit des outils utiles pour l'ensemble de l'application. L'un d'eux est le widget Thème, qui nous permet de changer les styles et les couleurs de nos applications en suivant le Directives de conception des matériaux. Un autre outil utile est le widget Navigator, qui gère un ensemble de widgets d'application à la manière d'une pile de navigation, où nous pouvons naviguer vers un écran en appuyant dessus sur le navigateur ou revenez en arrière en le faisant apparaître. Nous utiliserons les deux widgets dans l'application. Navigator est déjà appliqué ici lorsque nous définissons la propriété home de le widget MaterialApp. Navigator fonctionne de manière route vers widget; c'est-à-dire qu'il y a quelques façons de définir des itinéraires spécifiques pointant vers des widgets spécifiques, et lorsque nous naviguons vers un

route, il pourra naviguer vers le widget correspondant. En réglant le maison avec un widget, nous disons au Navigateur d'utiliser ce widget comme la route «/» (la route initiale de l'application).

Comme vous pouvez le voir, le widget FavorsPage a quelques paramètres de constructeur rempli. Continuez à lire pour voir ce qu'ils sont.

Dans cette première étape, nous examinerons la structure initiale de la mise en page de l'application, qui évoluera pour la fin du livre avec de nouveaux styles et widgets. Dans le prochain chapitre, vous apprendrez comment ajouter des méthodes de saisie utilisateur à l'aide de taps et de champs de formulaire. Plus tard, dans le [chapitre 6, Theming and Styling](#), nous verrons comment personnaliser l'apparence de l'application en utilisant le thème matériel. Alors, commençons par jeter un œil aux dispositions d'écran.

[142]

Épisode 161

Widgets: création de dispositions dans Flutter

Chapitre 4

Favorise l'écran d'accueil de l'application

Le premier écran de l'application est l'écran d'accueil, qui sera composé de quatre onglets listant les faveurs et leurs statuts:

En attente de faveurs : les faveurs demandées par certains amis auxquelles nous n'avons pas répondu encore

En cours / faire des faveurs : les faveurs acceptées; c'est-à-dire les faveurs que nous faisons maintenant:

[143]

Épisode 162

Widgets: création de dispositions dans Flutter

Chapitre 4

Faveurs terminées : Les **faveurs** déjà terminées

Faveurs refusées: Une liste de faveurs que nous avons refusé de faire (non acceptées):

La liste contiendra toutes les faveurs de l'application, séparées par catégories, comme indiqué. Au sommet de la mise en page, nous avons une instance TabBar qui sera utilisée pour changer l'onglet dans la liste souhaitée. Ensuite, sur chaque onglet, nous avons une liste d'éléments de carte, qui contiennent des actions correspondant à sa catégorie.

[144]

Épisode 163

Nous avons créé des classes Friend et Favor pour représenter les données de l'application. Tu peut regarder cela de plus près dans le code source du chapitre (le répertoire hands_on_layouts) pour ce livre. Les voici, simples classes de données, qui ne contiennent aucune logique métier avancée.

De plus, le bouton d'action flottant en bas de l'écran doit rediriger vers le **Demandez un écran de faveur**, où l'utilisateur pourra demander une faveur à un ami.

Le code de mise en page

Tout d'abord, nous définissons notre page d'accueil comme une instance StatelessWidget, comme nous nous en soucions maintenant uniquement à propos de la mise en page et n'ont aucune action à gérer qui entraînerait un état changement. C'est pourquoi le widget parent, MyApp, transmet des valeurs aux champs de liste définis. N'oubliez pas que lorsqu'un widget est sans état, sa description est définie par le widget parent lors de sa création. Ceci est illustré dans le code suivant:

```
La classe FavorsPage étend StatelessWidget {
    // utilisation des valeurs fictives du fichier de fléchettes mock_favors pour le moment
    Liste finale <Favor> pendingAnswerFavors;
    Liste finale <Favor> acceptéeFavors;
    Liste finale <Favor> completedFavors;
    Liste finale <Favor> refuséeFavors;

    FavorsPage ({
        Clé clé,
        this.pendingAnswerFavors,
        this.acceptedFavors,
        this.completedFavors,
        this.refusedFavors,
    }): super (clé: clé);

    @passer autre
    Widget build (contexte BuildContext) {...} // par souci de concision
}
```

[145]

Épisode 164

Comme indiqué dans le code précédent, le widget est défini par les listes spécifiques aux faveurs. Aussi, remarquez le paramètre clé. Bien que ce ne soit pas vraiment nécessaire ici, nous le définissons comme recommandé de le faire comme une bonne pratique.

Jetons un coup d'œil à la méthode build () pour voir ce qui compose le widget:

```
@passer autre
Construction du widget (contexte BuildContext) {
    renvoie DefaultTabController (
        longueur: 4,
        enfant: Scaffold (
            appBar : AppBar (
                title: Texte ("Vos faveurs"),
                en bas: TabBar (
                    isScrollable: vrai,
                    onglets: [
                        _buildCategoryTab ("Requêtes"),
                        _buildCategoryTab ("Faire"),
                        _buildCategoryTab ("Terminé"),

```

```

        _buildCategoryTab ("Refusé"),
    ],
),
),
corps : TabBarView (
enfants: [
_favorsList ("Demandes en attente", pendingAnswerFavors),
_favorsList ("Faire", AcceptedFavors),
_favorsList ("Completed", completedFavors),
_favorsList ("Refusé", refuséFavors),
],
),
floatingActionButton : FloatingActionButton (
onPressed: () {},
info-bulle: "Demander une faveur",
enfant: Icône (Icons.add),
),
),
);
}
}

```

[146]

Épisode 165

Widgets: création de dispositions dans Flutter

Chapitre 4

Le premier widget présent dans la sous-arborescence de widgets FavorsPage est le DefaultTabController widget, qui gère le changement d'onglet pour nous. Après cela, nous avons un widget Scaffold, qui implémente la structure de base de Material Design. Ici, nous utilisons déjà des de ces éléments, y compris la barre d'application et le bouton d'action flottant. Ce widget est très utile pour concevoir des applications qui suivent le Material Design car il fournit des propriétés utiles basé sur les lignes directrices:

Dans AppBar, nous avons ajouté un titre à l'aide d'un widget Texte. Dans certains cas, nous pouvons également y ajouter des actions ou une mise en page personnalisée. Ici, nous avons ajouté un Instance TabBar juste en bas de la barre d'application qui affichera les onglets.

Dans FloatingActionButton, nous n'avons pas non plus trop changé; nous avons seulement ajouté une icône à l'aide du widget Icône, qui contient une icône Material Design fourni par le cadre.

La propriété body du widget Scaffold est l'endroit où nous concevons la mise en page elle-même. Il est défini comme suit: un widget TabBarView affiche le widget correspondant pour l'onglet sélectionné dans l'instance DefaultTabController définie précédemment. Sa propriété enfantine est ce qui requiert l'attention; ça correspond aux onglets de la barre d'onglets et renvoie le widget correspondant de chaque onglet.

Les éléments de la barre d'onglets sont créés par la méthode _buildCategoryChip (), comme suit:

```

La classe FavorsPage étend StatelessWidget {
// ... champs, méthode de construction et autres
Widget _buildCategoryTab (titre de la chaîne) {
retour Tab (
enfant: Texte (titre),
);
}
}

```

Comme vous pouvez le voir, la fonction crée un élément d'onglet de catégorie en créant simplement un Tab> Sous-arborescence de texte, où titre est l'identifiant de l'élément.

[147]

Épisode 166

*Widgets: création de dispositions dans Flutter**Chapitre 4*

De la même manière, chaque section de liste de favoris est définie dans sa propre méthode, `_favorsList()`:

```
La classe FavorsPage étend StatelessWidget {
  // ... champs, méthode de construction et autres

  Widget _favorsList(String title, List <Favor> favors) {
    colonne de retour (
      mainAxisAlignment: MainAxisAlignment.max,
      enfants: <Widget> [
        Rembourrage(
          enfant: Texte (titre),
          padding: EdgeInsets.only (haut: 16,0),
        ),
        Étendu(
          enfant: ListView.builder (
            physique: BouncingScrollPhysics (),
            itemCount: favors.length,
            itemBuilder: (contexte BuildContext, index int) {
              faveur finale = faveurs [index];
              Carte de retour (
                clé: ValueKey (favor.uuid),
                margin: EdgeInsets.symmetric (vertical: 10,0,
                horizontal: 25,0),
                enfant: Rembourrage (
                  enfant: Colonne (
                    enfants: <Widget> [
                      _itemHeader (faveur),
                      Texte (favor.description),
                      _itemFooter (faveur)
                    ],
                  ),
                  remplissage: EdgeInsets.all (8,0),
                ),
              );
            },
          ),
        ],
      );
    }
}
```

Le widget de section de faveur est représenté par un widget de colonne qui a deux widgets enfants:

Un widget Texte (avec un parent Padding) contenant le titre de la section, comme précédemment
Une instance ListView qui contiendra chacun des éléments favoris

[148]

Épisode 167

*Widgets: création de dispositions dans Flutter**Chapitre 4*

Cette liste est construite de manière distincte des précédentes. Ici, nous avons utilisé le constructeur nommé `ListView.builder()`. Ce constructeur de liste attend `itemCount`

et les instances itemBuilder, que nous définissons à l'aide de la liste passée en argument dans le appel à `_favorsList()`:

`itemCount` est simplement la taille de la liste.
`itemBuilder` doit être une fonction qui renvoie le widget correspondant au élément dans une position spécifique. Cette fonction reçoit `BuildContext`, comme le `build()` du widget, ainsi qu'une position d'index (ici, nous avons utilisé la index pour obtenir la faveur correspondante dans la liste des faveurs).

Cette forme de création d'éléments est optimale pour les grandes listes, les listes qui se développent au cours du cycle de vie, ou même des listes à défilement infini (que vous avez peut-être déjà vues dans certaines applications), car construit des éléments uniquement s'ils sont nécessaires, évitant ainsi le gaspillage de ressources de calcul.

Changer la physique de la liste des faveurs avec (`physics`):

`BouncingScrollPhysics()` provoque le rebond de la liste effet vu dans les listes iOS.

La valeur de la fonction `itemBuilder` crée un widget `Carte` pour chaque faveur dans les faveurs liste d'arguments en obtenant l'élément correspondant avec `favor = favors [index] ;`

La partie restante du constructeur est la suivante:

```
Carte de retour (
    clé: ValueKey (favor.uuid),
    margin : EdgeInsets.symmetric (vertical: 10,0, horizontal: 25,0),
    enfant: Rembourrage (
        enfant: Colonne (
            enfants: <Widget> [
                _itemHeader (faveur),
                Texte (favor.description),
                _itemFooter (faveur)
            ],
        ),
        remplissage: EdgeInsets.all (8.0),
    ),
);
;
```

Lorsque nous parlons d'éléments de liste, nous aurons toujours besoin d'une clé pour le widget, du moins lorsque nous ajouterez-y la gestion des événements tap. En effet, les listes dans Flutter peuvent recycler de nombreux éléments lors des événements de défilement, et en ajoutant une clé, nous affirmerons que le widget spécifique a un état spécifique qui lui est associé.

[149]

Épisode 168

Widgets: création de dispositions dans Flutter

Chapitre 4

La nouvelle partie ici est la propriété `margin` du widget `Carte`, qui ajoute une marge au widget. Dans ce cas, nous ajoutons 10,0 creux pour le haut et le bas, et 25,0 pour la gauche et droite. Son corps enfant est divisé en trois parties:

Tout d'abord, il y a l'en-tête, qui montre l'ami qui a fait la demande de faveur, défini dans la fonction `_itemHeader()`:

```
Row _itemHeader (faveur favor) {
    return Row (
        enfants: <Widget> [
            CircleAvatar (
                backgroundImage: NetworkImage (
                    favor.friend.photoURL,
                ),
            ),
            Etendu(
                enfant: Rembourrage (
                    padding: EdgeInsets.only (gauche: 8,0),
                    enfant: Texte ("$ {favor.friend.name} vous a demandé de ...
                ")),
            )
        ],
    );
}
```

L'en-tête est défini comme un sous-arbre Row> [CircleAvatar, Expanded]. Il commence avec une définition de ligne (fonctionne comme le widget Colonne, mais dans l'axe horizontal) qui a une instance CircleAvatar, une image de cercle qui représente un utilisateur. Ici, nous ont utilisé le fournisseur NetworkImage; nous lui passons simplement une URL d'image et laissons ça charge pour nous. L'espace restant du widget Row est utilisé par Text avec quelques Rembourrage qui montre le nom de l'ami.

Deuxièmement, il y a le contenu, qui est juste un widget de texte avec la faveur la description.

Enfin, il y a le pied de page, qui contient les actions disponibles pour la faveur demande en fonction de la catégorie de faveur, définie dans le _itemFooter () fonction:

```
Widget _itemFooter (faveur de faveur) {
    if (favor.isCompleted) {
        format final = DateFormat ();
        Conteneur de retour (
            margin: EdgeInsets.only (haut: 8,0),
            alignment: Alignment.centerRight,
            enfant: Chip (
```

[150]

Épisode 169

Widgets: création de dispositions dans Flutter

Chapitre 4

```
label: Text ("Terminé à:
\$ {format.format (favor.completed)} "),
),
);
}
if (favor.isRequested) {
return Row (
    mainAxisAlignment: MainAxisAlignment.end,
    enfants: <Widget> [
        FlatButton (
            enfant: Texte ("Refuser"),
            onPressed: () {}),
        ),
        FlatButton (
            enfant: Texte ("Do"),
            onPressed: () {}),
    )
],
);
}
if (favor.isDoing) {
return Row (
    mainAxisAlignment: MainAxisAlignment.end,
    enfants: <Widget> [
        FlatButton (
            enfant: texte ("abandonner"),
            onPressed: () {}),
        ),
        FlatButton (
            enfant: texte ("complet"),
            onPressed: () {}),
    ]
),
}
}
return Container ();
```

[151]

Épisode 170

*Widgets: création de dispositions dans Flutter**Chapitre 4*

La fonction `_itemFooter()` renvoie un widget en fonction du statut de faveur.

Les statuts de faveur sont définis par les getters de la classe `Favor`:

Dans la phase de **demande** (la faveur n'a pas encore été acceptée ou refusée), nous renvoie un widget `Row` avec deux instances de `FlatButton` matérielles dessus avec le actions disponibles correspondantes, refuser ou faire. `FlatButton` est un matériau Bouton de conception qui n'a pas d'élévation ou de couleur d'arrière-plan.

Pour la **faire** étape, nous revenons un widget `Row` avec ou rejeté complète actions comme `FlatButtons`.

Pour l' état **terminé** , nous affichons la date et lheure terminées formatées en utilisant la classe `DateFormat` de Dart dans un widget `Chip` pour différencier du reste du texte.

Dans le statut **refusé** , nous renvoyons un widget Conteneur sans taille contraintes; c'est un conteneur vide (il ne prend pas de place sur un disposition).

Vous pouvez utiliser les méthodes de classe d'assistance `EdgeInsets` chaque fois que vous êtes définir le remplissage ou la marge. Il a des méthodes utiles pour cela. Vérifiez page de documentation officielle: <https://api.flutter.dev/flutter/painting/EdgeInsets-class.html>

Comme nous l'avons vu dans l'implémentation des listes de favoris, il existe différents widgets composant le disposition. Notez cependant que nous ne gérons aucune action de l'utilisateur ici; nous vérifierons dans sur tout cela dans le prochain chapitre. Jetons un coup d'œil à l'écran de demande de faveur.

Notez la propriété `onPressed` sur `FlatButton`; il définit l'action lorsque l'utilisateur appuie dessus. Nous examinerons cela dans le [chapitre 5](#), *Manipulation Saisie et gestes de l'utilisateur* , alors continuez!

[152]

Épisode 171

*Widgets: création de dispositions dans Flutter**Chapitre 4*

L'écran de demande de faveur

L'écran de demande de faveur sera l'endroit où l'interaction utilisateur-application se produit. Pour l'instant, nous ne regarderons que la disposition de cet écran. Au fur et à mesure que le livre avance, nous rejoindrons les pièces pour sélectionner l'ami pour demander la faveur, et également enregistrer la faveur sur la télécommande Firebase base de données:

[153]

Épisode 172

Widgets: création de dispositions dans Flutter

Chapitre 4

Le widget d'écran de demande de faveur contient également un widget d'échafaudage de conception matérielle avec une barre d'application qui contient des actions cette fois. Le corps du widget Scaffold contient des champs qui aura des informations d'entrée de l'utilisateur pour la création d'une demande de faveur.

Le code de mise en page

Le widget RequestFavorPage est également sans état pour le moment car nous ne nous soucions que de sa mise en page actuellement:

```
La classe RequestFavorPage étend StatelessWidget {  
    Liste finale des amis <Amis>;  
  
    RequestFavorPage ({Key key, this.friends}): super (key: key);  
  
    @passer autre  
    Widget build (BuildContext context) {...} // par souci de brièveté  
}
```

Comme vous pouvez le voir, la seule chose dans la description du widget est la liste d'amis, qui doit être fourni par le widget parent car il s'agit d'une instance StatelessWidget pour le moment.

Pour savoir comment naviguer entre les écrans (c'est-à-dire à partir de la liste des favoris à l'écran **Demander une faveur**), passez au [chapitre 7](#). *Routage: navigation entre les écrans*, où l'on parle de routage et de navigation.

La méthode build () du widget commence comme suit:

```
@passer autre
```

```

    retour Scaffold {
      appBar: AppBar (
        title: Text ("Demander une faveur"),
        leader: CloseButton (),
        actions: <Widget> [
          FlatButton (
            enfant: Text ("SAVE"), textColor: Colors.white, onPressed: () {
              {},
            },
          ),
        ],
        body: ... // continue ci-dessous
      ...
    }
  }
}

```

[154]

Épisode 173

Widgets: création de dispositions dans Flutter

Chapitre 4

appBar contient ici deux nouvelles propriétés:

La propriété principale, qui est un widget affiché *avant* le titre. Dans ce cas, nous utilisons un widget CloseButton qui est un bouton intégré au matériau Widget Navigator (plus à ce sujet dans le [chapitre 7, Routage: navigation entre Écrans](#)).

La propriété actions, qui reçoit une liste de widgets à afficher après le titre; dans ce cas, nous affichons une instance FlatButton à l'aide de laquelle nous enregistrerons le demande de faveur.

Le corps de Scaffold définit la disposition dans un widget Colonne. Il contient deux nouveaux properties: le premier est mainAxisSize, qui définit la taille dans l'axe vertical; ici nous utilisons MainAxisSize.min pour qu'il n'occupe que l'espace nécessaire. La deuxième est crossAxisAlignment, qui définit où aligner les enfants sur l'axe horizontal. Par défaut, Column aligne ses enfants horizontalement au centre. En utilisant cette propriété, nous peut changer ce comportement. Il y a trois widgets enfants dans Column qui prendront l'utilisateur contribution:

Un widget DropdownButtonFormField qui répertorie le widget DropdownMenuItem éléments dans une fenêtre contextuelle lorsque vous appuyez sur:

```

    ...
    DropdownButtonFormField (
      objets: amis
      .carte(
        (f) => DropdownMenuItem (
          enfant: Text (f.nom),
        ),
      )
      .lister(),
    ),
    ...
  }
}

```

Ici, nous utilisons la méthode map () du type Dart Iterable, où chaque élément de la liste (amis, dans ce cas) est mappé à un nouveau Widget DropdownMenuItem. Ainsi, chaque élément de la liste d'amis sera affiché comme élément de widget dans la liste déroulante.

[155]

Épisode 174

Widgets: création de dispositions dans Flutter

Chapitre 4

Un widget TextFormField qui permet la saisie de texte en tapant sur le clavier:

```
TextFormField (
    maxLines: 5,
    inputFormatters: [LengthLimitingTextInputFormatter(200)],
),
```

Le widget TextFormField permet la saisie de texte. En ajoutant inputFormatters à il, nous pouvons configurer son apparence à l'écran. Ici, nous limitons simplement la longueur totale du texte saisi à 200 caractères en utilisant le LengthLimitingTextInputFormatter, qui est fournie par la bibliothèque flutter / services.

Découvrez tous les utilitaires fournis à partir du flutter / services package sur la page Web du package: <https://flutter.dev/services/>

Un widget DateTimePickerFormField qui permet à l'utilisateur de sélectionner un DateTime et la mappe à un type DateTime Dart:

```
DateTimePickerFormField (
    inputType: InputType.both,
    format: DateFormat("EEEE, MMMM j, aaaa 'à' h: mma"),
    modifiable: faux,
    décoration: InputDecoration (
        labelText: 'Date / Heure', hasFloatingPlaceholder: false),
    onChanged: (dt) {},
),
```

Le widget DateTimePickerFormField n'est pas un widget intégré de Flutter. C'est un plugin tiers de la bibliothèque `datetime_picker_formfield`. Ici, nous définissons certains propriétés pour modifier son apparence:

inputType: sélection de la date, de l'heure ou des deux.
format: un type Dart DateFormat pour définir le format de représentation sous forme de chaîne la valeur.
modifiable: indique si le widget doit être modifiable manuellement par l'utilisateur.
décoration: utilisé pour définir une décoration pour le champ de saisie dans une conception de matériau façon. Notez que nous ne l'avons pas défini pour les autres champs de saisie.
onChanged: rappel appelé avec la nouvelle valeur sélectionnée par l'utilisateur.

[156]

Épisode 175

Widgets: création de dispositions dans Flutter

Chapitre 4

Pour découvrir toutes les options disponibles et comment utiliser le Widget `DateTimePickerFormField`, veuillez visiter https://pub.dev/packages/datetime_picker_formfield.

Outre les champs de saisie, il existe également des widgets Conteneur et Texte dans la colonne pour aide à la mise en forme et à la conception de l'écran. Jetez un œil au code source du chapitre pour le code de mise en page complet.

Créer des widgets personnalisés

Lors de la création d'interfaces utilisateur avec Flutter, nous devrons toujours créer des widgets personnalisés; nous

ne peut pas et ne veut pas y échapper. Après tout, la composition des widgets pour la construction des interfaces uniques sont ce que Flutter permet si bien.

Dans l'application, nous avons déjà créé une partie de la mise en page, et la seule Les widgets que nous avons créés sont les widgets FavorsPage et RequestFavorPage.

Vous avez peut-être noté également qu'en raison de la manière de composer les mises en page dans Flutter, le code peut devenir énorme et difficile à entretenir. Pour y remédier, nous avons créé de petites méthodes qui ont divisé la création du widget en parties pour construire la mise en page complète.

Diviser les widgets en petites méthodes est bon pour aider le code à devenir plus petit, mais c'est pas aussi bon pour Flutter. Dans notre cas, nous n'avons pas encore de mise en page complexe, donc c'est OK, mais dans le cas d'une mise en page complexe où l'arborescence des widgets peut changer plusieurs fois, ayant les widgets en tant que méthodes intégrées n'aideront pas le framework à optimiser le processus de rendu.

Pour aider le framework à optimiser le processus de rendu, nous devrions plutôt diviser notre méthodes en petits widgets utiles. Ainsi, l'[arborescence des widgets | Les opérations de l'arborescence d'éléments](#) être optimisé. N'oubliez pas que le type de widget aide le framework à savoir quand un widget change et doit être reconstruit, ce qui a un impact sur l'ensemble du processus de rendu. Alors revisitons notre widget FavorsPage et convertissons les petites méthodes de widget en de nouvelles petites widgets.

La méthode `_favorsList()` (voir le code source attaché) peut être refactorisée en un nouveau widget FavorsList. Ensuite, la propriété `itemBuilder` du widget FavorsList peut être refactorisé pour renvoyer un widget `FavorCardItem` qui renvoie l'élément de carte:

```
La classe FavorCardItem étend StatelessWidget {
  faveur finale;

  const FavorCardItem ({Key key, this.favor}): super (key: key);
```

[157]

Épisode 176

Widgets: création de dispositions dans Flutter

Chapitre 4

```
@passer outre
Construction du widget (contexte BuildContext) {
  Carte de retour (
    clé: ValueKey (favor.uuid),
    margin: EdgeInsets.symmetric (vertical: 10,0, horizontal: 25,0),
    enfant: Rembourrage (
      enfant: Colonne (
        enfants: <Widget> [
          _itemHeader (faveur),
          Texte (favor.description),
          _itemFooter (faveur)
        ],
      ),
      remplissage: EdgeInsets.all (8,0),
    ),
  );
}
Widget _itemHeader (Favoriser) {...} // par souci de concision
Widget _itemFooter (Favoriser) {...} // par souci de concision
}
```

La seule chose qui change est l'ajout d'une nouvelle classe avec les champs finaux appropriés question pour le rendu du widget; la méthode `build()` est presque la même que la méthode `_buildFavorsList()` précédente.

Notez que l'élément de carte de faveur contient toujours les parties d'en-tête et de pied de page comme méthodes, `_itemHeader()` et `_itemActions()` respectivement. De cette façon, ils sont petits assez pour ne pas nuire au processus de rendu. Mais rappelez-vous, en les divisant en widgets ne ferait pas de mal non plus.

Avec cette technique d'utilisation du widget de fractionnement, nous allons donner le cadre juste suffisamment d'informations sur nos widgets, et ils se comporteront comme des widgets intégrés et pourront pour être optimisé comme des widgets intégrés.

Je vous recommande de lire cet article de blog intéressant sur le widet performance: <https://iirakran.com/>

[158]

Épisode 177*Widgets: création de dispositions dans Flutter**Chapitre 4*

Sommaire

Dans ce chapitre, nous avons vu chacun des types de widgets Flutter disponibles et leurs différences. les widgets sans état ne sont pas reconstruits fréquemment par le framework; sur l'autre part, les widgets avec état sont reconstruits à chaque fois que leur objet State associé change (ce qui pourrait être lorsque la fonction `setState()` est utilisée, par exemple). On a également vu que Flutter est livré avec de nombreux widgets qui peuvent être combinés pour créer des interfaces utilisateur uniques, et qu'ils n'ont pas non plus besoin d'être des composants visuels sur l'écran de l'utilisateur; ils peuvent être mise en page, style et même des widgets de données, tels que `InheritedWidget`. Nous avons commencé le développement d'une petite application que nous continuons de développer dans les prochains chapitres; nous y ajouterons des fonctions spécifiques pendant que nous présenterons de nouveaux concepts importants sur Battement.

Dans le chapitre suivant, nous verrons comment ajouter une interaction utilisateur à l'application en ajoutant réponses aux taps des utilisateurs et aux entrées de données qui seront stockées ultérieurement dans Firebase.

[159]

Épisode 178

5

Gestion des entrées utilisateur et Gestes

Avec l'utilisation de widgets, il est possible de créer une interface riche en ressources visuelles qui permet également l'interaction de l'utilisateur par des gestes et la saisie de données. Dans ce chapitre, vous allez apprendre à connaître les widgets utilisés pour gérer les gestes de l'utilisateur et recevoir et valider l'utilisateur entrée, ainsi que la façon de créer nos propres entrées personnalisées.

Les sujets suivants seront traités dans ce chapitre:

- Gérer les gestes de l'utilisateur
- Comprendre les widgets d'entrée
- Validation des entrées d'apprentissage
- Créer des entrées personnalisées

Gérer les gestes de l'utilisateur

Une application mobile ne serait rien sans une sorte d'interactivité. Le Flutter framework permet de gérer les gestes de l'utilisateur de toutes les manières possibles, du simple appui au faire glisser et déplacer les gestes. Les événements d'écran du système gestuel de Flutter sont séparés en deux couches, comme suit:

Couches de pointeur : ce sont les couches qui ont des événements de *pointeur*, qui représentent l'utilisateur interactions avec des détails tels que l'emplacement tactile et le mouvement sur l'appareil écran.

Épisode 179

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Gestes : les gestes dans Flutter sont des événements d'interaction au plus haut niveau de définition, et vous en avez peut-être déjà vu certains en action, comme les robinets, traîne, et l'échelle, par exemple. En outre, ils sont le moyen le plus typique de implémentation de la gestion des événements.

Pointeurs

Flutter démarre la gestion des événements dans un calque de bas niveau (**calques de pointeur**), où vous pouvez gérer chaque événement de pointeur et décidez comment le contrôler, par exemple en faisant glisser ou en appuyant une seule fois.

Le framework Flutter implémente la distribution d'événements sur l'arborescence des widgets en suivant un séquence d'événements:

PointerDownEvent est l'endroit où l'interaction commence, avec un pointeur entrant dans contact avec un certain emplacement de l'écran de l'appareil. Ici, le framework recherche l'arborescence de widgets pour le widget qui existe à l'emplacement du pointeur dans le écran. Cette action s'appelle un test de réussite.

Chaque événement suivant est envoyé au widget le plus interne qui correspond au placement, puis à soulever l'arborescence des widgets des parents jusqu'à la racine.

Cette propagation des actions événementielles ne peut pas être interrompue. L'événement pourrait être PointerMoveEvent, où l'emplacement du pointeur est modifié. Ça pourrait également PointerUpEvent ou PointerCancelEvent.

Une interaction peut se terminer par PointerUpEvent ou PointerCancelEvent.

Le premier ici est l'endroit où le pointeur cesse d'être en contact avec l'écran, tandis que ce dernier signifie que l'application ne reçoit plus d'événements sur le pointeur (l'événement n'est pas terminé).

Flutter fournit la classe Listener, qui peut être utilisée pour détecter l'interaction du pointeur événements dont nous avons déjà discuté. Vous pouvez envelopper une arborescence de widgets avec ce widget pour gérer les événements de pointeur sur sa sous-arborescence de widget.

Consultez la page de documentation de classe Listener à <https://api.flutter.dev/flutter/widgets/Listener-class.html>.

[flutter.dev / fl](https://api.flutter.dev/flutter/widgets/Listener-class.html)

[161]

Épisode 180

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Gestes

Bien que possible, il n'est pas toujours pratique de gérer nous-mêmes les événements de pointeur en utilisant le Widget d'écoute. Au lieu de cela, les événements peuvent être gérés sur la deuxième couche du Flutter système de geste. Les gestes sont reconnus à partir de plusieurs événements de pointeur, et même plusieurs pointeurs individuels (multitouch). Il existe plusieurs types de gestes qui peuvent manipulé:

Appuyer : une seule pression / touche sur l'écran de l'appareil.

Appuyez deux fois : appuyez deux fois rapidement sur le même emplacement sur l'écran de l'appareil.

Appuyer et appuyer longuement : une pression sur l'écran de l'appareil, similaire à appuyer, mais en contactant l'écran pendant une longue période avant la libération.

Glisser : Une pression qui commence par un pointeur en contact avec l'écran à un endroit, qui est ensuite déplacé et cesse de contacter à un autre endroit sur l'appareil écran.

Pan : similaire aux événements de glissement. Dans Flutter, ils ont une direction différente; la poêle les gestes couvrent les traînées horizontales et verticales.

Echelle : deux pointeurs utilisés pour un mouvement de glissement pour utiliser un geste d'échelle. C'est aussi similaire à un geste de zoom.

Comme le widget Listener pour les événements de pointeur, Flutter fournit le GestureDetector widget, qui contient des rappels pour tous les événements précédents. Nous devrions les utiliser selon l'effet que nous voulons obtenir.

Robinet

Voyons comment implémenter l'événement tap à l'aide du onTap du widget GestureDetector rappeler:

```
// fait partie de tap_event_example.dart (code source complet dans les fichiers joints)
```

```
la classe _TapWidgetExampleState étend l'état <TapWidgetExample> {
  int _counter = 0;

  @passer outre
  Construction du widget (contexte BuildContext) {
    retournez GestureDetector (
      en fût: () {
```

```
    setState(() {
        _counter++;
    });
},
```

[162]

Épisode 181

*Gestion des entrées et des gestes de l'utilisateur**Chapitre 5*

```
enfant: Conteneur (
    couleur: Colors.grey,
    enfant: Centre (
        enfant: Texte (
            "Nombre de clics: \$ _counter",
            style: Theme.of(context).textTheme.display1,
        ),
        ),
        ),
    );
}
}
```

C'est l'implémentation d'état d'un widget qui contient l'exemple. Il a un seul compteur pour afficher le nombre de tapotements effectués sur l'écran. Dans cet exemple, la propriété onTap détient un rappel qui met à jour l'état du widget après un appui sur l'écran, en incrémentant la valeur de compteur.

Vous pouvez trouver le code source du [chapitre 5, Gestion des entrées utilisateur et Gestures](#), sur GitHub.

Tapez deux fois

Le rappel du double tap est très similaire dans le code:

```
// fait partie de doubletap_event_example.dart (code source complet dans le
des dossiers)

GestureDetector (
    onDoubleTap: () {
        setState(() {
            _counter++;
        });
    },
    child: ... // par souci de brièveté
);
```

La seule différence par rapport à l'élément précédent est la propriété attribuée, onDoubleTap, qui sera appelé chaque fois que des doubles taps sont rapidement effectués au même endroit sur le écran.

[163]

Épisode 182

*Gestion des entrées et des gestes de l'utilisateur**Chapitre 5*

Appuyez et maintenez

Encore une fois, la différence avec les exemples précédents est minime:

```
// fait partie de press_and_hold_event_example.dart (code source complet dans le
https://translate.googleusercontent.com/translate\_f
```

fichiers joints)

```
GestureDetector (
    onLongPress : () {
        setState () {
            _counter++;
        });
    },
    child: ... // par souci de brièveté
);
```

La seule différence par rapport à l'élément précédent est la propriété attribuée, `onLongPress`, qui sera appelé chaque fois qu'un tap est effectué et maintenu pendant un certain temps - un appui *long* - avant d'être libéré de l'écran.

Faire glisser, déplacer et mettre à l'échelle

Les gestes de glisser, de panoramique et d'échelle sont similaires, et dans Flutter, nous devons décider lequel utiliser dans chaque situation, car ils ne peuvent pas être utilisés tous ensemble dans le même `GestureDetector` widget.

Les gestes de glissement sont séparés en gestes *verticaux* et *horizontaux*. Même les rappels sont séparés dans Flutter.

Traînée horizontale

Voyons à quoi ressemble la **version horizontale** dans le code:

```
// fait partie de drag_event_example.dart (code source complet dans les fichiers joints)

GestureDetector (
    onHorizontalDragStart: (Détails DragStartDetails) {
        setState () {
            _move = Offset.zero;
            _dragging = vrai;
        });
    },
    onHorizontalDragUpdate: (Détails de DragUpdateDetails) {
        setState () {
            _move += détails.delta;
        }
    },
    onHorizontalDragEnd: (Détails DragEndDetails) {
        setState () {
            _dragging = faux;
            _dragCount++;
        });
    },
    child: ... // par souci de brièveté
)
```

[164]

Épisode 183

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

```
    });
},
onHorizontalDragEnd: (Détails DragEndDetails) {
    setState () {
        _dragging = faux;
        _dragCount++;
    });
},
child: ... // par souci de brièveté
)
```

Cette fois, nous avons besoin d'un peu plus de travail que pour les événements tap. Dans l'exemple, nous avons trois propriétés présentes dans l'état:

`_dragging`: utilisé pour mettre à jour le texte visualisé par l'utilisateur lors du glissement.
`_dragCount`: Cela accumule le nombre total d'événements de glissement effectués depuis le début finir.
`_move`: qui accumule le décalage du glissement appliquée au texte en utilisant le constructeur `translate` du widget `Transform`.

Nous vérifierons un peu plus les widgets `Transform` sur [Chapitre 14](#), *Manipulations graphiques du widget*.

Comme vous pouvez le voir, les callbacks de glisser reçoivent des paramètres liés à chaque événement - DragStartDetails, DragUpdateDetails et DragEndDetails - qui contiennent des valeurs cela peut aider à chaque étape du traînage.

Glissement vertical

La **version verticale** de la traînée est presque la même que la version horizontale. Le significatif les différences sont dans les propriétés de rappel, qui sont onVerticalDragStart, onVerticalDragUpdate et onVerticalDragEnd.

Ce qui change pour les rappels verticaux et horizontaux en termes de code est la valeur de propriété delta de la classe DragUpdateDetails. Pour horizontal, il ne changera que la partie horizontale du décalage, et pour la verticale, le contraire est le cas.

[165]

Épisode 184

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

La poêle

La **version pan** est également très similaire. Les différences significatives cette fois s'ajoutent aux les propriétés de rappel, qui sont désormais onPanStart, onPanUpdate et onPanEnd. Pour pan glisse, les deux décalages d'axe sont évalués; autrement dit, les deux valeurs delta dans DragUpdateDetails sont présents, de sorte que le glissement n'a aucune limitation de direction.

Vous pouvez trouver le code source de les gestes / lib / example_widgets / pan_example_event.dart fichier sur GitHub.

Échelle

La **version à l'échelle** n'est rien de plus qu'un panoramique sur plus d'un pointeur. Voyons quoi la version à l'échelle du panoramique ressemble à:

```
// fait partie de scale_event_example.dart (code source complet dans le
des dossiers)

GestureDetector (
  onScaleStart: (Détails ScaleStartDetails) {
    setState () {
      _scale = 1,0;
      _resizing = vrai;
    });
  },
  onScaleUpdate: (Détails ScaleUpdateDetails) {
    setState () {
      _scale = détails.scale;
    });
  },
  onScaleEnd: (Détails ScaleEndDetails) {
    setState () {
      _resizing = faux;
      _scaleCount++;
    });
  },
  child: ... // par souci de brièveté
)
```

Épisode 185

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Le code ici est très similaire aux précédents. Nous avons trois propriétés dans l'état:

- `_resizing`: Ceci est utilisé pour mettre à jour le texte visualisé par l'utilisateur lors du redimensionnement en utilisant le geste d'échelle.
- `_scaleCount`: Cela accumule le nombre total d'événements d'échelle effectués depuis le début finir.
- `_scale`: Cela stocke la valeur d'échelle du paramètre `ScaleUpdateDetails`, et cela plus tard est appliqué au widget Texte en utilisant le constructeur d'échelle de le widget Transformer.

Comme vous pouvez le voir, les rappels d'échelle ressemblent beaucoup aux rappels de glissement en ce qu'ils recevoient les paramètres liés à chaque événement - `ScaleStartDetails`, `ScaleUpdateDetails`, et `ScaleEndDetails` - qui contiennent des valeurs qui peuvent aider à chaque étape de l'échelle un événement.

Gestes dans les widgets matériels

Les widgets Material Design et iOS Cupertino ont de nombreux gestes abstraits pour certains propriété en utilisant le widget `GestureDetector` en interne dans leur code. Par exemple, les widgets matériels tels que `RaisedButton` utilisent le widget `InkWell` à côté de l'événement `tap`. Il fait l'effet d'éclaboussure sur le widget cible. En outre, la propriété `onPressed` de `RaisedButton` expose la fonctionnalité `tap` qui peut être utilisée pour implémenter l'action de le bouton. Prenons l'exemple suivant:

```
// partie du fichier main.dart (exemple de répertoire "input" joint)
RaisedButton(
    onPressed: () {
        print("Exécution de la validation");
        // ... valider
    },
    enfant: Texte ("valider"),
)
```

Un enfant `Text` est affiché dans le `RaisedButton` et sa pression est gérée dans le `onPressed` méthode, comme indiqué précédemment.

Épisode 186

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Widgets d'entrée

La gestion des gestes est un bon point de départ pour l'interaction avec l'utilisateur, mais c'est évidemment pas assez. L'obtention de données utilisateur est ce qui ajoute du contenu à de nombreuses applications.

Flutter fournit de nombreux widgets de données d'entrée pour aider le développeur à obtenir différents types de informations de l'utilisateur. Nous avons déjà vu certains d'entre eux dans [Chapitre 4](#), *Widgets: Création de dispositions dans Flutter*, y compris `TextField` et différents types de sélecteur et Widgets de sélection.

Bien que nous puissions gérer toutes les données entrées par l'utilisateur par nous-mêmes (disons, dans une racine widget qui contient tous les champs de saisie), cela peut devenir fastidieux, car cela pourrait nous conduire ayant de nombreux champs et donc nous finirions probablement par augmenter la complexité du code. Scission tous les widgets d'entrée en petits morceaux aide, mais ne résout pas tout.

Flutter fournit deux widgets pour aider à organiser l'entrée dans le code, le valider et fournir retour rapide à l'utilisateur. Ce sont les widgets Form et TextFormField.

FormField et TextField

Le widget TextFormField fonctionne comme une classe de base pour créer notre propre champ de formulaire, utilisé pour intégrer

Widget de formulaire. Ses fonctions sont les suivantes:

- Pour aider le processus de réglage et de récupération de la valeur d'entrée actuelle
- Pour valider la valeur d'entrée actuelle
- Pour fournir des commentaires à partir des validations

TextFormField peut vivre sans les widgets Form, mais ce n'est pas typique - uniquement lorsque nous avons, disons, un seul TextFormField à l'écran.

De nombreux widgets d'entrée intégrés de Flutter sont livrés avec un widget TextFormField correspondant la mise en oeuvre. Par exemple, le widget TextField a le TextFormField. le

Le widget TextFormField facilite l'accès à la valeur TextField et ajoute également Form les comportements qui lui sont liés (comme la validation).

Un widget TextField permet à l'utilisateur de saisir du texte avec un clavier. Le widget TextField expose la méthode onChanged, qui peut être utilisée pour écouter les changements dans son valeur. Une autre façon d'écouter les changements consiste à utiliser un contrôleur (voir *Utilisation d'un contrôleur* section).

[168]

Épisode 187

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Utilisation d'un contrôleur

Lorsqu'il est utilisé isolé d'un formulaire, c'est-à-dire en utilisant le widget TextField, nous devons utiliser sa propriété de contrôleur pour accéder à sa valeur. Ceci est fait avec la classe TextEditingController:

```
_controller final = TextEditingController.fromValue (
    TextEditingValue(text: "Valeur initiale"),
);
```

Après avoir instancié le TextEditingController, nous le définissons dans le contrôleur propriété du widget TextField afin qu'il "contrôle" le widget texte:

```
Champ de texte(
    contrôleur: _controller,
);
```

Comme vous pouvez le voir, nous pouvons également définir une valeur initiale pour TextField.

TextEditingController est notifié chaque fois que le widget TextField a une nouvelle valeur. Pour écouter les changements, nous devons ajouter un écouteur à notre _controller:

```
_controller.addListener (_textFieldEvent);
```

_textFieldEvent doit être une fonction qui sera appelée à chaque fois que le widget TextField change.

Consultez l'exemple complet dans les fichiers de chapitre joints.

Accéder à l'état de TextFormField

Si nous utilisons le widget TextFormField, les choses deviennent plus simples:

```
_key final = GlobalKey<FormFieldState<String>>();  
...  
TextFormField(  
    key: _key,  
);
```

[169]

Épisode 188

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Nous pouvons ajouter une clé à notre TextFormField qui pourra ensuite être utilisée pour accéder au widget état actuel via la valeur key.currentState, qui contiendra la valeur mise à jour du champ.

Le type spécialisé de clé fait référence au type de données avec lesquelles le champ de saisie fonctionne. dans le exemple précédent, il s'agit de String, car il s'agit d'un widget TextField, donc la clé dépend sur le widget particulier utilisé.

La classe FormFieldState <String> fournit également d'autres méthodes et propriétés utiles pour traiter avec FormField:

validate () appellera le rappel du validateur du widget, qui devrait vérifier son valeur actuelle et renvoie un message d'erreur, ou null si elle est valide.

hasError et errorText résultent de validations précédentes utilisant le précédent fonction. Dans les widgets de matériaux, par exemple, cela ajoute un petit texte près du champ, fournissant une rétroaction appropriée à l'utilisateur sur l'erreur.

save () appellera le rappel onSaved du widget. C'est l'action qui se produit lorsque la saisie est effectuée par l'utilisateur (lors de son enregistrement).

reset () mettra le champ dans son état initial, avec la valeur initiale (le cas échéant), effacer également les erreurs de validation.

Forme

Avoir un FormFieldWidget nous aide à accéder et à valider individuellement ses informations. Mais, pour résoudre le problème d'avoir trop de champs, nous pouvons utiliser le widget Formulaire. La forme widget regroupe les instances de FormFieldWidget de manière logique, ce qui nous permet d'effectuer opérations, y compris l'accès aux informations sur le terrain et leur validation plus simple.

Le widget Formulaire nous permet d'exécuter facilement les méthodes suivantes sur tous les champs descendants:

save (): Cela appellera la méthode de sauvegarde de toutes les instances FormField et fonctionnera comme avant. C'est comme une sauvegarde par lots de tous les champs.

validate (): cela appellera la méthode de validation de toutes les instances FormField, provoquant les erreurs apparaissent toutes à la fois.

reset (): Cela appellera la méthode de réinitialisation de toutes les instances FormField. Cela apportera la forme entière à son état initial.

[170]

Épisode 189

Accéder à l'état du formulaire

Fournir l'accès à l'objet associé à l'état actuel du formulaire est utile afin que nous puissions lancer son validation, enregistrez son contenu ou réinitialisez-le de n'importe où dans l'arborescence des widgets (c'est-à-dire un bouton presse). Il existe deux façons d'accéder à l'état associé au widget Formulaire.

Utiliser une clé

Le widget Form est utilisé avec le compagnon d'une clé de type FormState qui contient helpers pour gérer tous les enfants de ses instances TextFormField:

```
_key final = GlobalKey<FormFieldState<String>>();  
...  
Forme(  
    clé: _key,  
    enfant: Colonne (  
        enfants: <Widget> [  
            TextFormField (),  
            TextFormField (),  
        ],  
    ),  
);
```

Ensuite, nous pouvons utiliser la clé pour récupérer l'état associé au Form et appeler sa validation avec `_key.currentState.validate ()`. Voyons maintenant la deuxième option.

Utilisation d'InheritedWidget

Le widget Form est livré avec une classe utile pour se passer de la nécessité d'ajouter une clé et bénéficient toujours de ses avantages.

Chaque widget Formulaire de l'arborescence est associé à un InheritedWidget. Forme et beaucoup d'autres widgets exposent cela dans une méthode statique appelée `of ()`, où nous passons BuildContext, et il *regarde dans l'arbre* pour trouver l'état correspondant que nous recherchons. Sachant cela, si nous devons accéder au widget Form quelque part en dessous dans l'arborescence, nous pouvons utiliser `Form.of ()`, et nous avons accès aux mêmes fonctions que nous aurions si nous utilisions la propriété key:

```
// fait partie de l'exemple input / main.dart (code source complet joint)  
// build () dans la classe InputFormInheritedStateExamplesWidget  
  
Forme(  
    enfant: Colonne (  
        mainAxisAlignment: MainAxisAlignment.min,  
        enfants: <Widget> [  
            TextFormField (
```

[171]

Épisode 190

```
validateur: (valeur de chaîne) {  
    valeur de retour.isEmpty? "ne peut pas être vide": null;  
},  
),  
TextFormField (),  
Constructeur(  
    constructeur: (BuildContext context) => RaisedButton (  
        onPressed: () {  
            print ("Exécution de la validation");  
            final valide = Form.of (contexte) .validate ();  
            print ("valide: $ valide");  
        },  
        enfant: Texte ("valider"),  
    ),  
),  
],  
,
```

);

...

Portez une attention particulière au widget Builder utilisé pour rendre RaisedButton. Comme nous avons vu précédemment, le widget hérité peut être regardé sur l'arbre. Considérez l'utilisation suivante de RaisedButton directement dans le widget Colonne, comme suit:

```
Colonne(
    enfants: [
        // ... autres enfants, supprimés par souci de concision
        TextFormField(),
        RaisedButton (
            onPressed: () {
                print ("Exécution de la validation");
                final valide = Form.of (contexte) .validate (); // cela ne fonctionnerait pas
                                                    // (mauvais contexte)
                print ("valide: $ valide");
            },
            enfant: Texte ("valider"),
        ),
    ],
...
)
```

Lorsque nous utilisons Form.of (contexte), nous transmettons le contexte actuel du widget. Dans le précédent exemple, le contexte utilisé dans le rappel onPressed sera le contexte InputFormInheritedStateExamplesWidget, et ainsi, la recherche de l'arborescence sera ne parvient pas à trouver un widget Formulaire. En utilisant le widget Builder, nous déléguons sa construction à un callback, cette fois en utilisant le bon contexte (celui de l'enfant), et quand il recherche l'arborescence, il trouvera avec succès l'instance FormState.

[172]

Épisode 191

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Validation des entrées (formulaires)

La gestion de plusieurs widgets TextFormField est acceptable lorsque vous parlez de quelques valeurs, mais lorsque le la quantité de données augmente, en les organisant à l'écran, en validant tout correctement, et fournir rapidement des commentaires aux utilisateurs peut devenir plus difficile. C'est pourquoi Flutter fournit le Widget de formulaire.

Validation de l'entrée utilisateur

La validation de l'entrée utilisateur est l'une des principales fonctions du widget Formulaire. Afin de rendre le la saisie des données saisies par l'utilisateur est cohérente, il est fondamental de la vérifier, car l'utilisateur probablement ne connaît pas toutes les valeurs autorisées.

Le widget Form, combiné aux instances TextFormField, aide le développeur à afficher un message d'erreur approprié si certaines valeurs d'entrée doivent être corrigées avant d'enregistrer le formulaire data via sa fonction save ().

Nous avons déjà vu, dans les exemples de formulaire précédents, comment valider le champ Form valeurs:

1. Créez un widget Form avec un TextFormField dessus.
2. Définissez la logique de validation sur chaque propriété du validateur TextFormField:

```
TextFormField (
    validate: (valeur de chaîne) {
        valeur de retour.isEmpty? "La valeur ne peut pas être vide": null;
    },
)
```

3. Appelez validate () sur FormState en utilisant sa clé ou la méthode Form.of discuté précédemment. Cela appellera chaque méthode enfant TextFormField validate (), et lorsque la validation est réussie, elle retournera vrai et faux dans le cas contraire.
4. validate () renvoie un booléen pour que nous puissions manipuler son résultat et faire notre logique basé sur cela.

[173]

Épisode 192

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Entrée personnalisée et TextFormField

Nous avons vu comment les widgets Form et TextFormField facilitent la manipulation des entrées et validation. En outre, nous savons que Flutter est livré avec une série de widgets d'entrée qui sont

Les variantes de TextFormField, et ainsi, contiennent des fonctions d'assistance pour accéder et valider les données, pour exemple.

L'extensibilité et la flexibilité de Flutter sont partout dans le cadre. Alors, créer des champs personnalisés est logiquement possible, où nous pouvons ajouter notre propre méthode d'entrée, exposer validation via le rappel du validateur, et utilisez également les méthodes save () et reset ().

Créer des entrées personnalisées

Créer une entrée personnalisée dans Flutter est aussi simple que créer un widget normal, avec le méthodes supplémentaires décrites précédemment. Nous le faisons normalement en étendant le widget TextFormField <inputType>, où inputType est le type de valeur de l'entrée widget.

Ainsi, le processus typique est le suivant:

1. Créez un widget personnalisé qui étend le widget StatefulWidget (pour garder une trace de value) et accepte l'entrée de l'utilisateur en encapsulant un autre widget d'entrée, ou en personnalisant l'ensemble du processus, par exemple en utilisant des gestes.
2. Créez un widget qui étend TextFormField qui affiche essentiellement le widget d'entrée créé à l'étape précédente et expose également ses champs.

[174]

Épisode 193

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Exemple de widget d'entrée personnalisé

Plus tard, au [chapitre 8](#), *Firebase Plugins*, nous verrons comment ajouter une authentification à notre application. Pour maintenant, nous allons créer un widget personnalisé qui sera similaire à celui utilisé à cette étape. L'authentification sera basée sur les services d'authentification Firebase utilisant le numéro de téléphone, où le numéro de téléphone fourni reçoit un code de vérification à six chiffres qui doit correspondre la valeur du serveur afin de réussir la connexion. Pour l'instant, ce sont toutes les informations dont nous avons besoin à savoir pour la création du widget d'entrée personnalisé. Voici à quoi ça va ressembler:

Le widget sera un simple widget d'entrée à six chiffres, qui deviendra plus tard un `FormField` widget et exposez les méthodes `save()`, `reset()` et `validate()`.

[175]

Épisode 194

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Plus tard, sur l'écran de connexion, nous utiliserons le Flutter plugin `community_code_input` pour remplacer ce widget. Plus d'informations peuvent être trouvés à <https://.../pub> de

Créer un widget d'entrée

Nous commençons par créer un widget personnalisé normal. Ici, nous exposons quelques propriétés. Garder à sachez que dans une vraie application, nous exposerions probablement plus que les propriétés exposé ici, mais c'est suffisant pour cet exemple:

```
La classe VerificationCodeInput étend StatelessWidget {
  BorderSide borderSide final;
  final onChanged;
  contrôleur final;

  ... // autres parties supprimées par souci de concision
}
```

La seule propriété importante exposée ici est le contrôleur. Nous verrons la raison dans quelques moments. Commençons par vérifier la classe State associée:

```

la classe _VerificationCodeInputState étend l'état <VerificationCodeInput> {
  @passer autre
  Construction du widget (contexte BuildContext) {
    retourne TextField (
      contrôleur: widget.controller ,
      inputFormatters: [
        WhitelistingTextInputFormatter (RegExp ("[0-9]")),
        LengthLimitingTextInputFormatter (6),
      ],
      textAlign: TextAlign.center,
      décoration: InputDecoration (
        bordure: OutlineInputBorder (
          borderSide: widget.borderSide,
        ),
      ),
      keyboardType: TextInputType.number,
      onChanged: widget.onChanged,
    );
  }
}

```

[176]

Épisode 195

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Comme vous pouvez le voir, le widget est simplement un TextField avec une personnalisation prédéfinie:

WhitelistingTextInputFormatter nous permet de spécifier une expression regex avec les caractères autorisés pour l'entrée. En définissant le type de clavier avec keyboardType: TextInputType.number, nous pouvons également limiter le caractères aux nombres.

LengthLimitingTextInputFormatter spécifie une limite maximale de caractères pour l'entrée.

En outre, une bordure est ajoutée via la classe OutlineInputBorder.

Prenez note de la partie importante de ce code: controller: widget.controller. Ici, nous définissons le contrôleur du widget TextField pour qu'il soit notre propre contrôleur afin que nous puissions prendre le contrôle de sa valeur.

Transformez le widget en widget TextFormField

Pour transformer le widget en un widget TextFormField, nous commençons par créer un widget qui étend le Classe TextFormField, qui est un StatefulWidget avec certaines fonctionnalités Form.

Cette fois, commençons par vérifier l'objet State associé au nouveau widget. Faisons-le par le diviser en plusieurs parties:

```

// partie initiale de _VerificationCodeFormFieldState
TextEditingController final _controller = TextEditingController (texte: "");

@passer autre
void initState () {
  super.initState ();
  _controller.addListener (_controllerChanged);
}

```

À partir du code précédent, vous pouvez vérifier qu'il a un seul champ _controller, qui représente le contrôleur utilisé par le widget TextFormField. Il doit être dans l'État donc il persiste contre les modifications de mise en page. Comme vous pouvez le voir, il est initialisé dans initState () fonction. Cela s'appelle la première fois que l'objet widget est inséré dans l'arborescence des widgets. Ici, nous y ajoutons un auditeur, afin de savoir quand la valeur est modifiée dans le _controllerChanged écouteur.

[177]

Épisode 196

*Gestion des entrées et des gestes de l'utilisateur**Chapitre 5*

Le reste du widget est le suivant:

```
void _controllerChanged () {
    didChange (_controller.text);
}

@passer autre
void reset () {
    super.reset ();
    _controller.text = "";
}

@passer autre
void dispose () {
    _controller?.removeListener (_controllerChanged);
    super.dispose ();
}
```

Il existe également d'autres méthodes importantes que nous devons ignorer pour que cela fonctionne correctement:

Avec initState (), nous pouvons trouver son équivalent opposé dans le disposer () méthode. Ici, nous nous arrêtons pour écouter les changements dans le contrôleur.

La méthode reset () est remplacée, nous pouvons donc définir le _controller.text sur vide, rendant à nouveau le champ de saisie vide.

L'écouteur _controllerChanged () notifie le super

État de FormFieldState via sa méthode didChange () afin qu'il puisse mettre à jour son état (via setState ()) et informez tout widget Formulaire qui le contient de la modification.

Examinons maintenant le code du widget TextFormField pour voir comment cela fonctionne:

```
La classe VerificationCodeFormField étend TextFormField <String> {
    contrôleur final TextEditingController;

    VerificationCodeFormField ({
        Clé clé,
        FormFieldSetter <String> onSaved,
        this.controller,
        Validateur FormFieldValidator <String>,
    }): super (
        clé: clé,
        validateur: validateur,
        générateur: (champ FormFieldState <String>) {
            _VerificationCodeFormFieldState state = champ;
            retourne VerificationCodeInput (
                contrôleur: state.controller,
            );
    );
}
```

[178]

Épisode 197

*Gestion des entrées et des gestes de l'utilisateur**Chapitre 5*

```
},
);

@passer autre
FormFieldState <String> createState () =>
    _VerificationCodeFormFieldState();
```

}

La nouvelle partie ici est dans le constructeur. Le widget FormField contient le générateur callback qui devrait construire son widget d'entrée associé. Il passe l'état actuel de la objet afin que nous puissions créer le widget et conserver les informations actuelles. Comme vous pouvez le voir, nous l'utilisons pour passer le contrôleur construit dans l'état, donc il persiste même lorsque le champ est reconstruit.

C'est ainsi que nous maintenons le widget et l'état synchronisés, et intégrons également avec le Classe de formulaire.

Vous pouvez vérifier le code source complet de ce widget FormField personnalisé dans le fichier verification_code_input_widget.dart de l'entrée projet d'exemples.

Mettre tous ensemble

Maintenant que nous savons comment utiliser les événements gestuels et les widgets d'entrée pour ajouter une interaction utilisateur à nos écrans d'application, il est temps d'incrémenter notre application avec ces fonctions. Revisitons nos écrans pour y ajouter des gestes et des validations d'entrée.

Écran des faveurs

Le premier écran de l'application répertorie les différentes faveurs et leurs statuts. Outre la liste, le seules les actions que l'utilisateur peut effectuer sont les suivantes:

[179]

Épisode 198

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Écran des faveurs (Ceci est une image de l'écran des faveurs. Les autres informations (floues et superposées) ne sont pas importantes ici.

1. Concentrez-vous sur la section de catégorie de faveur sélectionnée. Ceci est déjà fait pour nous par le Widget DefaultTabController (il existe un widget ListView qui gérera faire glisser / faire défiler les gestes en interne).

[180]

Épisode 199

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

2. Refusez ou faites les faveurs demandées. Par exemple, une faveur a été demandée par un ami, et l'utilisateur peut l'accepter ou le rejeter. Donc, en appuyant sur l'un des boutons fait que la faveur change son statut en **Refused** ou **Doing**.
3. Similaire au cas précédent, mais cette fois, une demande de faveur acceptée est en attente d'achèvement, et ces boutons permettent à l'utilisateur d'abandonner ou de terminer un favoriser; c'est-à-dire qu'en appuyant dessus, le statut de faveur passe à **Refusé** et **Terminé**, respectivement.
4. Enfin, nous avons le bouton **Demandez une faveur**, qui ouvre essentiellement une deuxième application écran au robinet pour nous permettre de demander une faveur à certains de nos amis.

Comme vous pouvez le voir à partir des gestes précédents, nous allons traiter de tap, scroll, et faire glisser des gestes. Nous pouvons être effectués directement avec GestureDetector, mais, comme nous utilisons les widgets Button et ListView, cela change un peu. Rappelez-vous, Flutter's les widgets intégrés sont également composés de nombreux autres widgets intégrés, nous allons donc traiter avec GestureDetector indirectement.

En pratique, nous manipulerons les robinets par nous-mêmes, les autres gestes étant gérés par le widgets que nous avons utilisés: défilement avec ListView et glissements et taps avec TabBar et TabView.

Appuyez sur les gestes sur l'onglet Favoris

Comme nous l'avons souligné précédemment, le DefaultTabController modifie l'onglet actuellement visible widget lorsque l'utilisateur appuie sur la barre d'onglets ou glisse vers la gauche ou la droite sur la vue. En utilisant ce widget, nous n'avons pas besoin de spécifier un contrôleur dans TabBar et TabView descendante.

Pour plus de détails sur le widget TabController, consultez la page de documentation sur [https: / .. / docs / fl](https://.. / docs / fl)

[TabController](#)- classe. Htr

[181]

Épisode 200

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Appuyez sur les gestes sur FavorCards

À partir de la propriété favour du widget FavorCardItem, nous pouvons manipuler son statut en modifier ses valeurs de champ acceptées et complétées. Cependant, cela ne supprimera pas l'élément à partir de la liste actuelle et ajoutez-le à la nouvelle liste cible. Pour ce faire, nous aurions besoin d'accéder au liste actuelle, supprimez l'élément favori à partir de là et ajoutez-le à la nouvelle liste, en fonction de le bouton enfoncé.

Nous pourrions utiliser notre liste globale de faveurs directement dans la méthode onPressed de l'élément de carte, mais ceci impliquerait de distribuer la logique métier via les widgets, ce qui semble bien maintenant, mais peut se compliquer facilement.

Alors, où devrions-nous gérer cette action efficacement? Nous pourrions gérer toutes ces actions en le widget FavorsPage, qui contient toutes les listes de faveurs. Mais attendez - FavorsPage est un StatelessWidget, les listes de faveurs sont chargées dans sa méthode de constructeur, et comme c'est sans état, ils seront chargés à chaque reconstruction du widget, perdant nos modifications.

Faire des FavorsPage un StatefulWidget

La première étape pour rendre notre application interactive est de changer FavorsPage en un StatefulWidget:

```
La classe FavorsPage étend StatefulWidget {
  FavorsPage () {
    Clé clé,
  }: super (clé: clé);

  @passer autre
  State < StatefulWidget > createState () => FavorsPageState ();
}
```

La première chose que nous changeons est l'ancêtre de FavorsPage, et maintenant son seul travail est de retourner un Instance FavorsPageState dans la méthode createState ():

```
La classe FavorsPageState étend l'état < FavorsPage > {
  // utilisation des valeurs fictives du fichier de fléchettes mock_favors pour le moment
  List < Favor > pendingAnswerFavors;
  List < Favor > acceptéFavors;
  List < Favor > completedFavors;
  List < Favor > refuséFavors;

  @passer autre
  void initState () {
    super.initState ();
  }
```

[182]

Épisode 201

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

```
pendingAnswerFavors = Liste ();
AcceptedFavors = Liste ();
completeFavors = Liste ();
refuséFavors = Liste ();

loadFavors ();
}

void loadFavors () {
  pendingAnswerFavors.addAll (mockPendingFavors);
  AcceptedFavors.addAll (mockDoingFavors);
  completeFavors.addAll (mockCompletedFavors);
  refuséFavors.addAll (mockRefusedFavors);
```

```

    }

    @passer outre
    Widget build (BuildContext context) {...} // masqué par souci de brièveté
}

```

Maintenant, l'objet State contient les informations qui doivent être conservées entre les reconstructions, et cet objet sera l'emplacement de toutes les actions pour les faveurs. Bien qu'il ne soit pas optimal, il sera au moins centralisée en un seul endroit. Je dirais que nous avons besoin d'une sorte de architecture pour faire cela correctement: MVP, MVVM, BloC et Redux sont quelques exemples. Cependant, pour garder les choses simples, nous utiliserons l'approche que nous avons adoptée ici.

Vous pouvez consulter le guide officiel de gestion de l'État comme première étape pour architecture d'application ainsi que quelques alternatives d'architecture disponibles à https://./flutter_mngt_and_architecture-101_scope

Alors, commençons par gérer les actions de demande en attente. Ils ont été définis comme **Refuse** ou **Do**. Pour les gérer, nous devons passer un gestionnaire à la propriété onPressed de nos widgets FlatButton déjà définis dans le FavorCardItem.

À partir de la méthode onPressed du bouton, nous devons accéder d'une manière ou d'une autre à FavorsPageState pour effectuez ces actions. Cela peut être fait avec la méthode ancestorStateOfType () de la classe BuildContext, qui recherche dans l'arborescence un objet State du type donné:

```

// fait partie de la classe FavorsPageState
static FavorsPageState of (contexte BuildContext) {
    return context.ancestorStateOfType (TypeMatcher < FavorsPageState > ());
}

```

[183]

Épisode 202

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Un modèle courant pour fournir cette fonction consiste à ajouter une méthode statique sur le type donné, appelé de, qui effectuera l'appel à la fonction de cadre. Ceci est fait pour fournir un moyen abrégé d'accéder à l'état avec moins de code.

Refuser la gestion des actions

Voici à quoi ressemble le bouton **Refuser** après avoir utilisé la fonctionnalité susmentionnée:

```

// fait partie de la classe hands_on_input / lib / main.dart FavorCardItem
// Méthode _itemFooter
FlatButton (
    enfant: Texte ("Refuser"),
    onPressed: () {
        FavorsPageState.of (contexte) .refuseToDo (faveur);
        // nous avons changé _itemFooter pour obtenir le contexte afin que nous
        // peut-il récupérer l'état de la page des faveurs
    },
)

```

En appelant FavorsPageState.of (context), nous avons accès à l'état actuel du FavorsPageState type associé au contexte.

Pour appliquer le changement, nous appelons la méthode refuseToDo (favor) de la Classe FavorsPageState implémentée comme suit:

```

void refuseToDo (Favoriser la faveur) {
    setState () {
        pendingAnswerFavors.remove (faveur);

        refuséFavors.add (favor. copyWith (
            accepté: faux
        ));
    });
}

```

[184]

Épisode 203

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Comme vous pouvez le noter, la méthode `setState()` est utilisée ici pour notifier le framework à reconstruire les widgets intéressés. À l'intérieur de son rappel, nous supprimons la faveur de la liste en attente et ajoutez-en une version modifiée à la liste des refus. La version modifiée est obtenue par faire une copie de la faveur originale et changer sa propriété acceptée. C'est ainsi que le

La méthode `copyWith` de la classe `Favor` regarde dans le code:

```
Favoriser la copie avec ({
    String uid,
    Description de la chaîne,
    DateHeure dueDate,
    booléen accepté,
    DateTime terminée,
    Ami ami,
}) {
    retourner faveur (
        uid: uid ?? this.uid,
        description: description ?? cette.description,
        dueDate: dueDate ?? this.dueDate,
        accepté: accepté ?? ceci.accepté,
        terminé: terminé ?? ceci est terminé,
        ami ami ?? cet ami,
    );
}
```

Notez qu'il utilise l'opérateur prenant en charge les valeurs nulles (??) pour créer une nouvelle instance de faveur avec le les valeurs d'origine (si définies), ou celles reçues comme arguments.

La méthode `copyWith()` est très courante dans le monde Flutter, alors essayez de s'y habituer. Il est présent dans de nombreux widgets du framework Flutter et Des classes. Ce n'est pas obligatoire, mais c'est un bon modèle.

Faire la gestion des actions

Voici à quoi ressemble le bouton "Faire" après avoir utilisé la technique précédente:

```
FlatButton (
    enfant: Texte ("Do"),
    onPressed: () {
        FavorsPageState.of (context) .acceptToDo (faveur);
    },
)
```

[185]

Épisode 204

Et la méthode acceptToDo (favor) correspondante se fait comme suit:

```
void acceptToDo (Favoriser la faveur) {
    setState () {
        pendingAnswerFavors.remove (faveur);

        AcceptedFavors.add ( favor . copyWith (Accepté: vrai));
    });
}
```

Comme vous pouvez le voir, c'est presque la même chose que la méthode refuseToDo (); les seules différences sont dans la liste des cibles et le statut accepté.

Les actions **Abandonner** et **Terminer** sont également très similaires aux précédentes ceux. Veuillez consulter les fichiers source joints pour voir à quoi ils ressemblent.

Appuyez sur le bouton Demander une faveur

Lorsque l'utilisateur appuie sur le bouton d'action flottant avec le signe plus en bas de la page, ils devraient voir le widget RequestFavorPage à l'écran:

```
Navigator.of (contexte) .push (
    MaterialPageRoute (
        constructeur: (contexte) => RequestFavorPage (
            amis: mockFriends,
        ),
    ),
);
```

Nous faisons cela en utilisant le widget Navigator, qui affiche un nouveau widget à l'écran.

Pour l'instant, vous pouvez voir que le geste a été géré comme un autre bouton. Consultez le [chapitre](#)

[7. Routage: navigation entre les écrans](#), pour plus de détails sur le fonctionnement de ce widget.

[186]

Épisode 205

L'écran Demander une faveur

L'écran **Demander une faveur** a quelques gestes à gérer:

[187]

Episode 206

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Voici comment fonctionne le processus:

1. Le bouton de *fermeture* est déjà géré par le widget CloseButton, avec le Widget Navigator (ceci est géré en interne pour nous).
2. Le bouton **SAVE** validera les informations d'entrée de l'utilisateur et enverra le favorisez la demande à un ami.

Le bouton de fermeture

Le widget CloseButton est intégré à Navigator. Il fait apparaître le dernier widget poussé à partir de là, revenant à la précédente. Nous n'avons pas à mettre en œuvre un geste dessus. Par en utilisant le Navigateur pour pousser le widget sur l'écran, nous pouvons utiliser le bouton de fermeture pour retirez-le.

Le bouton **SAVE**

Le bouton **ENREGISTRER** sera chargé de valider et d'enregistrer les nouvelles demandes de faveur. L'enregistrement sera couvert dans le [chapitre 8](#), *Plugins Firebase*, lorsque nous parlerons de Firebase l'intégration.

Le widget RequestFavorPage doit également être converti en StatefulWidget, car nous aurons besoin de conserver des informations et de manipuler les nouvelles demandes de faveur avec des actions. Ce sera l'endroit où nous stockerons la faveur plus tard sur Firebase.

Encore une fois, nous utilisons cela pour centraliser toutes les actions liées aux faveurs, qui sont peu nombreuses dans notre application. L'architecture d'application telle que MVP, MVVM ou BloC pourrait être la solution pour une vraie application.

Validation de l'entrée à l'aide du widget Formulaire

Nous devons ajouter le widget Formulaire à notre layout pour pouvoir valider tous les champs à la fois pendant la sauvegarde. Cela se fait en enveloppant simplement nos widgets de champ de formulaire avec un formulaire widget. Nous définissons également la propriété key de notre formulaire avec une instance GlobalKey (_formKey dans

l'objet State du code suivant) afin que nous puissions l'utiliser plus tard dans la méthode save ():

```
La classe RequestFavorPageState étend l'état <RequestFavorPage> {
    _formKey final = GlobalKey <FormState>();
```

```
@passer outre
Construction du widget (contexte BuildContext) {
```

[188]

Épisode 207

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

```
// retourne la sous-arborescence du widget enveloppée dans un formulaire, caché par souci de brièveté.
}
}
```

La méthode save () ressemble aux précédentes:

```
FlatButton (
    enfant: texte ("SAVE"),
    textColor: Colors.white,
    onPressed: () {
        RequestFavorPageState.of (contexte) .save (); // nous pourrions appeler save ()
        // méthode directement comme nous
        // sont dans la même classe.
        // Laissé intentionnellement pour
        // exemplification.
    },
)
```

Il recherche dans l'arborescence l'état correspondant et lui demande de sauvegarder. La méthode save () fait le travail dur:

```
void save () {
    if ( _formKey .currentState.validate ()) {
        // stocke la demande de faveur sur Firebase
        Navigator.pop (contexte);
    }
}
```

OK, pour le moment, ça ne fait pas grand chose; il n'appelle que la validation du formulaire correspondant qui parcourt tous les champs du formulaire et les valide, comme vous le savez déjà.

Vérifiez les fichiers de code source attachés au chapitre pour vérifier la validation code pour les champs du formulaire.

[189]

Épisode 208

Gestion des entrées et des gestes de l'utilisateur

Chapitre 5

Sommaire

Dans ce chapitre, nous avons vu comment fonctionne la gestion des gestes dans le framework Flutter, ainsi que avec les méthodes de gestion des gestes, telles que toucher, toucher deux fois, panoramique et zoom, pour exemple. Nous avons vu quelques widgets qui utilisent GestureDetector seul pour gérer gestes. Nous avons également vu comment utiliser les widgets Form et FormField pour gérer l'entrée de données utilisateur.

Enfin, nous avons saisi notre projet et apporté quelques ajouts à la gestion des événements du actions sur les faveurs, ce qui nous a aidés à rendre l'application plus interactive.

Dans le prochain chapitre, nous apprendrons comment ajouter de la couleur à nos widgets, utiliser des thèmes, et entrez dans des applications plus pratiques des widgets Material Design et Cupertino en rendre notre application favors plus attrayante.

[190]

Épisode 209

6

Thème et style

La création d'interfaces utilisateur avec des thèmes et des styles intégrés donne à une application un aspect professionnel et facile à utiliser. De plus, le framework permet la création de créations personnalisées et uniques thèmes et styles. Pour ce faire, vous apprendrez à personnaliser l'apparence d'une application en ajouter des polices personnalisées, utiliser des thèmes et explorer les célèbres normes de plate-forme, iOS Cupertino et Google Material Design. De plus, vous verrez comment utiliser les médias requêtes pour un style dynamique.

Chaque application doit avoir sa propre identité. Notre application Favors, par exemple, doit avoir la sienne couleurs et styles. Savoir appliquer des styles, des couleurs et des polices personnalisées est fondamental pour réaliser cela dans n'importe quelle application.

Les sujets suivants seront traités dans ce chapitre:

Widgets de thème
Conception matérielle
iOS Cupertino
Utilisation de polices personnalisées
Style dynamique avec MediaQuery et LayoutBuilder

Widgets de thème

Développer une application va au-delà des bonnes fonctionnalités. C'est aussi l'expérience utilisateur que propose l'application.

La composabilité des widgets Flutter aide dans cette partie du développement. En définissant single responsabilité de chaque type de widget, nous pouvons choisir de définir les thèmes et le style qui s'appliquent à un seul widget, à tous les widgets d'un sous-arbre ou à l'ensemble de l'application.

Épisode 210

Thème et style

Chapitre 6

En utilisant le widget Thème, nous pouvons personnaliser l'apparence et la convivialité d'une application avec des couleurs personnalisées pour le texte, les messages d'erreur, les surlignages et les polices personnalisées. Flutter aussi utilise ce widget dans ses propres widgets. MaterialApp est un excellent exemple de la façon dont le les widgets internes du framework sont composés: il utilise le widget Thème en interne pour personnaliser l'apparence des widgets basés sur la conception de matériaux tels que les AppBars et les boutons. Allons vérifier Découvrez comment utiliser les widgets Thème dans la pratique pour appliquer différents styles à d'autres widgets Flutter.

Widget de thème

Dans Flutter, tout est un widget, et nous pouvons construire l'interface utilisateur en ajoutant widgets utilisant les propriétés enfant et enfants de chaque widget. Le widget Thème se comporte comme n'importe quel autre, il définit des propriétés et peut avoir un enfant.

Le widget Thème fonctionne également avec la technique InheritedWidget, de sorte que chaque widget peut y accéder en utilisant Theme.of(context), qui appelle en interne le helper inheritFromWidgetOfExactType de la classeBuildContext. C'est comment les widgets Material Design utilisent le widget Thème pour se styliser:

[192]

Épisode 211

*Thème et style**Chapitre 6*

Ainsi, les données de thème sont appliquées aux widgets décroissants mais peuvent être remplacées dans les parties locales de l'arborescence des widgets. Dans le diagramme précédent, le thème avec le numéro 2 remplacera le thème avec le numéro 1 défini au tout début de l'arbre. Le sous-arbre numéro 2 aura un thème différent du reste de l'arbre.

De plus, avec cette structure, il est possible de créer un nouveau thème complet pour certains widgets, ou pour hériter d'un thème de base et ne modifier que certaines propriétés pour affecter le sous-arbre.

Lorsque vous créez des widgets avec iOS Cupertino, il existe également le CupertinoTheme et CupertinoThemeData équivalents à Theme et ThemeData respectivement, qui sont présents dans le widget Material Design suite.

La classe ThemeData aide le widget Thème dans la tâche de style. Voyons cela en détails.

ThèmeDonnées

Le widget Thème contient une propriété appelée data, qui accepte une valeur ThemeData qui contient toutes les informations sur le style, la luminosité du thème, les couleurs, la police, etc.

Lors de la rédaction de ce livre, des alternatives aux directives iOS Cupertino sont en cours de développement et ne sont pas présents dans le canal stable de Flutter. (Le code de ce livre utilise le canal stable.)

En utilisant les propriétés de la classe ThemeData, vous serez en mesure de personnaliser toutes les applications. styles associés, tels que les couleurs, la typographie et des composants spécifiques. Vous pouvez choisir de suivre les directives de conception matérielle de Google qui ciblent la conception d'applications pour mobile, Web et appareils de bureau ou iOS Cupertino spécifiques à la plate-forme Apple.

Lors de la création de thèmes, vous pouvez choisir de suivre les directives de conception matérielle de Google qui cible la conception d'applications pour les appareils mobiles, Web et de bureau ou iOS Cupertino qui sont spécifiques à la plate-forme Apple. Les deux directives de conception ont des particularités en raison des plates-formes cibles et fait des recherches à ce sujet. Le choix de suivre ou non le Material Design, iOS Cupertino ou aucun d'entre eux n'est le vôtre, Flutter a des widgets basés sur des thèmes conçus pour les deux, donc vous pouvez appliquer les directives avec précision ou concevoir à votre manière.

Nous explorerons les directives de Material Design et iOS Cupertino plus loin dans les sections suivantes.

[193]

Épisode 212

*Thème et style**Chapitre 6*

La coloration est un sujet important dans la thématisation des widgets. Donnez suffisamment de contraste aux textes sur arrière-plans, ou pour mettre en valeur certains éléments de l'interface utilisateur, par exemple, nécessitent l'utilisation correcte de couleurs. **La luminosité** est l'une des propriétés clés de la classe ThemeData qui aide à manipuler les couleurs, voyons voir.

Luminosité

Une propriété importante d'un thème est la luminosité. La définition de cette propriété est aussi importante comme définissant les couleurs du thème, comme son nom l'indique, il expose la luminosité de l'application thème. Avec cette propriété, les cadres peuvent déterminer le texte, les boutons, les couleurs de surbrillance pour faire un contraste suffisant entre le contenu d'arrière-plan et de premier plan.

C'est ce que la classe ThemeData docs (<https://api.flutter.dev/f>

ThemeData . Clas t:

"La luminosité du thème général de l'application. Utilisé par des widgets tels que des boutons pour déterminez la couleur à choisir lorsque vous n'utilisez pas la couleur principale ou d'accentuation. "

Cela permet de contraster entre le texte, les boutons et l'arrière-plan des matériaux (avec Material Conception de widgets). La classe ThemeData a un constructeur de secours () qui renvoie une lumière thème via la valeur Brightness.light. Vous pouvez utiliser ses constructeurs dark () et light () pour l'essayer vous-mêmes.

Lors du choix des couleurs primaires et d'accentuation, il est important d'expérimenter primaryColorBrightness et accentColorBrightness correspondants. Battement estime la luminosité sur la base de certains calculs de la luminosité des couleurs, mais c'est toujours bon à expérimenter et à vérifier.

De nombreuses autres propriétés ThemeData se rapportent directement au style, c'est pourquoi nous ne les explorons pas davantage. N'hésitez pas à consulter tous les propriétés disponibles dans la classe ThemeData sur <https://docs.flutter.io> flottement / me

Maintenant, plongeons dans une thématisation.

[194]

Épisode 213

Thème et style

Chapitre 6

Theming en pratique

Le style des widgets dans Flutter peut être effectué de plusieurs manières, et tout ce qui concerne les styles est basé sur un widget Thème. Il est temps de voir comment cela fonctionne. Disons, par exemple, que nous avons une application simple, comme suit:

```
class MyAppDefaultTheme étend StatelessWidget {
  @passer outre
  Construction du widget (contexte BuildContext) {
    Conteneur de retour (
      couleur: Colors.red,
      enfant: Centre (
        enfant: Texte (
          "Texte simple",
          textDirection: TextDirection.ltr,
        ),
      ),
    );
  }
}
```

Comme vous pouvez le voir, nous utilisons simplement un widget Container comme widget racine sans un widget Thème. Donc, nous pouvons supposer que nous n'avons aucun style appliqué à son descendant widgets. En outre, la propriété textDirection est nouvelle à ce stade. Lors de l'utilisation le widget MaterialApp dans notre mise en page, il fournit une valeur textDirection par défaut pour nous implicitement. Plus d'informations à ce sujet dans la section suivante.

Nous pouvons, par exemple, utiliser le widget Thème pour changer le style d'un widget Texte. le

La classe ThemeData contient la propriété textTheme, qui contient le style Text

configuration suivant les directives de conception des matériaux:

```
Texte(
    "Texte simple",
    textDirection: TextDirection.ltr,
    style: Theme.of(context).textTheme.display1,
),
```

[195]

Épisode 214

Thème et style

Chapitre 6

La propriété style du widget Text accepte une valeur TextStyle que nous pouvons obtenir le widget Thème. Cependant, comme vous vous en souvenez peut-être, nous n'avons pas spécifié de widget Thème dans notre arborescence d'applications. Dans l'exemple précédent, cela fonctionne car, la méthode Theme.of renvoie une solution de secours pour le widget ThemeData lorsque celle-ci n'est pas définie. Si vous exécutez le code, vous verrez que le widget Texte est affiché avec une taille de police plus grande que la taille par défaut, ceci car nous utilisons le style display1 de Material Design.

Nous pouvons également personnaliser le style; voici un exemple:

```
class MyAppCustomTheme étend StatelessWidget {
    @passer autre
    Construction du widget (contexte BuildContext) {
        Conteneur de retour (
            couleur: Colors.blue,
            enfant: Centre (
                enfant: Thème (
                    data: Thème.of(context).copyWith (
                        textTheme: Thème.of(context).textTheme.copyWith (
                            display1: TextStyle (
                                couleur: Colors.yellow,
                            ),
                        ),
                    ),
                    enfant: Texte (
                        "Texte simple",
                        textDirection: TextDirection.ltr,
                        style: Theme.of(context).textTheme.display1,
                    ),
                ),
            ),
        );
    }
}
```

Dans ce cas, nous ajoutons un widget Thème juste avant le widget Texte et le personnalisons en utilisant sa méthode copyWith:

Nous faisons une copie du widget Thème par défaut à partir de l'application et ne modifions que son propriété textTheme. La fonction copyWith n'est pas obligatoire, cependant, elle est vu très souvent lors du développement d'applications Flutter, alors habituez-vous! Comme avant, cette fois, nous faisons une copie de textTheme à partir du thème de base et modifiez uniquement sa propriété display1 en un nouvel objet de style de texte.

[196]

Épisode 215

Thème et style

Chapitre 6

Nous nous attendons à voir un texte jaune, mais ce n'est pas ce que nous voyons, non? C'est parce que nous sommes en utilisant le paramètre de contexte à partir du niveau de l'arborescence racine, qui recherchera l'arborescence et pas trouver une instance de thème, renvoyant la solution de secours, comme nous l'avons vu dans notre premier exemple. Faire ce travail, nous pouvons utiliser le widget Builder, qui déléguera la construction du widget Texte:

```
Constructeur(  
    constructeur: (contexte) => Texte (  
        "Texte simple",  
        textDirection: TextDirection.ltr,  
        style: Theme.of(contexte).textTheme.display1,  
    ),  
)
```

Cela fonctionne car le widget Builder délègue la construction pour qu'elle se produise à un niveau inférieur du tree, en passant son instance de contexte au niveau inférieur, qui trouvera le bon

Instance de thème lors de la recherche dans l'arborescence. Ainsi, lorsque nous exécutons le code précédent, le texte widget est affiché avec le bon style display1, qui est presque le même que le style de texte par défaut, seule sa couleur est différente, maintenant jaune.

Les exemples précédents ont été définis dans différentes classes d'applications. Tu peux trouver le code source de themes / lib / main.dart sur GitHub et essayez-le pour vous-même en commentant la fonction runApp précédente et décommentez celui que vous souhaitez tester.

Comme le thème fait référence au style d'application, nous devons toujours nous soucier de la plate-forme sous-jacente l'application est en cours d'exécution, voyons comment la classe Platform peut vous aider.

Classe de plateforme

Lors du développement d'applications mobiles pour plusieurs plates-formes, nous pouvons avoir besoin de créer des conceptions pour différentes cibles. Pour ce faire, nous pouvons utiliser la classe Platform, qui nous aide à obtenir informations sur l'environnement, principalement le système d'exploitation cible, via ses getters:

```
isAndroid  
isFuchsia  
isIOS  
isLinux  
isMacOS  
isWindows
```

[197]

Épisode 216

Thème et style

Chapitre 6

Avec ces getters, nous pouvons faire en sorte que tout notre arbre de widgets ait des implémentations spécifiques pour chaque plateforme. Voici un exemple:

```
// fait partie de l'exemple de thème / lib / main.dart

La classe PlatformSpecificWidgets étend StatelessWidget {  
    @passer autre  
    Construction du widget (contexte BuildContext) {  
        retour Platform.isAndroid  
            ? MaterialApp (  
                thème: ThemeData (primaryColor: Colors.grey),  
            )  
            : CupertinoApp (  
                thème: CupertinoThemeData (primaryColor:
```

```

        CupertinoColors.lightBackgroundGray),
    );
}

```

Comme vous pouvez le voir, en fonction de la *plate-forme cible*, nous changeons notre widget d'application (et notre thème également) vers MaterialApp et ThemeData (pour Android) ou CupertinoApp et CupertinoThemeData pour toute autre plate-forme cible.

Consultez le site Web de la documentation <https://docs.flutter.io>

[flutter / classe](#)
[classe impor](#)

Nous avons vu comment utiliser le widget Thème et les classes d'assistance, comme ThemeData et Platform classe pour appliquer des styles à nos widgets. Les directives de conception matérielle et iOS Cupertino sont présent dans la base de nombreux widgets dans Flutter. Voyons ses fondamentaux pour pouvoir suivre ces spécifications efficacement.

Conception matérielle

La conception matérielle est les directives de conception de Google pour aider les développeurs à créer des expériences numériques. Présent dans Flutter et évolue toujours avec la plateforme avec le ajout de nouveaux widgets qui suivent les spécifications des composants de Material Design.

L'importance des styles de conception matérielle pour la plate-forme Flutter est évidente. Il y a déjà une partie des lignes directrices de conception matériel site dédié à ce (<https://material.io/>)

[198]

Épisode 217

Thème et style

Chapitre 6

Les principaux widgets Material Design de Flutter sont MaterialApp et Scaffold. Les deux aident aux développeurs de concevoir une application en suivant les directives de Material Design sans trop travail.

Si vous voulez savoir ce qu'est la conception matérielle en détail, veuillez vérifier en dehors <https://material.io/>

Le premier widget de base pour appliquer les directives Material dans les applications Flutter est MaterialApp widget. Voyons comment cela fonctionne en détail.

Widget MaterialApp

Le widget Thème n'est pas le seul moyen de thème une application. Le widget MaterialApp est l'autre seul widget qui accepte également une valeur ThemeData via sa propriété theme. En plus du thème, MaterialApp ajoute des propriétés d'assistance pour la localisation, par exemple, et aussi la navigation entre les écrans, que nous vérifierons dans [Chapitre 7, Routage: navigation Entre les écrans](#).

En ajoutant un widget MaterialApp en tant que widget racine de l'application, vous indiquez votre intention de suivre les directives de conception matérielle, et c'est le but, n'est-ce pas?

Maintenant qu'il sait que vous suivez les directives de Material Design, le cadre sera légèrement différent par rapport au thème par défaut. Dans le code suivant, nous ne spécifiez un style pour notre texte:

```

La classe MaterialAppDefaultTheme étend StatelessWidget {
  @passer outre
  Construction du widget (contexte BuildContext) {
    retour MaterialApp (
      accueil: Container (
        couleur: Colors.white,

```

```

enfant: Centre (
    enfant: Texte (
        "Texte simple",
        // textDirection: TextDirection.ltr, pas besoin
        // maintenant grâce à materialapp
    ),
),
);
}
}

```

[199]

Épisode 218

Thème et style

Chapitre 6

Dans ce code, nous avons déclaré notre intention de suivre les directives de conception matérielle en en ajoutant le widget MaterialApp comme widget racine, cela produira un retour à un pas très style DefaultTextStyle attrayant pour avertir le développeur qu'il n'utilise pas Material Design efficacement dans les widgets de texte. Le résultat du code précédent est comme suit:

[200]

Épisode 219

En d'autres termes, nous devrions toujours envelopper les widgets de texte dans certains éléments basés sur la conception de matériaux. widget pour appliquer correctement les styles de typographie proposés par les directives. Le widget Matériel est l'exemple le plus simple; il a une propriété `DefaultTextStyle` et d'autres propriétés de conception de matériaux typiques, telles que l'élévation de l'effet d'ombre du des lignes directrices.

Notez que nous n'avons pas non plus fourni de propriété `textDirection` du widget `Text` cette fois. L'une des fonctions de `MaterialApp` est de nous permettre d'appliquer l' **internationalisation** à notre application, et `textDirection` est basée sur les paramètres régionaux ambients.

Nous verrons comment travailler avec la localisation dans [Chapter 13, Amélioration de l'expérience utilisateur](#).

En utilisant le widget `MaterialApp`, nous avons vu comment lancer la conception matérielle suivante des lignes directrices. Un autre widget important pour vous aider dans cette tâche est le widget `Scaffold`. Voyons voir comment l'utiliser dans la section suivante.

Widget d'échafaudage

Nous avons vu au [chapitre 4, Widgets: Construire des dispositions dans Flutter](#), que le widget `Scaffold` a propriétés qui aident à construire la mise en page avec un look Material Design. Son but est comme important comme widget `MaterialApp`; il aide le développeur à suivre le Material Design directives en ajoutant simplement les widgets correspondants à ses propriétés. Notre application Favors L'écran principal suit certains aspects de la conception matérielle.

[201]

Épisode 220

Voyons voir:

[202]

Épisode 221

Thème et style

Chapitre 6

Ici, nous avons utilisé certains composants de Material Design ainsi que le widget Scaffold.

Certaines des pièces utilisées sont les suivantes:

La *barre d'application* affichée en haut de l'application contient généralement un titre et un contexte utilisateur des actions telles que des filtres ou des paramètres. Dans cet exemple, via la propriété `appbar` du widget Scaffold, nous montrons un widget AppBar qui a un titre et un

TabBar pour afficher les onglets.

Le *bouton d'action flottant* est l'un des composants les plus connus de Material Design; c'est un bouton rond flottant généralement affiché dans le coin inférieur droit de l'écran. Dans cet exemple, il contient l'action principale de l'application, **Demander une faveur**, suivant les directives de conception des matériaux.

Maintenant que nous avons vu à quoi ressemble le thème par défaut dans certains widgets. Voyons comment créer notre propre thème personnalisé avec les couleurs de notre choix.

Thème personnalisé

Jusqu'à présent, notre application Favors n'a utilisé aucune propriété de Theme ou ThemeData. Il est temps de personnalisez le style de l'application pour la rendre plus attrayante. Voilà à quoi ça va ressembler après avoir refactorisé ses styles:

Votre page de faveurs (Ceci est une image de votre page de faveurs. Les autres informations (floues) ne sont pas importantes ici)

[203]

Épisode 222

Thème et style

Chapitre 6

Nous allons commencer par créer une définition lightTheme personnalisée. Nous avons peu de façons de colorer notre application: nous pouvons définir des couleurs personnalisées pour chacune des propriétés de couleur de la classe ThemeData (il a des propriétés pour chacun des widgets disponibles de Material Design, tels que des cartes ou des boutons). La clé est d'expérimenter les propriétés de couleur et les directives.

N'oubliez pas que vous pouvez toujours remplacer la définition du thème de l'application dans le arborescence de widgets (avec une couleur de carte différente, par exemple) en l'enveloppant dans un autre widget de thème.

Maintenant, définissons ThemeData:

```
final lightTheme = ThemeData (
    primarySwatch: Colors.lightGreen,
    primaryColor: Colors.lightGreen.shade600, // pas nécessaire lorsque
                                                // primarySwatch est défini
                                                // comme ci-dessus
    accentColor: Colors.orangeAccent.shade400,
    primaryColorBrightness: Brightness.dark,
    cardColor: Colors.lightGreen.shade100,
);
```

Nous avons défini un nouveau widget ThemeData qui est clair par défaut, et nous avons a modifié sa propriété primarySwatch. Nous avons utilisé une couleur basée sur la palette Matériel, où nous pouvons définir certaines couleurs et l'ensemble du schéma sera dérivé de cet échantillon. Bien que le thème par défaut soit clair (fond clair / textes sombres), nous avons défini primaryColorBrightness à Brightness.dark pour que le texte apparaissant au-dessus d'un l'arrière-plan est blanc par défaut.

Notez également que nous avons défini le thème dans un nouveau fichier Dart pour vous aider avec le code organisation. Nous devons donc l'importer pour pouvoir l'utiliser dans notre application:

```
retour MaterialApp (
    thème: lightTheme,
    accueil: FavorsPage (),
);
```

Comme vous pouvez vous y attendre, l'application utilise le lightTheme importé via sa propriété de thème.

Pour la définition du schéma de couleurs de l'application, nous avons utilisé la couleur outil du site Web de Material Design. Jetez un oeil à l' [adresse https://io.material.io](https://io.material.io) matériaux / couleurs. Un autre conseil: si vous utilisez mac os pour le débogage, l'outil de thèmes de matériaux peut vous aider à créer votre propre thème. Check it out à <https://io.material.io>

[204]

Épisode 223

Thème et style

Chapitre 6

Changer les couleurs ne suffit pas pour changer l'apparence de l'application. Une autre chose que nous pouvons faire consiste à modifier les styles de texte et à utiliser les styles Material Design. Comme nous l'avons vu auparavant, c'est fait à l'aide de la propriété style des widgets Texte. Donc, après avoir fait quelques changements, notre

Les cartes de faveur pourraient mettre en valeur certaines parties du texte.
Dans les en-têtes de la liste, par exemple, nous avons ajouté un style pour l'agrandir:

```
final TextStyle = Thème.of(context).textTheme.title;
```

Nous obtenons le style titleStyle du thème de l'application et l'appliquons directement au texte widget:

```
Texte(  
    Titre,  
    style: titleStyle,  
)
```

La même chose est faite pour les autres widgets de texte de l'application. Comme vous pouvez le voir dans notre application Favors Par exemple, il est facile de modifier les styles de nos widgets en utilisant le widget Thème et les classes d'assistance. Tu pouvez consulter le code source de ce chapitre sur GitHub pour plus de détails, et nous vous encourageons vous expérimentez avec certaines valeurs pour la pratique.

Maintenant que vous connaissez les bases de la conception matérielle, présentons iOS Cupertino pour avoir quelques informations comparatives.

iOS Cupertino

L'objectif de rendre une application native est important sur Flutter. Dans cet esprit, un beaucoup d'efforts sont faits pour amener le côté Cupertino du cadre au même niveau de couverture comme côté Material Design. Lors de la rédaction de ce livre, de nombreux Cupertino des widgets ont été ajoutés au framework.

L'idée est que leur comportement est fidèle aux applications natives, ce n'est donc pas une tâche facile. Le la communauté joue un rôle important dans cette tâche en utilisant les composants et en donnant retour d'information.

Comme les widgets Material Design, CupertinoApp, CupertinoPageScaffold, et CupertinoTabScaffold sont les principaux widgets de Cupertino disponibles dans Flutter.

Nous n'entrons pas dans CupertinoPageScaffold et les widgets CupertinoTabScaffold en détails ici. Tu peux vérifier ceux-ci et tous les widgets Cupertino disponibles en détail <https://flutter.io/>

<https://flutter.io/>

[205]

Épisode 224

Thème et style

Chapitre 6

L'alternative iOS Cupertino au widget MaterialApp est le widget CupertinoApp, voyons ses propriétés clés et comment il se compare au widget MaterialApp.

CupertinoApp

CupertinoApp se comporte de la même manière pour Cupertino que MaterialApp pour Material Conception. Il ajoute des fonctionnalités et des installations frappantes permettant au développeur de suivre la conception modèles de Cupertino. Par exemple, l'application utilise, par défaut, un parchemin rebondissant c'est typique sous iOS, une police personnalisée différente d'Android, et plus encore.

En plus du thème, CupertinoApp ajoute des propriétés d'assistance pour la localisation, par exemple, et aussi la navigation entre les écrans, que nous explorerons au [chapitre 7, Routage: navigation Entre les écrans](#).

Cela fonctionne de la même manière que Material Design. Nous pouvons choisir d'utiliser CupertinoApp ou non. Ainsi, nous pouvons toujours utiliser les widgets CupertinoTheme et CupertinoThemeData, le de la même manière que nous le ferions pour le Material Design. Ce qui change de l'un à l'autre la pratique sont ses propriétés disponibles.

Comme cela est très similaire à la section précédente; nous n'allons pas entrer dans détail ici. Vous pouvez expérimenter avec les thèmes et jeter un oeil à la ci-joint le dossier `cupertino_theme` pour de petits exemples d'utilisation.

Bien que ce ne soit pas recommandé, nous pouvons toujours tout mélanger dans le code, en faisant certaines parties suivez Material Design et certains suivent Cupertino. Nous pouvons créer deux classes d'applications, une pour Material Design et un pour Cupertino. Nous pouvons même créer une classe d'application générique qui change la disposition du widget basée sur la plate-forme (la classe Platform).

Expérimetons certains des widgets iOS Cupertino dans notre application Favors.

Cupertino en pratique

Notre application Favors est conçue pour utiliser des composants Material Design, mais nous pouvons choisir de faire il semble plus natif dans iOS en utilisant les widgets Cupertino. Cela peut être fait avec une combinaison de conditions lors de la construction de nos widgets à l'aide de la classe Platform.

[206]

Épisode 225

Thème et style

Chapitre 6

Nous pouvons, par exemple, concevoir une alternative à Cupertino pour le premier écran des faveurs:

Épisode 226

Thème et style

Chapitre 6

Comme vous pouvez le voir, dans la variante iOS Cupertino, nous avons la barre de navigation en bas de l'écran à la place. OK, cela n'a pas l'air très bien, mais l'important est l'idée de créer des mises en page personnalisées en fonction de la plate-forme cible. Flutter vous donne les outils; vous devez les utiliser correctement.

Vous pouvez trouver le code source de l'exemple hands_on_cupertino_theme sur GitHub pour voir toutes les conditions et modifications apportées pour utiliser Cupertino widgets. La partie thème est omise car elle fonctionne de manière très similaire à Conception matérielle.

Nous devons vérifier la plate-forme cible et créer différents widgets en fonction de celle-ci. Cela peut être assez compliqué, donc une alternative consiste à développer des classes de widgets séparées pour chaque plate-forme et de ne pas mélanger tout le code, ce qui aide à l'organisation.

Dans notre exemple, nous n'avons fait que le premier écran afin d'illustrer comment l'arbre peut être conditionné en fonction de la plate-forme. L'application Favors aura le même style sur les deux plates-formes. Voyons maintenant comment utiliser des polices personnalisées pour donner une orientation de marque aux applications.

Utilisation de polices personnalisées

Material Design et Cupertino fournissent de bonnes polices pour la conception d'applications, mais parfois il est utile de changer la police par défaut en une police plus axée sur la marque / le produit.

Comme la police est spécifiée dans le widget Thème, nous pouvons l'ajouter au thème de l'application racine, et puis il est appliquée à l'ensemble de l'application. Si vous préférez spécifier la police par widget, c'est aussi possible. La première étape pour utiliser une police personnalisée dans les applications Flutter consiste à importer des fichiers de polices dans le projet.

Importation de polices dans le projet Flutter

Cette fois, nous utiliserons directement notre application Favors pour montrer l'exemple. Nous serons l'importation et l'utilisation d'une police personnalisée comme police par défaut pour l'ensemble de l'application.

Pour ce faire, nous pouvons mettre les fichiers de polices dans un sous-répertoire de projet et plus tard, déclarer ces polices fichiers dans le fichier pubspec.yaml. Dans cet exemple, nous utiliserons les polices Ubuntu trouvées sur le site Web de Google Fonts.

Épisode 227

Thème et style

Chapitre 6

Découvrez les différentes polices disponibles sur <https://fonts.google.com>

La première étape consiste à ajouter les fichiers au répertoire du projet. Il est courant de mettre la police fichiers dans un sous-répertoire polices / ou assets / du projet Flutter. Ici, nous utiliserons un fonts / répertoire:

Après cela, nous devons déclarer les actifs de police dans le fichier pubspec.yaml afin que le framework saura où trouver la police souhaitée lors de la mise en forme du texte:

```
// Fichier pubspec.yaml - le code source complet se trouve dans hands_on_fonts
dossier d'exemple
// .. masqué par souci de concision
battement:
    utilise-matière-conception: vrai
polices:
    - famille: Ubuntu
        polices:
            - atout: fonts / Ubuntu-Regular.ttf
            - atout: fonts / Ubuntu-Italic.ttf
                style: italique
            - atout: fonts / Ubuntu-Medium.ttf
                poids: 500
            - atout: fonts / Ubuntu-Bold.ttf
                poids: 700
```

[209]

Épisode 228

Thème et style

Chapitre 6

Comme vous pouvez le voir, nous avons défini la police en quelques sections:

Le champ de famille nomme la police dans le contexte du cadre. Il n'a pas besoin de correspondant aux noms de fichiers de police. Il sera utilisé pour y faire référence dans le code, alors soyez cohérent.

Après cela, nous avons un champ de polices suivi d'une liste de champs d'actif du police importée. Tous les actifs spécifiés seront inclus dans l'actif de l'application paquet. Nous devons spécifier chaque actif avec des détails correspondant à son style:

weight: Ceci détermine le poids de la police dans l'élément. Il correspond aux valeurs d'énumération FontWeight appliquées pendant mise en page, spécifiez-le correctement.

style: Ceci détermine si le fichier d'actif correspond à un contour de police normal ou une variante en *italique*. Ces valeurs correspondent à l'énumération FontStyle.

Consultez la documentation pour savoir comment spécifier le poids et propriétés de police. Vous pouvez consulter la documentation de la classe `FontWeight`, `FontStyle`, `FontVariants` et `FontFeatureVariants`.

Après avoir importé la police dans le projet, appliquons la police à nos widgets Texte.

Remplacer la police par défaut dans l'application

L'étape suivante consiste à rendre la police active dans l'application. Nous pouvons le faire dans le thème racine dans un widget MaterialApp et CupertinoApp, ou, si nous préférons, nous pouvons ajouter une police directement à un widget Texte via sa propriété style:

```
final lightTheme = ThemeData (
    fontFamily: "Ubuntu",
    primarySwatch: Colors.lightGreen,
    primaryColor: Colors.lightGreen.shade600,
    accentColor: Colors.orangeAccent.shade400,
    primaryColorBrightness: Brightness.dark,
    cardColor: Colors.lightGreen.shade100,
);
```

Notre application utilise désormais la famille de polices Ubuntu par défaut dans tous les widgets contenant du texte. N'oubliez pas que ce comportement peut être remplacé dans de petites sections de l'application, si vous préférez, par en utilisant les widgets Thème ou en modifiant directement la propriété de style des widgets Texte.

[210]

Épisode 229

Thème et style

Chapitre 6

Si vous essayez d'utiliser une variante d'épaisseur en gras d'une famille de polices personnalisée qui n'est pas déclaré dans le fichier pubspec.yaml, le framework utilisera le plus génériques pour la police et tentera d'extrapoler les contours pour le poids et style demandés.

Comme vous pouvez le voir, appliquer une police personnalisée à l'ensemble de l'application en important simplement le police souhaitée et en la déclarant dans le projet. Un autre aspect important de la thématisation et le style consiste à adapter les mises en page pour différents appareils. Widgets MediaQuery et LayoutBuilder peut vous aider dans cette tâche, voyons voir.

Style dynamique avec MediaQuery et LayoutBuilder

L'adaptation d'une mise en page à une plate-forme peut nous aider à répondre à un public plus large. Mais un autre chose à réaliser est le nombre massif d'appareils différents, ce qui pose d'autres défis développeur.

Développer pour prendre en charge plusieurs tailles d'écran est un défi qui sera toujours présent dans le vie d'un développeur, nous avons donc besoin de mécanismes pour nous adapter au mieux à cela. Flutter, encore une fois, nous donne les outils dont nous avons besoin pour connaître l'écosystème que l'application exécute afin que nous peut agir en conséquence.

Pour vous aider dans cette tâche, les principales classes fournies par Flutter sont LayoutBuilder et MediaQuery.

LayoutBuilder

Le widget LayoutBuilder fournit une propriété de générateur de LayoutWidgetBuilder type. Bien qu'il soit similaire au widget Builder, LayoutWidgetBuilder est livré avec informations supplémentaires sur la taille du widget parent à l'intérieur d'une valeur BoxConstraints.

Avec ces informations, la méthode de construction peut être modifiée en fonction de l'espace disponible. Ainsi, dans différents appareils, il y aura une quantité d'espace disponible différente dans le widget racine de l'arbre, ce qui peut également limiter la taille de ses enfants. En utilisant ce widget, nous pouvons choisir s'il faut ou non afficher certaines parties de la mise en page.

Ce widget dépend de la taille du widget parent, il est donc reconstruit à chaque fois que la taille changements. Cela peut se produire de différentes manières sur les appareils mobiles. L'exemple le plus simple est celui où l'orientation de l'application change, c'est-à-dire lorsque l'utilisateur fait pivoter le téléphone.

[211]

Épisode 230

Thème et style

Chapitre 6

Voyons comment réagir à un changement de taille à l'écran. Dans cet exemple, nous allons modifier la façon dont deux widgets sont affichés en fonction de l'espace disponible. Ainsi, les widgets sont affichés les uns sur les autres lorsqu'il n'y a pas assez de place pour eux (on évalue ceci en utilisant l'instance BoxConstraints donnée par le widget LayoutBuilder), ou side-by-côté quand il y a plus d'espace disponible (comme en position paysage):

```
class MyApp étend StatelessWidget {
  @passer autre
  Construction du widget (contexte BuildContext) {
    retour MaterialApp (
      accueil: LayoutBuilder (
        générateur: (contexte BuildContext, contraintes BoxConstraints) {
          // construire la mise en page en fonction des valeurs de contraintes
        }
      )
    )
  }
}
```

Comme vous pouvez le voir, nous avons ajouté un widget LayoutBuilder, et nous pouvons créer la mise en page en fonction des contraintes données:

```
if (contraintes.maxWidth <= 500) {
  colonne de retour (
    mainAxisSize: MainAxisSize.max,
    enfants: <Widget> [
      Étendu(
        enfant: Conteneur (
          couleur: Colors.green,
          enfant: Centre (enfant: Texte ("1")),
        ),
      ),
      Étendu(
        enfant: Conteneur (
          couleur: Colors.blue,
          enfant: Centre (enfant: Texte ("2")),
        ),
      ),
    ],
  );
}
```

Conditionnellement, nous affichons un widget Colonne lorsque la largeur disponible est inférieure à 500. Et quand on a assez de place on change le widget retourné:

```
return Row (
  mainAxisSize: MainAxisSize.max,
```

[212]

Épisode 231

Thème et style

Chapitre 6

```
enfants: <Widget> [
  Étendu(
    enfant: Conteneur (
      couleur: Colors.yellow,
      enfant: Centre (enfant: Texte ("1")),
    ),
  ),
  Étendu(
    enfant: Conteneur (
      couleur: Colors.violet,
      enfant: Centre (enfant: Texte ("2")),
    ),
  ),
];
```

Dans ce cas, nous retournons un widget Row, car nous avons suffisamment d'espace (supérieur à 500). Voici à quoi cela ressemble dans différentes orientations:

[213]

Épisode 232

Thème et style

Chapitre 6

Comme vous pouvez le voir, nous apportons des modifications à la mise en page en fonction non seulement de l'orientation, mais aussi de la largeur disponible. Une autre façon de répondre aux changements de taille disponible consiste à utiliser la classe MediaQuery. Voyons maintenant comment fonctionne l'alternative MediaQuery.

MediaQuery

MediaQuery est un descendant d'InheritedWidget qui contient des informations sur la taille de l'écran entier, et pas seulement le widget parent. En tant que widget InheritedWidget, ce fournит également la méthode MediaQuery.of précédemment introduite, qui recherche l'arborescence pour une instance MediaQuery.

Son utilisation est conditionnée par la présence d'une instance dans le contexte. Cela peut être fait facilement en ajoutant une instance de WidgetsApp en tant que widget racine. WidgetsApp n'est pas spécifique à la plate-forme, comme MaterialApp ou CupertinoApp, qui utilisent cette classe dans leur implémentation interne.

Voyons comment utiliser la classe MediaQuery pour créer une mise en page réactive.

Exemple MediaQuery

Jusqu'à présent, notre application Favors n'est pas réactive en termes de taille d'écran. Il affiche une liste verticale de cartes qui remplissent l'espace disponible sur l'écran. Pour les smartphones typiques, cela a l'air bien, mais voici à quoi cela ressemble sur les appareils avec un écran plus grand:

[214]

Épisode 233

Thème et style

Chapitre 6

Comme vous pouvez le voir, chaque carte remplit chaque ligne et elles sont beaucoup plus grandes que nécessaire. nous pouvons changez-le en fonction de la taille de l'écran et faites en sorte que la liste affiche plus d'éléments s'il y a plus d'espace que nécessaire pour afficher une carte.

En utilisant la classe MediaQuery, nous avons effectué un calcul pour modifier le nombre de cartes affichées par ligne:

```
// fait partie du fichier hands_on_mediaquery / lib / main.dart

La classe FavorsList étend StatelessWidget {
    // ... masqué par souci de concision
    @passer autre
    Construction du widget (contexte BuildContext) {
        colonne de retour (
            mainAxisSize: MainAxisSize.max,
            enfants: <Widget> [
                Remboursement(
                    enfant: Texte (
                        Titre,
                        style: titleStyle,
```

[215]

Épisode 234

Thème et style

Chapitre 6

```

),
padding: EdgeInsets.only (haut: 16,0),
),
Étendu(
enfant: _builldCardsList (contexte),
),
],
);
}

```

Dans le code précédent, en enveloppant la liste des faveurs dans un widget développé, tous l'espace disponible du widget Colonne est occupé, et on laisse la logique de redimensionnement avec

MediaQuery à la méthode _buildCardsList () comme suit:

```

const kFavorCardMaxWidth = 450,0; // une largeur maximale de carte

La classe FavorsList étend StatelessWidget {
// ... masqué par souci de concision

Widget _builldCardsList (contexte BuildContext) {
final screenWidth = MediaQuery.of (contexte) .size.width;
cartes finalesPerRow = max (screenWidth ~/ kFavorCardMaxWidth, 1);
// fonction max () du paquet dart: math
if (screenWidth> 400) {
retournez GridView.builder (
physique: BouncingScrollPhysics (),
itemCount: favors.length,
scrollDirection: Axis.vertical,
itemBuilder: (contexte BuildContext, index int) {
faveur finale = favors [index];
return FavorCardItem (faveur: faveur);
},
gridDelegate: SliverGridDelegateWithFixedCrossAxisCount (
childAspectRatio: 2,8,
crossAxisCount: cardsPerRow,
),
);
}
renvoie ListView.builder (
physique: BouncingScrollPhysics (),
itemCount: favors.length,
itemBuilder: (contexte BuildContext, index int) {
faveur finale = favors [index];
return FavorCardItem (faveur: faveur);
},
);
}
}

```

[216]

Épisode 235

Thème et style

Chapitre 6

Pour que le redimensionnement fonctionne correctement, nous avons apporté quelques modifications à FavorCardItem pour rendre la mise en page plus adaptable à la mise en page changements. Vous pouvez trouver le code source de l'exemple hands_on_mediaquery sur GitHub.

Dans le code précédent, vous pouvez voir que nous pouvons diviser la largeur d'écran disponible (prise de MediaQuery.of (context) .size.width) par la largeur maximale souhaitée d'un card (kFavorCardMaxWidth) et enregistrez-le dans la variable cardsPerRow. Plus tard, nous l'utilisons pour Vérifiez s'il y a de la place pour une autre carte d'affilée. Ensuite, s'il y a de la place, nous listons les cartes en utilisant un widget GridView affichant les colonnes cardsPerRow. S'il n'y a pas de place pour plus qu'une seule carte, nous affichons un widget ListView comme auparavant. Voici le résultat:

Il existe d'autres widgets Flutter disponibles pour cette tâche, alors peut-être une meilleure approche serait d'utiliser un conteneur autre qu'une liste pour afficher les cartes d'une manière plus flexible. D'autres classes peuvent aider sur les ajustements de mise en page. Voyons certains d'entre eux dans la section suivante.

[217]

Épisode 236

Thème et style

Chapitre 6

Classes réactives supplémentaires

Il existe quelques autres widgets qui aident à créer des mises en page réactives:

CustomMultiChildLayout vous donne la liberté de choisir comment un ensemble d'enfant les widgets sont disposés à l'écran à l'aide d'un délégué classe: MultiChildLayoutDelegate.

FittedBox change sa taille et sa position enfant en fonction d'un avertissement spécifique. Avoir un Regarder <https://.../de> les valeurs disponibl

AspectRatio tente de forcer la taille de son enfant en fonction d'un aspect spécifique rapport.

En utilisant toutes ces classes disponibles, nous pouvons rendre nos mises en page Flutter adaptatives. Nous sommes capables pour styliser nos widgets et personnaliser l'ensemble de l'application.

Sommaire

La personnalisation des applications en termes de styles est fondamentale pour créer une expérience unique pour le utilisateur et atteindre les objectifs de l'application. Connaitre les classes de framework Flutter qui aident à ce sujet est cruciale pour le développement de toute application, y compris notre application Favors tout au long du livre.

Dans ce chapitre, nous avons vu quelques façons de changer le style de nos applications. En utilisant les widgets Theme et ThemeData nous pouvons spécifier des styles qui changeront tous les widgets en dessous d'eux dans l'arbre. En outre, en utilisant les classes d'application disponibles, MaterialApp et CupertinoApp, nous pouvons changer le style de l'ensemble de l'application d'une manière simple.

Nous avons vu comment ajouter une famille de polices personnalisée à notre application afin de pouvoir modifier le look par défaut de nos textes et étiquettes. Enfin, nous avons vu qu'il est possible de changer la façon dont notre l'application se présente dans différentes tailles ou orientations à l'aide de MediaQuery et LayoutBuilder classes, ou même spécifiquement pour la plate-forme cible en utilisant la classe Platform.

Dans le chapitre suivant, nous allons apprendre comment fonctionne la navigation entre les écrans dans Flutter, et comment utiliser la propriété Navigator pour modifier ce qui est visible par l'utilisateur.

[218]

Épisode 237

sept

Routage: navigation entre Écrans

Les applications mobiles sont généralement organisées sur plusieurs écrans. Dans Flutter, l'itinéraire correspondant à un écran est géré par le widget Navigateur de l'application. Le Navigateur widget gère la pile de navigation, en poussant une nouvelle route ou en passant à la précédente. Dans ce chapitre, vous apprendrez à utiliser le widget Navigator pour gérer les itinéraires de votre application et comment ajouter des animations de transition entre les écrans.

Les sujets suivants seront traités dans ce chapitre:

- Comprendre le widget Navigator
- Comprendre les itinéraires
- Apprendre les transitions
- Explorer les animations de héros

Comprendre le widget Navigator

Les applications mobiles contiennent souvent plus d'un écran. Si vous êtes un Android ou iOS développeur, vous connaissez probablement les classes Activity ou ViewController qui représentent écrans sur ces plates-formes respectivement.

Une classe importante dans la navigation entre les écrans dans Flutter est le widget Navigator qui est responsable de la gestion des changements d'écran avec une idée d'historique logique.

Un nouvel écran dans Flutter est juste un nouveau widget qui est placé devant un autre. C'est géré par le concept de Routes, qui définit la navigation possible dans l'application. Comme toi peut-être déjà deviné, la classe Route est une aide pour Flutter pour travailler sur la navigation workflow.

Épisode 238

Routage: navigation entre les écrans

Chapitre 7

Les principaux acteurs de la couche de navigation sont les suivants:

- Navigator: le gestionnaire d'itinéraire
- Superposition: le navigateur l'utilise pour spécifier les apparences des itinéraires
- Route: un point de terminaison de navigation

Navigateur

Le widget Navigator est le principal acteur dans la tâche de passer d'un écran à un autre. La plupart du temps, nous allons changer d'écran et transmettre des données entre eux, ce qui est une autre tâche importante pour le widget Navigator.

La navigation dans Flutter se fait dans une structure de *pile*. La structure de la pile convient à cette tâche car son concept est très similaire au comportement d'un écran:

Nous avons un élément en *haut de la pile*. Dans Navigator, l'élément le plus haut sur la pile se trouve l'écran actuellement visible de l'application.

Le dernier élément inséré est le premier à être retiré de la pile (généralement arbitré comme **dernier entré, premier sorti (LIFO)**). Le dernier écran visible est le premier supprimé.

Comme stack, les principales méthodes du widget Navigator sont push () et pop () .

Recouvrir

Dans sa mise en œuvre, Navigator utilise le widget Overlay. Ce qui suit est de la Documentation:

"Les superpositions permettent aux widgets enfants indépendants d'apparaître au-dessus d'autres widgets en les insérant dans la pile de la superposition."

La superposition permet à chacun de ces widgets de gérer sa participation à la superposition en utilisant des objets OverlayEntry.

Nous allons passer par quelques étapes pour vérifier que la manière la plus courante d'utiliser un navigateur et sa superposition est avec les widgets d'application, WidgetsApp, MaterialApp et CupertinoApp, qui offrent plusieurs façons de gérer la navigation dans le widget Navigator.

[220]

Épisode 239

Routage: navigation entre les écrans

Chapitre 7

Pile de navigation / historique

Comme vous l'avez peut-être déjà remarqué, la méthode push () ajoute un nouvel écran en haut de la *pile de navigation*. Pop (), à son tour, le supprime de la pile de navigation.

Donc, en résumé, la pile de navigation est la pile d'écrans qui sont entrés en scène grâce à la méthode push () du widget Navigator.

La pile de navigation est également appelée **historique de navigation**.

Route

Les éléments de la pile de navigation sont des itinéraires, et il existe plusieurs façons de les définir dans Battement.

Lorsque nous voulons naviguer vers un nouvel écran, nous y définissons un nouveau widget Route, en plus à certains paramètres définis comme une instance de RouteSettings.

RouteSettings

Il s'agit d'une classe simple qui contient des informations sur l'itinéraire pertinent pour le navigateur

widget. Les principales propriétés qu'il contient sont les suivantes:

name: identifie l'itinéraire de manière unique. Nous l'explorerons en détail dans le prochain section.

arguments: Avec cela, nous pouvons tout transmettre à l'itinéraire de destination.

Vous pouvez vérifier plus d'informations sur cette classe dans les documents: <https://docs.flutter.io/flutter>

[221]

Épisode 240

Routage: navigation entre les écrans

Chapitre 7

MaterialPageRoute et CupertinoPageRoute

La classe Route est une abstraction de haut niveau via la fonction de navigation. Toutefois, nous ne l'utilisera pas directement, comme nous l'avons vu qu'un écran est une route dans Flutter. Différent les plates-formes peuvent nécessiter des changements d'écran pour se comporter différemment. Dans Flutter, il existe des alternatives implémentations de manière adaptative à la plate-forme. Ce travail est effectué avec MaterialPageRoute et CupertinoPageRoute, qui s'adaptent respectivement à Android et iOS. Alors, nous devons décider lors du développement d'une application d'utiliser le Material Design ou iOS Transitions Cupertino, ou les deux, selon le contexte.

Mettre tous ensemble

Il est temps de découvrir comment utiliser le widget Navigator dans la pratique. Créons une base flux pour accéder à un deuxième écran et revenir en arrière. Cela ressemblera à quelque chose comme ceci:

[222]

Épisode 241

Routage: navigation entre les écrans

Chapitre 7

La manière de base d'utiliser un widget Navigator est comme n'importe quel autre - en l'ajoutant à l'arborescence des widgets:

```
La classe NavigatorDirectlyApp étend StatelessWidget {
  @passer autre
  Construction du widget (contexte BuildContext) {
    retour Directionnalité (
      enfant: Navigateur (
        onGenerateRoute: (Paramètres RouteSettings) {
          retourner MaterialPageRoute (
            constructeur: (BuildContext context) => _screen1 (context));
          },
        ),
        textDirection: TextDirection.ltr,
      );
    }
    _screen1 (BuildContext context) {...} // masqué par souci de concision
    _screen2 (BuildContext context) {...} // masqué par souci de concision
  }
}
```

Le widget Directionnalité a été ajouté ici afin que nous puissions afficher

Widgets de texte. N'oubliez pas que WidgetsApp et les variantes gèrent cela et plus pour nous.

Le widget Navigator contient une propriété onGenerateRoute, un rappel qui est responsable de la création d'un widget Route basé sur un objet RouteSettings passé comme un argument.

Dans l'exemple précédent, vous pouvez voir que nous n'avons pas utilisé l'argument settings; au lieu, nous avons renvoyé une route par défaut. L'approche la plus courante vérifierait le nom des paramètres propriété, qui fonctionne comme l'identifiant de l'itinéraire. Le framework utilise le nom '/' comme la route initiale par défaut et fera un appel initial au rappel, en le passant comme un argument. Ainsi, l'exemple précédent utilise le widget renvoyé _screen1 comme élément initial route.

Consultez la section *Routes nommées* plus loin dans ce chapitre pour plus de détails et exemples de noms de routes.

[223]

Épisode 242

Routage: navigation entre les écrans

Chapitre 7

Le résultat du rappel onGenerateRoute est un objet Route. Nous avons utilisé le type MaterialPageRoute ici. Dans sa mise en œuvre la plus élémentaire, nous devrions passer un onGenerateRoute le rappelle également. Il doit renvoyer un widget à afficher en tant que Route. Vous vous demandez peut-être: pourquoi ne pas utiliser une propriété enfant pour ajouter directement le widget enfant? Ses la création dépend du contexte dans lequel elle est construite, car le widget Navigator peut créer ce widget Route dans différents contextes.

Mais si vous vérifiez le code suivant, vous verrez que nous pouvons naviguer d'un écran à un autre en cliquant sur le bouton correspondant. Nous pouvons voir cela dans la méthode _screen1, pour exemple:

```

Widget _screen1 (contexte BuildContext) {
    Conteneur de retour (
        couleur: Colors.green,
        enfant: Colonne (
            mainAxisSize: MainAxisSize.max,
            mainAxisAlignment: MainAxisAlignment.center,
            enfants: <Widget> [
                Texte ("Écran 1"),
                RaisedButton (
                    enfant: Texte ("Aller à l'écran 2"),
                    onPressed: () {
                        Navigator.of (contexte) .push (
                            MaterialPageRoute (
                                constructeur: (contexte BuildContext) {
                                    return _screen2 (contexte);
                                },
                            ),
                        );
                    },
                ),
            ],
        );
    );
}

```

Ici, vous pouvez vérifier que le widget Navigator est accessible en utilisant son Navigator.
méthode statique. Vous serez familier avec cela maintenant et, comme vous le devinez peut-être, c'est la façon dont nous accédons à l'ancêtre Navigator correspondant à partir d'un contexte spécifique, et oui, nous pouvons avoir de nombreux widgets Navigator dans une arborescence. C'est génial, car nous pouvons avoir différents éléments de navigation indépendante dans les sous-sections d'une application.

[224]

Épisode 243

Routage: navigation entre les écrans

Chapitre 7

De retour à l'exemple, jetons un coup d'œil au rappel onPressed du widget RaisedButton, où nous poussons une nouvelle Route dans la navigation. De là, la valeur que nous passons à la poussée est similaire à celle renvoyée par le rappel onGenerateRoute dans le Navigateur précédent ajouté.

Pour résumer, notre widget Navigator supérieur utilise le rappel onGenerateRoute uniquement pour initialisez la navigation en fournissant la Route initiale. Plus tard, les boutons d'écran ont été ajouté pour pousser une nouvelle Route vers la navigation, en utilisant la méthode push () du

Widget Navigateur:

```

Bouton // sur l'écran 2 pour revenir en arrière
onPressed: () {
    Navigator.of (contexte) .pop ();
},
// _NavigatorDirectlyAppState

```

Le widget _screen2 est presque égal à _screen1; la seule différence est qu'il apparaît tout seul depuis la navigation et retourne au widget _screen1.

Il y a cependant un problème avec l'exemple précédent. Si nous appuyons sur le bouton retour Android, par exemple, sur l' **écran 2**, nous devrions revenir à l' **écran 1** en conséquence, mais que n'est pas le cas. Comme nous avons ajouté le widget Navigator par nous-mêmes, le système n'est pas conscients de cela: nous devons également le gérer nous-mêmes.

Pour gérer le bouton de retour, nous devons utiliser WidgetsBindingObserver, qui peut être utilisé pour réagir aux messages de cycle de vie liés à une application Flutter. Comme vous pouvez le voir dans la source codes sur GitHub (dans le répertoire de navigation), nous avons d'abord converti notre application en StatefulWidget et ajouté WidgetsBindingObserver en tant que mixin à notre classe State. Nous avons également commencé le observer dans initState () avec WidgetsBinding.instance.addObserver (this); et arrêté l'observateur avec WidgetsBinding.instance.removeObserver (this); sur disposer(). Avec cette configuration, nous pouvons remplacer la méthode didPopRoute ()

à partir de WidgetsBindingObserver et gérez ce qui se passe lorsque le système indique à l'application pour afficher un itinéraire. La méthode didPopRoute () est décrite comme suit dans la documentation:

"[Il est] appelé lorsque le système demande à l'application d'afficher l'itinéraire actuel. Par exemple, sur Android, cela est appelé lorsque l'utilisateur appuie sur le bouton de retour."

[225]

Épisode 244

Routage: navigation entre les écrans

Chapitre 7

Dans la méthode didPopRoute (), nous devons ouvrir une Route à partir de notre widget Navigator. Cependant, nous ne pouvons pas accéder à Navigator via sa méthode statique de la méthode, car nous n'avons pas le contexte ci-dessous ici. Nous pouvons également ajouter une clé au Navigateur et accéder à son état ici:

```
// navigation_directly.dart
la classe _NavigatorDirectlyAppState étend l'état <NavigatorDirectlyApp> {
  _navigatorKey final = GlobalKey <NavigatorState> ();
  // ... autres champs et méthodes

  // fait partie de la méthode de construction
  Navigator(
    clé: _navigatorKey,
    ...
  )
}
```

Et nous pouvons également ajouter la méthode didPopRoute ():

```
@passer outre
Futur <bool> didPopRoute () {
  return Future.value (_navigatorKey.currentState.pop ());
}
```

Ici, nous avons utilisé la méthode pop () de l'état Navigator pour afficher l'itinéraire le plus haut depuis la navigation. Comme cette méthode attend un vrai retour si l'observateur était géré à partir de la notification d'itinéraire pop, nous le renvoyons également à partir des valeurs pop Navigator, donc que lorsqu'il n'y a plus de routes à partir de celui-ci vers pop, le comportement par défaut se produit toujours (il quitte l'application).

La manière WidgetsApp

Comme nous l'avons vu précédemment, ce n'est pas la manière la plus pratique d'utiliser Navigator dans notre applications: nous avons beaucoup de choses à gérer qui pourraient être évitées.

La façon typique de l'utiliser consiste à utiliser les widgets d'application. Ils offrent des propriétés et méthodes pour inclure la navigation dans l'application:

builder: La propriété builder nous permet d'ajouter un chemin alternatif au Navigator, qui est ajouté par WidgetsApp.

home: nous permet de spécifier le widget équivalent à la première route dans l'application (normalement '/').

[226]

Épisode 245

*Routage: navigation entre les écrans**Chapitre 7*

`initialRoute`: nous permet de changer l'itinéraire initial de l'application (par défaut '/').

`navigatorKey` et `navigatorObserver`: nous permet de spécifier les valeurs au widget `Navigator` intégré.

`onGenerateRoute`: crée des widgets basés sur le nom des paramètres de l'itinéraire, comme celui utilisé dans l'exemple précédent. C'est le rappel pour créer

Routes à partir d'un argument `RouteSettings`.

`onUnknownRoute`: spécifie un rappel pour générer une route pour quand il y a un échec dans un processus de création de Route (par exemple, un chemin introuvable).

`pageRouteBuilder`: similaire à `onGenerateRoute`, mais spécialisé sur le type `PageRoute`.

`routes`: Accepte une Map <String, WidgetBuilder>, où nous pouvons ajouter une liste de itinéraires de notre application avec ses blocs de construction correspondants.

L'écriture de l'exemple précédent est plus facile car nous pouvons ignorer tous les implémentations, comme l'observateur du bouton retour ou la touche de navigation:

```
La classe NavigatorWidgetsApp étend StatelessWidget {
  @passer autre
  _NavigatorWidgetsAppState createState () => _NavigatorWidgetsAppState ();
}

la classe _NavigatorWidgetsAppState étend l'état <NavigatorWidgetsApp> {
  @passer autre
  Construction du widget (contexte BuildContext) {
    retourne WidgetsApp (
      couleur: Colors.blue,
      maçon(
        constructeur: (contexte) => _screen1 (contexte),
      ),
      pageRouteBuilder: <Void> (paramètres RouteSettings, WidgetBuilder
      constructeur) {
        return MaterialPageRoute (générateur: générateur, paramètres: paramètres);
      },
    );
  }
  _screen1 (BuildContext context) {...} // masqué par souci de concision
  _screen2 (BuildContext context) {...} // masqué par souci de concision
}
```

[227]

Épisode 246

*Routage: navigation entre les écrans**Chapitre 7*

Comme vous pouvez le voir, l'implémentation précédente est beaucoup plus simple que la première; nous venons spécifiez la propriété `home` et `pageRouteBuilder` à partir de l'application, et le reste fonctionne automatiquement:

Chez nous, nous définissons l'itinéraire initial de la navigation. Nous l'ajoutons dans un constructeur pour déléguer sa création pour un niveau bas dans l'arborescence, donc quand il recherche pour trouver un

Navigateur, cela fonctionnera.

Dans `pageRouteBuilder`, nous définissons quel type d'objet `PageRoute` doit être construit lors de la navigation entre les itinéraires.

Nous pouvons le rendre encore meilleur en utilisant des routes nommées. Voir la section suivante.

Consultez également la documentation de `WidgetsApp` pour plus de détails sur l'utilisation ces propriétés combinées. à: <https://flutter.io/docs/development/ui/navigation/named-routes>

Le code source complet de ces exemples se trouve dans la navigation project dans le répertoire des exemples de chapitre.

Itinéraires nommés

Le nom de l'itinéraire est un élément important de la navigation. C'est l'identification de l'itinéraire avec son gestionnaire, le widget Navigateur.

Nous pouvons définir une série d'itinéraires avec des noms associés à chacun d'eux. Il fournit un niveau d'abstraction à la signification d'un itinéraire et d'un écran. En passant, ils peuvent être utilisés dans une structure de chemin; en d'autres termes, ils peuvent être considérés comme des sous-programmes.

Jetez un œil à la propriété `home` de `WidgetsApp`. Il définit implicitement le widget d'itinéraire de départ pour le widget Navigateur. Il est appelé le «/» chemin.

[228]

Épisode 247

Routage: navigation entre les écrans

Chapitre 7

Déplacement vers des itinéraires nommés

Notre exemple précédent utilisant le widget `WidgetsApp` est très simple, mais nous pouvons le transformer en une manière plus organisée de faire les choses. En utilisant des routes nommées, nous pouvons effectuer les opérations suivantes:

- Organisez les écrans de manière claire
- Centralisez la création d'écrans
- Passer les paramètres aux écrans

Regardons ça:

```
// navigation_widgetsapp_named_routes.dart
la classe _NavigatorNamedRoutesWidgetAppState étend
État <NavigatorNamedRoutesWidgetApp> {
    @passer autre
    Construction du widget (contexte BuildContext) {
        retourne WidgetsApp (
            couleur: Colors.blue,
            routes: {
                '/': (contexte) => _screen1 (contexte),
                '/ 2': (contexte) => _screen2 (contexte),
            },
            pageRouteBuilder: <Void> (paramètres RouteSettings, WidgetBuilder
constructeur) {
                return MaterialPageRoute (générateur: générateur, paramètres: paramètres);
            },
        );
    }
}
```

Dans l'exemple précédent, vous pouvez voir que nous avons utilisé la propriété `routes` pour définir un routage table pour que le navigateur sache ce qu'il faut créer pour chaque chemin.

Nous pouvons toujours utiliser la propriété `home` si nous le souhaitons, comme illustré dans l'exemple suivant:

```
WidgetsApp (
    maçon(
        constructeur: (contexte) => _screen1 (contexte),
    ),
```

```
routes: {
  '/2': (contexte) => _screen2(contexte),
},
...
)
```

[229]

Épisode 248

*Routage: navigation entre les écrans**Chapitre 7*

Notez qu'en faisant cela, nous ne devons pas ajouter la route «/» à la carte des routes.

Un autre avantage de l'utilisation des routes nommées est de pousser de nouvelles routes. Nous pouvons utiliser le `pushNamed` lorsque nous voulons naviguer vers l' **écran 2** à partir de l' **écran 1** :

```
Navigator.of(context).pushNamed('/2');
```

De cette façon, nous n'avons pas besoin de créer l'objet `Route` à chaque appel; il utilisera notre ancien constructeur défini dans la carte des routes de `routesWidgetsApp`.

Arguments

La méthode `pushNamed` accepte également les arguments, à passer à la nouvelle `Route`:

```
Navigator.of(context).pushNamed('/2', arguments: "Bonjour de l'écran 1");
```

Dans ce cas, nous devons utiliser `onGenerateRoute` de `WidgetsApp` afin que nous ayons accès à ces arguments via l'objet `RouteSettings`:

```
// navigation_widgetsapp_named_routes_arguments.dart
classe _NavigatorNamedRoutesArgumentsAppState
  étend l'état <NavigatorNamedRoutesArgumentsApp> {
  @passer autre
  Construction du widget (contexte BuildContext) {
    retourne WidgetsApp (
      couleur: Colors.blue,
      onGenerateRoute: (paramètres) {
        if (settings.name == '/') {
          retourner MaterialPageRoute (
            constructeur: (contexte) => _screen1(contexte)
          );
        } else if (settings.name == '/2') {
          retourner MaterialPageRoute (
            constructeur: (contexte) => _screen2(contexte, paramètres.arguments)
          );
        },
      );
    }
  ...
}
```

Après cela, nous utilisons l'argument normalement trouvé dans le générateur `_screen2`, pour afficher un message supplémentaire.

[230]

Épisode 249

*Routage: navigation entre les écrans**Chapitre 7*

Lors de l'utilisation de la création d'itinéraires à la demande, il semble plus facile de passer

arguments, car vous construirez le widget au moment dont vous avez besoin et pouvez personnaliser la création en passant des arguments selon vos besoins.

Récupération des résultats de Route

Lorsqu'un itinéraire est poussé vers la navigation, nous pouvons vouloir nous attendre à quelque chose en retour. par exemple, lorsque nous demandons quelque chose à l'utilisateur dans une nouvelle route, nous pouvons prendre la valeur renvoyée via le paramètre result de la méthode pop () .

La méthode push et ses variantes renvoient un Future. L'avenir se résout lorsque l'itinéraire est popped et la valeur de Future est le paramètre de résultat de la méthode pop () .

Nous avons vu que nous pouvons passer des arguments à une nouvelle Route. Comme le chemin inverse est également possible, au lieu d'envoyer un message sur le deuxième écran, nous pouvons prendre un message lorsqu'il revient.

Dans l'écran 2 , nous assurons simplement de renvoyer quelque chose lorsque vous faites le pop depuis Navigator:

```
// fait partie de navigation_widgetsapp_navigation_result.dart
classe _NavigatorResultAppState
étend l'état <NavigatorResultApp> {

Widget _screen2 (contexte BuildContext) {
    // ... masqué par souci de brièveté
    RaisedButton (
        enfant: Texte ("Retour à l'écran 1"),
        onPressed: () {
            Navigator.of (context) .pop ("Au revoir depuis l'écran 2");
        },
    ),
    ...
}
}
```

Le deuxième argument de la méthode pop est le résultat de la route.

Dans l'écran de l'appelant, nous devons reprendre le résultat:

```
// fait partie de navigation_widgetsapp_navigation_result.dart
classe _NavigatorResultAppState
étend l'état <NavigatorResultApp> {

Widget _screen1 (contexte BuildContext) {
    // ... masqué par souci de brièveté
}
```

[231]

Épisode 250

Routage: navigation entre les écrans

Chapitre 7

```
RaisedButton (
    enfant: Texte ("Aller à l'écran 2"),
    onPressed: () async {
        message final = attendre Navigator.of (context) .pushNamed ('/ 2') ???
        "Entré du bouton de retour";
        setState (()
            _message = message;
        );
    },
),
...
}
}
```

Veuillez consulter le code source de ce chapitre sur GitHub pour le exemple.

Le résultat de push est un futur que nous devons prendre en utilisant le mot-clé await. Ici, nous venons de définir à une nouvelle variable _message qui est affichée dans un texte.

Si vous ne vous souvenez pas comment travailler avec Future, jetez un œil en arrière

Transitions d'écran

Le changement d'écran doit être fluide du point de vue de l'expérience utilisateur. Nous avons vu que les widgets Navigator fonctionnent sur une superposition pour gérer les itinéraires. La transition entre les routes sont également gérées à ce niveau.

Comme nous l'avons vu, MaterialPageRoute et CupertinoPageRoute sont des classes qui ajoutent un route modale vers la superposition avec une transition adaptative à la plate-forme entre l'ancien et le nouveau Route.

Sur Android, par exemple, la transition d'entrée de la page fait glisser la page vers le haut et la fait fondre. La transition de sortie fait de même en sens inverse. Sur iOS, la page glisse depuis le droit et sort en sens inverse. Flutter nous permet également de personnaliser ce comportement en ajoutant le notre transitions entre les écrans.

[232]

Épisode 251

Routage: navigation entre les écrans

Chapitre 7

PageRouteBuilder

PageRouteBuilder est la définition d'une création de Route. La documentation fournit la définition suivante:

"Une classe utilitaire pour définir des itinéraires de page uniques en termes de rappels."

Si vous vous en souvenez, WidgetsApp contient une propriété pageRouteBuilder où nous définissons quel PageRoute doit être utilisé par notre application, et où les transitions sont normalement défini.

PageRouteBuilder contient plusieurs rappels et propriétés pour aider dans la PageRoute définition. Voici quelques exemples:

transitionsBuilder: le rappel du constructeur pour la transition, où nous construisons le passage de l'itinéraire précédent au nouvel itinéraire
transitionDuration: La durée de la transition
barrièreCouleur et barrièreDismissible: Ceci définit les itinéraires partiellement couverts du modèle et non en plein écran

Consultez la documentation complète pour plus de détails sur PageRouteBuilder
classe: <https://docs.flutter.dev/flutter/widgets/PageRouteBuilder-class.html>

Transitions personnalisées en pratique

Nous pouvons créer une transition personnalisée et l'appliquer globalement dans notre application en utilisant pageRouteBuilder:

```
// fait partie de navigation_transition.dart
la classe _NavigatorTransitionAppState étend l'état <NavigatorTransitionApp> {
  @passer autre
  Construction du widget (contexte BuildContext) {
    retourne WidgetsApp (
      couleur: Colors.blue,
      routes: {
        '/': (contexte)=> _screen1 (contexte),
        '/2': (contexte)=> _screen2 (contexte),
      },
      pageRouteBuilder: <Void> (paramètres RouteSettings, WidgetBuilder
      constructeur) {
```

```
retourne PageRouteBuilder (
```

[233]

Épisode 252

Routage: navigation entre les écrans

Chapitre 7

```
transitionsBuilder :  
    (Contexte BuildContext, animation, SecondaryAnimation, widget) {  
        retourne nouvelle SlideTransition (  
            position: nouveau Tween <Offset> (  
                begin: const Offset (-1.0, 0.0),  
                fin: Offset.zero,  
            ) .animé (animation),  
            enfant: widget,  
        );  
    },  
    pageBuilder : (contexte BuildContext, _, __) => constructeur (contexte),  
);  
},  
);  
...  
}
```

En faisant cela, nous changeons la transition par défaut d'une classe MaterialPageRoute à notre transition de diapositive personnalisée. Nous procédons comme suit:

Notre pageRouteBuilder renvoie maintenant une instance de PageRouteBuilder.
Nous implémentons son rappel pageBuilder pour renvoyer nos widgets normalement, en appeler le rappel du constructeur.
Nous implémentons son callback transitionBuilder pour renvoyer un nouveau widget, généralement une instance AnimatedWidget ou similaire. Ici, nous retournons un Widget SlideTransition qui encapsule la logique d'animation pour nous: a transition de gauche à droite, jusqu'à ce qu'il devienne entièrement visible.

Nous n'avons pas encore vérifié les animations dans les détails. Aller au [chapitre 15, Animations](#), pour en savoir plus.

Une autre façon d'implémenter des transitions personnalisées consiste à créer à la demande la Route objet. Dans ce cas, une bonne approche serait d'étendre la classe PageRouteBuilder et créer une transition réutilisable.

[234]

Épisode 253

Routage: navigation entre les écrans

Chapitre 7

Animations de héros

Le nom *Hero* peut sembler étrange au début, mais tous ceux qui ont utilisé une application mobile a déjà vu ce genre d'animation. Si vous développez pour des plates-formes mobiles, vous ont déjà entendu parler ou travaillé avec *des éléments partagés*, c'est-à-dire des éléments qui persistent entre les écrans. C'est la définition d'un héros.

Flutter contient des moyens pour faciliter la création de ce type de mouvement. C'est pourquoi nous pouvons voir comment les animations Hero fonctionnent avant même que nous approfondissions le sujet des animations lui-même.

Le joueur le plus important cette fois est le widget Hero. En règle générale, il ne s'agit que d'un seul morceau de l'interface utilisateur pour laquelle il est logique de voler d'une Route à une autre.

Le widget Hero

Dans Flutter, un héros est un widget qui vole entre les écrans. Voici un exemple:

[235]

Épisode 254

Routage: navigation entre les écrans

Chapitre 7

Le héros, en réalité, n'est pas le même objet d'un écran à l'autre. Cependant, de l'utilisateur perspective, c'est. L'idée est de créer un widget qui vit entre les écrans et qui change juste son apparence d'une certaine manière. Comme dans la capture d'écran précédente, l'élément augmente et se déplace en même temps que le nouvel écran apparaît. C'est ce que nous apprenons des trois images dans la capture d'écran précédente:

1. C'est à ce moment que nous tapons sur un élément de la liste. Par exemple, la transition commence alors que le l'écran détaillé s'affiche.
2. Une cinématique du processus de transition. Ici, le widget Hero changera sa position et taille jusqu'à ce qu'il corresponde au résultat final (3).
3. L'écran final, avec le héros de l' étape 1 , avec une nouvelle taille.

La documentation Flutter contient d'excellentes explications et exemples à propos de l'animation Hero. N'hésitez pas à le vérifier. à:<https://flutter.dev/docs/development/ui/animations/heros>

Implémentation des transitions Hero

Nous allons changer notre application Favors pour avoir une animation Hero entre les **Your favors** l'écran de liste et l'écran **Demande de faveur** , de sorte que lorsque nous tapons sur le Demander une faveur bouton flottant pour demander une faveur, il y aura une transition en douceur entre celui-ci et le suivant page. Le même effet fonctionne lorsque vous revenez de **Demande de faveur à Votre** écran **favorise** :

[236]

Épisode 255

Routage: navigation entre les écrans

Chapitre 7

Vos faveurs (Ceci est une image de vos faveurs. Les autres informations (qui se chevauchent) ne sont pas importantes ici)

Nous commençons le changement en ajoutant un widget Hero à notre arbre. Il devrait envelopper les widgets impliqués dans l'animation:

```
La classe FavorsPageState étend l'état <FavorsPage> {  
    // ...  
    @passer autre  
    Construction du widget (contexte BuildContext) {  
        // ...  
        FloatingActionButton: FloatingActionButton (  
            heroTag: "request_favor",  
            enfant: FloatingActionButton (  
                onPressed: () {  
                    Navigator.of(contexte).push (  
                        MaterialPageRoute (  
                            constructeur: (contexte) => RequestFavorPage (  
                                amis: mockFriends,  
                                ),  
                                ),  
                            );  
                },  
            ),  
        },  
    },  
},
```

[237]

Épisode 256

Routage: navigation entre les écrans

Chapitre 7

```
        info-bulle: "Demander une faveur",
        enfant: Icône (Icons.add),
    ),
),
...
}
```

La chose la plus importante à remarquer ici est la simplicité. Notre FloatingActionButton contient une propriété de balise heroTag qui le fait se comporter comme un Widget Hero, ce qui signifie qu'il peut animer une transition vers un autre écran. Pour la seconde écran, il suffit de répéter le processus:

```
// fait partie de la méthode de génération RequestFavorPageState
@passer autre
Construction du widget (contexte BuildContext) {
    retourner le héros (
        tag: "request_favor",
        enfant: Scaffold (
            // reste de l'échafaudage
        ),
    );
}
...

```

Consultez le fichier `hands_on_hero` sur GitHub.

Faites attention à la propriété de la balise: c'est là que la magie opère. Ce qui suit est de la Site Web Flutter:

"Il est essentiel que les deux widgets héros soient créés avec la même balise, généralement un objet qui représente les données sous-jacentes."

En outre, il est recommandé que les widgets Hero aient des arbres de widgets pratiquement identiques, voire mieux, soyez le même widget, pour les meilleurs résultats d'animation.

Dans notre exemple précédent, nous animions notre FloatingActionButton à l'ensemble **Demande de widget d'écran de faveur**. Cela fait un effet cool du bouton au nouveau écran. Cependant, il ne montre pas la meilleure capacité de l'animation Hero - partage éléments entre les écrans. En outre, le widget FloatingActionButton et la cible

Le widget Scaffold n'a rien de commun dans sa sous-arborescence de widgets, ce qui provoque notre effet n'est pas le meilleur possible, selon la documentation.

[238]

Épisode 257

Routage: navigation entre les écrans

Chapitre 7

Tenons-nous en à un autre exemple. Supposons que nous ayons un écran de détails pour nos faveurs, et quand l'utilisateur tape sur un FavorCardItem, il montre la faveur correspondante en plein écran, animer cette transition avec un widget Hero. Voici à quoi ressemblera l'effet:

Je sais que cela n'a peut-être pas l'air cool sur les captures d'écran, mais jetez un œil au code joint pour voir le potentiel du widget Hero.

Pour que l'avatar et le texte s'animent sur le nouvel écran pendant la transition, nous avons besoin pour créer deux héros, un pour l'image et un pour la description. C'est ce que nous avons modifié dans le widget FavorCardItem:

```
La classe FavorCardItem étend StatelessWidget {
...
@passer autre
Construction du widget (contexte BuildContext) {
...
    _itemHeader (contexte, faveur),
    Héros(

```

[239]

Épisode 258

Routage: navigation entre les écrans

Chapitre 7

```
        tag: "description _ \$ {favor.uuid}",
        enfant: Texte (
            favor.description,
            style: bodyStyle,
        ),
    ),
    _itemFooter (contexte, faveur)
...
}
...
}
```

De la même manière, nous avons modifié la méthode `_itemHeader` pour avoir un widget Hero emballer notre avatar:

```
Widget _itemHeader (contexte BuildContext, faveur faveur) {
...
Héros(
    tag: "avatar _ \$ {favor.uuid}",
    enfant: CircleAvatar (
        backgroundImage: NetworkImage (
            favor.friend.photoURL,
        ),
    ),
),
...
}
```

Faites attention à la propriété tag de Hero. Nous l'avons spécifié en utilisant l'uuid de la faveur valeur pour rendre le héros identifiable de manière unique dans le contexte.

Pour lancer l'écran des **détails des favoris**, nous avons besoin d'un petit changement dans notre widget FavorsList:

```
La classe FavorsList étend StatelessWidget {
...
@passer autre
```

```

Construction du widget (contexte BuildContext) {
...
    Étendu(
        enfant: ListView.builder (
            physique: BouncingScrollPhysics (),
            itemCount: favors.length,
            itemBuilder: (contexte BuildContext, index int) {
                faveur finale = faveurs [index];
            retourner InkWell (
                en fût: () {
                    Navigator.push (
                        le contexte,

```

[240]

Épisode 259

Routage: navigation entre les écrans

Chapitre 7

```

PageRouteBuilder (
    // transitionDuration: Durée (secondes: 3),
    // décommenter pour voir sa transition plus lente
    pageBuilder: (_, __, ___)=>
        FavorDetailsPage (favor: favor),
    ),
),
},
enfant: FavorCardItem (faveur: faveur),
);
),
),
),
),
...
}
...
}
}
```

Nous avons enveloppé notre FavorCardItem dans un widget InkWell pour gérer les tapotements dessus. Quand l'utilisateur appuie dessus, un nouvel itinéraire sera poussé vers le navigateur pour afficher le widget FavorDetailsPage.

Nous avons utilisé PageRouteBuilder cette fois, au lieu de

MaterialPageRoute, car nous ne voulons pas d'effets Material dans ce transition. Consultez la documentation de PageRouteBuilder pour plus de détails, à: <https://.../api/> classe.htm

La dernière partie à examiner est le widget FavorDetailsPage. Ici, nous créons la finale l'apparence de l'écran des détails de la faveur, et en enveloppant l'avatar et la description de la faveur dans Widgets Hero, nous avons une transition impressionnante. Voici à quoi ressemble sa méthode build ():

```

// fait partie de hands_on_hero / lib / main.dart
la classe _FavorDetailsPageState étend l'état <FavorDetailsPage> {
@passer autre
    Construction du widget (contexte BuildContext) {
        final bodyStyle = Thème.of (contexte) .textTheme.display1;
        retour échafaudage (
            corps: Carte (
                enfant: Rembourrage (
                    rembourrage: EdgeInsets.symmetric (vertical: 10,0, horizontal: 25,0),
                    enfant: Colonne (
                        mainAxisSize: MainAxisSize.min,
                        crossAxisAlignment: CrossAxisAlignment.stretch,
                        enfants: <Widget> [
                            _itemHeader (contexte, widget.favor),

```

[241]

Épisode 260

```
Conteneur (hauteur: 16,0),  
Étendu(  
    enfant: Centre (  
        enfant: Hero (  
            tag: "description _ $ {widget.favor.uuid}",  
            enfant: Texte (  
                widget.favor.description,  
                style: bodyStyle,  
            ),  
            ),  
            ),  
            ),  
        ],  
        ),  
        ),  
    );  
);  
};
```

Et, de la même manière, le `_itemHeader()` est défini comme suit:

```
Widget _itemHeader (contexte BuildContext, faveur faveur) {
    final headerStyle = Thème.of(context).textTheme.display2;

    colonne de retour (
        mainAxisAlignment: MainAxisAlignment.min,
        crossAxisAlignment: CrossAxisAlignment.center,
        enfants: <Widget> [
            Héros(
                tag: "avatar_ ${favor.uuid}",
                enfant: CircleAvatar (
                    rayon: 60,
                    backgroundImage: NetworkImage (
                        favor.friend.photoURL,
                    ),
                ),
            ),
            Conteneur (hauteur: 16,0),
            Texte(
                "$ {favor.friend.name} vous a demandé de ...",
                style: headerStyle,
            ),
        ],
    );
}
```

[242]

Épisode 261

Comme vous pouvez le voir, il ressemble au widget `FavorCardItem`, visant à avoir un minimum de différences dans l'arbre pour obtenir un meilleur résultat de transition. Notez également que la chose principale à être concerné est la propriété `tag` de `Hero`, qui doit correspondre à la balise d'origine de l'effet travailler.

Veuillez consulter le code source ci-joint de ce chapitre pour le exemple.

Le navigateur a toujours son importance ici, tout comme les actions push ou pop qui déclenchent le héros animation (en signalant que l'itinéraire change).

Outre la propriété tag, Hero contient d'autres propriétés pour permettre la personnalisation du vol:

`transitionOnUserGestures`: pour activer / désactiver l'animation Hero sur l'utilisateur gestes tels que le retour sur Android

createRectTween et flightShuttleBuilder: rappels pour modifier le apparence de transition

placeholderBuilder: Un rappel pour renvoyer un widget qui peut être affiché dans le la place du héros source pendant la transition

Au [chapitre 15, Animations](#), au fur et à mesure que nous développons notre compréhension animations, vous pourrez travailler avec ces propriétés comme un naturel.

Les animations de héros sont faciles à implémenter dans Flutter, comme vous pouvez le voir, et même par défaut l'animation fournie par le framework peut suffire à créer un bon effet sur certaines pièces de mise en page.

Consultez la documentation sur le widget Hero: <https://docs.flutter.io/>

[io.flutter](#)

[243]

Épisode 262

Routage: navigation entre les écrans

Chapitre 7

Sommaire

Dans ce chapitre, nous avons vu comment ajouter une navigation entre nos écrans. Tout d'abord, nous devons Connaissez le widget Navigator, l'acteur principal de la navigation dans Flutter. nous ont vu comment il compose la pile de navigation ou l'historique en utilisant la classe Overlay.

Nous avons également vu un autre élément important de la navigation, l'itinéraire, et comment le définir pour utiliser dans nos applications. Nous avons vérifié différentes approches pour implémenter la navigation, la manière la plus typique étant d'utiliser le widget WidgetsApp.

Enfin, nous avons vu comment personnaliser les transitions entre les écrans pour changer la valeur par défaut les mouvements spécifiques à la plate-forme à partir des applications Material et iOS Cupertino, et aussi, comment utiliser Hero animations pour partager des éléments entre les transitions pour créer des effets sympas.

Dans le prochain chapitre, nous porterons notre application favors à un niveau supérieur en intégrant avec les services Firebase.

[244]

Épisode 263

3

Section 3: Développer pleinement Applications en vedette

Pour développer une application professionnelle, le développeur doit ajouter des fonctionnalités qui englobent un nombre de mécanismes avancés et personnalisés, utilisant des plugins pour étendre le framework à nécessaire.

Les chapitres suivants sont inclus dans cette section:

[Chapitre 8](#), Plugins Firebase

[Chapitre 9](#), Développer votre propre plugin Flutter

[Chapitre 10](#), Accès aux fonctionnalités de l'appareil depuis l'application Flutter

[Chapitre 11](#), Vues de la plate-forme et intégration de la carte

Épisode 264

Plugins Firebase 8

Les développeurs créent généralement des codes modulaires qui peuvent être utilisés dans plusieurs applications. Ce n'est pas différent dans le monde Flutter; la communauté est très impliquée dans le succès du framework et de nombreux plugins sont disponibles pour les développeurs. Dans ce chapitre, vous allez apprenez à connaître et à utiliser les plugins Firebase intéressants, tels que Auth, Cloud Firestore et ML Kit, pour créer une application complète sans backend complexe.

Les sujets suivants seront traités dans ce chapitre:

- Configuration du projet Firebase
- Authentification Firebase
- Cloud Firestore
- Stockage Firebase
- Firebase AdMob
- Kit Firebase ML

Présentation de Firebase

Firebase est un produit Google qui fournit plusieurs technologies pour plusieurs plates-formes. Si vous êtes un développeur mobile ou web, vous serez familiarisé avec cette plateforme incroyable.

Épisode 265

Plugins Firebase

Chapitre 8

Parmi ses technologies proposées, les plus importantes sont les suivantes:

- Hébergement** : permet le déploiement d'applications d'une seule page, Web progressif applications ou sites statiques.
- Base de données en temps réel** : une base de **données** NoSQL (base de données non relationnelle) sur le cloud. Avec cela, nous pouvons stocker et synchroniser les données en temps réel.
- Cloud Firestore** : une base de données NoSQL optimisée, axée sur les grands et évolutifs applications qui fournissent une prise en charge avancée des requêtes par rapport au temps réel base de données.
- Fonctions cloud** : fonctions déclenchées par de nombreux produits Firebase, tels que les précédents, ainsi que par l'utilisateur (en utilisant le SDK). Nous pouvons développer des scripts pour réagir aux changements de base de données, d'authentification des utilisateurs, etc.
- Surveillance des performances** : collectez et analysez les informations sur les applications du point de vue de l'utilisateur.
- Authentification** : facilite le développement de la couche d'authentification d'un application, améliorant l'expérience utilisateur et la sécurité. Il permet l'utilisation de plusieurs fournisseurs d'authentification, tels que e-mail / mot de passe, téléphone l'authentification, ainsi que Google, Facebook et d'autres systèmes de connexion.
- Firebase Cloud Messaging** : messagerie cloud pour échanger des messages entre applications et serveur, disponibles sur Android, iOS et Web.
- AdMob**: affiche des annonces pour monétiser les applications.

Kit d'apprentissage automatique : outils pour planter l' **apprentissage automatique** avancé (**ML**) ressourcés dans n'importe quelle application.

Flutter contient une variété de plugins pour fonctionner avec Firebase. Nous en utiliserons certains dans les sections suivantes pour intégrer notre application à ces super services.

Configurer Firebase

Nous ajouterons certaines des technologies Firebase à nos faveurs précédemment développées application, comme l'authentification Firebase et Cloud Firestore. Les étapes, cependant, sont toujours la même chose pour toute application Flutter.

La première étape pour connecter une application à Firebase consiste à créer un projet d'application Firebase.

[247]

Épisode 266

Plugins Firebase

Chapitre 8

Nous faisons cela sur l' outil de **console Firebase** ([https://console.firebaseio.com](https://console.firebase.google.com)) outil nous permet de gérer tous nos projets Firebase, d'activer / dé et surveiller l'utilisation:

1. Il s'agit de l'écran initial de la console Firebase où vous pouvez voir les projets et ajoutez également un nouveau projet:

[248]

Épisode 267

*Plugins Firebase**Chapitre 8*

2. Le processus de lancement d'un projet Firebase est simple et facile à suivre, comme indiqué dans la capture d'écran suivante:

3. Le projet va générer en quelques secondes comme ceci:

[249]

Épisode 268

*Plugins Firebase**Chapitre 8*

4. Une fois le projet créé, vous serez redirigé vers l'écran du projet, comme indiqué ici:

[250]

Episode 269

Plugins Firebase

Chapitre 8

5. L'écran suivant montre toutes les options concernant le projet ainsi que le raccourci de configuration vers les paramètres du projet:

Ici, nous configurons nos applications de projet, car nous pouvons avoir plusieurs applications par projet (qui est, un pour chaque plate-forme mobile) et vérifiez également les informations d'identification du projet utilisées pour configurer le SDK sur Flutter.

[251]

Épisode 270

Plugins Firebase

Chapitre 8

Connexion de l'application Flutter à Firebase

Comme nous l'avons vu précédemment, il est possible de configurer plusieurs applications à partir de plusieurs plates-formes pour se connecter à un projet Firebase. Dans la page du projet Firebase, nous avons le option pour ajouter des applications pour iOS, Android et Web.

Nous devons configurer deux applications dans Firebase: une pour iOS et une pour Android, car alors que nous développions des applications mobiles natives. Donc, si vous l'avez déjà fait configuration avant pour toute application, la section suivante peut paraître simple.

Configurer une application Android

Nous pouvons configurer une application Android via le raccourci de l'assistant de configuration Android dans la page générale du projet vue précédemment:

[252]

Épisode 271

Plugins Firebase

Chapitre 8

Cela amène la page de configuration à l'application Android illustrée dans la capture d'écran suivante:



Ici, le paramètre important est le nom du package qui est vérifié dans le SDK Firebase. le certificat de signature est également important pour l'authentification; nous allons couvrir cela sous peu.

Vous pouvez trouver le nom du package de votre application Android dans le fichier android / app / build.gradle, via l'applicationId propriété.

Après l'enregistrement, un fichier google-services.json est généré et doit être ajouté à notre projet d'application. Sous Android, il doit être situé dans le répertoire android / app.

[253]

Épisode 272

Plugins Firebase

Chapitre 8

La dernière étape consiste à ajouter le SDK Firebase aux fichiers Gradle. Sous Android, Gradle peut être vu comme l'équivalent pubspec Flutter. L'une de ses responsabilités est de gérer l'application dépendances:

1. Tout d'abord, nous ajoutons la dépendance google-services à classpath dans le fichier android / build.gradle comme ceci:

```
buildscript {
    repositories {
        google() // ajouter ceci s'il n'est pas présent
        ...
    }
    dependencies {
        ...
    }
}
```

```

classpath 'com.google.gms:google-services: 3.2.1' // ajouter
    // ce
    // ligne
}
}

```

2. Après cela, dans android / app / build.gradle, nous devons activer le plugin et ajouter une dépendance à la lib 'androidx.annotation', comme indiqué ci-dessous code:

```

// fait partie d'android / app / build.gradle
...
dépendances {
    implémentation 'androidx.annotation: annotation: 1.0.2'
    ...
}

// base de feu
// Ajoutez la ligne suivante au bas du fichier:
appliquer le plugin: 'com.google.gms.google-services'

La bibliothèque androidx.annotation n'est pas directement liée à Firebase. nous
devrait l'ajouter, cependant, car certaines bibliothèques en ont besoin en interne, comme le
ceux de Firebase.

```

3. Enfin, en exécutant la commande suivante, nous serons tous configurés dans Android environnement:

les paquets de flutter obtiennent

[254]

Épisode 273

Plugins Firebase

Chapitre 8

Configurer l'application iOS

Pour la version iOS, le processus est très similaire. En commençant par la configuration dans le Console Firebase, où nous définissons le nom du package comme nous l'avons fait pour Android.

Après cela, nous pouvons télécharger le GoogleService-Info.plist généré (équivalent iOS to google-services.json) et ajoutez-le au répertoire iOS ios / Runner du projet. Ses important de le faire dans Xcode en ouvrant le projet iOS dessus et en faisant glisser le fichier dans Xcode afin qu'il soit enregistré pour inclusion lors des builds.

L'étape d'ajout du fichier GoogleService-Info.plist est en train de changer,
selon les versions des plug-ins Flutter. Découvrez le plus approprié
chemin ici: <https://.. / fireb>

Contrairement à Android, il n'est pas nécessaire d'ajouter des dépendances iOS spécifiques pour Firebase. Le suivant l'étape consiste à travailler dans le contexte Flutter.

FlutterFire

Les applications Flutter reposent sur un ensemble de plug-ins Flutter pour accéder aux services Firebase. FlutterFire contient des implémentations spécifiques pour les plates-formes iOS et Android cibles.

Consultez la page des plugins FlutterFire pour plus d'informations sur les versions récentes des plugins Firebase: <https://.. / firebase>
<https://.. / flutter/> / plugins /.

Ajout de la dépendance FlutterFire au projet Flutter

Nous devrions ajouter le plugin principal à notre projet en tant que dépendance fondamentale initiale comme illustré dans le code suivant:

part de pubspec.yaml

dépendances:

```
...
firebase_core: 0.2.5 # Firebase Core
```

[255]

Épisode 274*Plugins Firebase**Chapitre 8*

En plus de cela, nous devrions ajouter toutes les dépendances Firebase si nécessaire. De plus, nous devrions ajoutez `firebase_auth` pour travailler avec l'authentification téléphonique:

```
# part de pubspec.yaml
dépendances:
...
firebase_core: 0.3.4 # Firebase Core
```

Remarque sur Android :

Comme nous utilisons les dernières versions des plugins Firebase basés sur les versions AndroidX des dépendances, notre projet d'application a été migré vers AndroidX. En raison de problèmes de compatibilité avec AndroidX, je vous recommande de Découvrez plus ici: <https://flutter.dev/c>

[packages](#)-et-plt

L'exécution de la commande `get flutter packages` termine le processus d'installation, ce qui signifie et maintenant nous pouvons commencer à travailler avec les plugins.

Si vous trouvez cela plus facile, vous pouvez suivre la documentation officielle de Firebase étapes pour l'initialisation de Firebase dans Flutter: <https://.. / firebase>

[docs / flu](#)

Authentification Firebase

Comme nous l'avons vu précédemment, Firebase contient une collection de technologies utiles et nous avons besoin pour configurer chacun dont nous pourrions avoir besoin pour notre projet. Configurons l'authentification couche de notre application. La couche d'authentification est fondamentale pour notre application; si vous vous en souvenez, le les demandes de faveur des utilisateurs sont adressées à des amis, et pour cela, nous avons besoin que l'utilisateur soit capable d'envoyer la demande à un utilisateur spécifique. Nous faisons cette identification en utilisant le numéro de téléphone de l'utilisateur comme identité. Nous devons le faire dans les étapes suivantes:

1. Ajouter le plug-in d'authentification Firebase au projet
2. Comme indiqué précédemment, nous devons simplement ajouter le plugin `firebase_auth` dépendance à notre `pubspec`, comme indiqué dans le code suivant:

```
# part de pubspec.yaml
dépendances:
...
firebase_core: 0.3.4 # Firebase Core
firebase_auth: 0.8.4 +5 # Firebase Auth // ajouter ceci
```

[256]

Épisode 275*Plugins Firebase**Chapitre 8*

3. Activer l'authentification téléphonique pour notre projet Firebase dans la console Firebase
4. Créer l'écran d'authentification
5. Vérifiez si l'utilisateur est connecté, et si ce n'est pas le cas, redirigez-vous vers la page de connexion

Activation des services d'authentification dans Firebase

Pour activer les services d'**authentification** dans Firebase, nous devons visiter la section **Authentification** dans la console Firebase comme illustré dans la capture d'écran suivante:

Après l'avoir activé, nous pouvons ajouter un numéro de téléphone de test pendant le développement afin de ne pas affecter l'utilisation des ressources pour les autres utilisateurs, comme indiqué ici:

[257]

Épisode 276

Il est important que vous configureriez un numéro de téléphone de test et un code de vérification. Pendant développement, votre application Android est signée avec un certificat de **débogage**. De cette façon, dans la connexion écran, lorsque vous êtes invité à entrer le numéro de téléphone, cela ne fonctionnera qu'avec le numéros de téléphone précédemment répertoriés. De plus, au lieu de recevoir le code de vérification, vous tapez simplement celui qui y est enregistré.

Après cette configuration, nous pouvons commencer à travailler sur le code Flutter.

[258]

Épisode 277

Plugins Firebase

Chapitre 8

Pour l'authentification avec des nombres réels et la réception d'un code de vérification, vous devez signer votre application en mode version. Plus d'informations sur le mode de sortie plus tard au [chapitre 12, Test, débogage et déploiement](#).

Écran d'authentification

Dans cet écran, nous n'allons pas parler des détails de mise en page. Le seul nouveau widget ici est le Widget Stepper, de Material Design. L'idée générale est que l'utilisateur entre son téléphone numéro, reçoit un code de validation et après l'avoir confirmé, se connecte. Nous avons également utilisé notre entrée personnalisée de [Chapitre 5, Gestion des entrées et des gestes de l'utilisateur](#) :

[259]

Épisode 278

*Plugins Firebase**Chapitre 8*

Comme vous pouvez le voir, la mise en page est simple et le widget Stepper aide sur le flux de travail de connexion, en suivant étape par étape les étapes suivantes:

1. L'utilisateur remplit son numéro de téléphone
2. L'utilisateur remplit le code de vérification (reçu par SMS)
3. L'utilisateur remplit le nom d'affichage et l'image de profil

Vous pouvez en savoir plus sur ce widget sur son material.io page: <https://material.io/components/steppers/>.

Connexion avec Firebase

Vous pouvez vérifier le code plein écran dans le projet hands_on_firebase joint. Le principal les fonctions ici sont `_sendVerificationCode()` et `_executeLogin()` de `LoginPageState`.

Si vous vérifiez le code source ci-joint, vous remarquerez que nous avons ajouté ce qui suit deux `<Step>`s à notre widget Stepper:

1. **Envoyer le code de vérification** : dans cette première étape, l'utilisateur remplit son numéro de téléphone à récupérer un code de vérification.
2. **Entrez le code de vérification à 6 chiffres récupéré** : pour confirmer l'identité de l'utilisateur. Après cela, l'utilisateur se connecte.

Outre les propriétés du widget Stepper, concentrons-nous sur son champ `onStepContinue`, qui est comme montré ici:

```
// fait partie de la méthode de génération LoginPageState. Le rappel Stepper:
onStepContinue: () {
    if (_currentStep == 0) {
        _envoyer le code de vérification();
    } else if (_currentStep == 1) {
        _executeLogin();
    } autre {
        _Enregistrer le profil();
    }
},
```

[260]

Épisode 279

*Plugins Firebase**Chapitre 8*

Ce champ attend un rappel qui est appelé lorsque l'utilisateur appuie sur le bouton **Continuer** de chaque étape. Comme nous conservons l'étape actuellement active dans le champ `_currentStep`, nous savons quelle action effectuer. Voyons donc comment chaque action est effectuée.

Nous avons personnalisé l'apparence des actions par étapes; vérifier

la méthode `_stepControlsBuilder` sur la classe `LoginPageState` pour le voir en détail. Consultez également la documentation de ce Stepper propriété: <https://.. / docs controlsBuilder.html>.

Envoyer du code de vérification

La première étape de l'authentification téléphonique est lorsque le serveur (Firebase, dans notre cas) envoie un code de vérification par SMS au numéro de téléphone saisi par l'utilisateur.

Pour ce faire, utilisez la méthode du SDK Firebase appelée `verifyPhoneNumber`, qui demande au serveur de *démarrer* une authentification téléphonique comme indiqué ici:

```
// Méthode _sendVerificationCode (LoginPageState) login_page.dart

void _sendVerificationCode () async {
    Final PhoneCodeSent codeSent = (String verId, [int forceCodeResend]) {
        _verificationId = verId;
        _goToVerificationStep ();
    };

    Final PhoneVerificationCompleted verificationSuccess = (FirebaseUser
utilisateur) {
        _connecté();
    };

    Final PhoneVerificationFailed verificationFail = (AuthException
exception) {
        goBackToFirstStep ();
    };

    Final PhoneCodeAutoRetrievalTimeout autoRetrievalTimeout = (String verId)
{
    this._verificationId = verId;
};

attendez FirebaseAuth.instance.verifyPhoneNumber (
    phoneNumber: _phoneNumber,
    codeSent: codeSent,
    verificationCompleted: verificationSuccess,
```

[261]

Épisode 280

Plugins Firebase

Chapitre 8

```
verificationFailed: verificationFail,
codeAutoRetrievalTimeout: autoRetrievalTimeout,
timeout: Durée (secondes: 0),
);
}
```

La méthode `verifyPhoneNumber` s'exécute de manière asynchrone (une autre avec `async` et renvoie `Future`), le mot-clé `await` est donc nécessaire avant le appel.

Voici quelques éléments importants à noter dans le code précédent:

`FirebaseAuth.instance` reflète l'instance unique du SDK d'authentification Firebase qui fait le pont entre Flutter et les bibliothèques d'authentification Firebase natives

Il existe plusieurs rappels à implémenter et des propriétés à définir sur le appel d'API d'authentification, à savoir ceux-ci:

- `phoneNumber`: numéro de téléphone auquel envoyer le code de vérification
- `codeSent`: appelé lorsque le code est envoyé à `phoneNumber`
- `verificationCompleted`: appelé lorsque le code est récupéré automatiquement par le SDK d'authentification Firebase
- `verificationFailed`: Appelé lorsqu'une erreur se produit pendant la vérification du numéro de téléphone
- `timeout`: temps maximum pendant lequel la bibliothèque attend une réponse

récupération, 0 signifie désactivé
 codeAutoRetrievalTimeout: appelé lorsque le
 le délai spécifié est atteint, ce qui signifie que la récupération automatique n'a pas fonctionné
 correctement (sauf s'il est mis à 0)

Lorsque le rappel codeSent est appelé, le widget Stepper se déplace
 à la deuxième étape, où l'utilisateur doit saisir son code de vérification

Il est fondamental pour vous d'inspecter le site FlutterFire ainsi que le
 documentation du plugin firebase_auth pour une compréhension du
 propriétés précédentes: <https://.. / pub dart auth>

De plus, nous avons désactivé la récupération automatique car elle ne fonctionne pas complètement à ce moment-là
 d'écrire ce livre; vous pouvez modifier les rappels pour tester par vous-même.

[262]

Épisode 281

Plugins Firebase

Chapitre 8

Vérification du code SMS

La deuxième étape consiste à vérifier que l'utilisateur a récupéré le code correct et, ce faisant, doit connectez-vous à l'application. Cela se fait dans la méthode signInWithCredential comme indiqué ici:

```
// Méthode _executeLogin (LoginPageState) login_page.dart

void _executeLogin () async {
  setState (() {
    _showProgress = true;
  });

  attendez FirebaseAuth.instance.signInWithCredential (
    PhoneAuthProvider.getCredential (
      verificationId: _verificationId, smsCode: _smsCode,
    )
  );

  FirebaseAuth.instance.currentUser (). Then ((utilisateur) {
    if (utilisateur! = null) {
      goToProfileStep ();
    }
  });
}
```

Comme vous pouvez le voir, il s'agit d'un simple appel à la méthode signInWithCredential depuis le Plug-in d'authentification Firebase qui attend les deux arguments suivants:

verificationId: Il s'agit de l'identifiant de l'ensemble du processus de connexion. Jeter un coup d'œil à les rappels précédents où nous recevons ceci et le stockons pour une utilisation ultérieure ici. Ce identifie le login afin que nous n'ayons pas besoin d'envoyer toutes les informations (téléphone numéro, dans ce cas) à nouveau.

smsCode: Le code que l'utilisateur a entré pour la validation; si les deux sont valides, la connexion sera réussir.

Si vous effectuez des tests, vous remarquerez que l'application ne s'affiche pas messages à l'utilisateur pour les avertir des erreurs de connexion (par exemple pour un code de vérification). Dans une application du monde réel, ce n'est pas le comportement idéal. Prendre regardez les rappels et essayez d'améliorer le comportement.

[263]

Épisode 282

Plugins Firebase

Chapitre 8

Mise à jour du profil et du statut de connexion

L'objet utilisateur Firebase contient plus que des numéros de téléphone, il contient un ensemble de des informations pour une autre méthode de connexion, comme le courrier électronique, par exemple, et contient également propriétés qui aident à définir le profil de l'utilisateur, comme un nom d'affichage et une photo URL. Ici, dans la dernière étape du processus de connexion, nous pouvons enregistrer le profil utilisateur avec son displayName afin que les autres utilisateurs puissent s'identifier facilement. Ceci est fait dans le _saveProfile () méthode comme indiqué ici:

```
// fait partie de la classe LoginPageState
void _saveProfile () async {
    setState (() {
        _showProgress = true;
    });
}

utilisateur final = attendre FirebaseAuth.instance.currentUser ();

mise à jour finale = UserUpdateInfo ();
updateInfo.displayName = _displayName;

attendre user.updateProfile (updateInfo);

// ... la dernière partie est expliquée ci-dessous
}
```

La méthode currentUser () est utile pour toute action liée à l'utilisateur connecté. Dans ce cas, nous l'obtenons et mettons à jour les informations demandées (le nom d'affichage, pour l'instant). UserUpdateInfo est une classe d'assistance pour stocker les données de mise à jour; dans la section suivante, nous serons en utilisant une autre propriété pour stocker l'URL de l'image du profil utilisateur.

Comme nous savons que l'utilisateur est connecté, nous pouvons rediriger vers la page Favors en utilisant le bien connu Classe de navigateur comme suit:

```
// dernière partie de _saveProfile () LoginPage
Navigator.of (contexte) .pushReplacement (
    MaterialPageRoute (
        constructeur: (contexte) => FavorsPage (),
    ),
);
```

[264]

Épisode 283

Plugins Firebase

Chapitre 8

Cet écran est l'écran initial de notre application. Cependant, nous ne devons pas demander à l'utilisateur de remplir tous l'information à chaque fois. Avant toute chose, il faut vérifier si l'utilisateur est déjà connecté, et s'ils le sont, redirigez simplement comme nous l'avons fait auparavant. Nous pouvons le faire en utilisant la méthode FirebaseAuth.instance.currentUser () à nouveau. Un bon endroit pour vérifier cela est la méthode initState () de la classe LoginPageState:

```
// fait partie de login_page.dart
La classe LoginPageState étend l'état <LoginPage> {
...
    @passer autre
    void initState () {
```

```

super.initState();

FirebaseAuth.instance.currentUser().Then((utilisateur) {
  if (utilisateur != null) {
    Navigator.of(context).pushReplacement(
      MaterialPageRoute(
        constructeur: (contexte) => FavorsPage(),
      ),
    );
  }
});

...
}

```

Comme vous pouvez le voir, si l'utilisateur actuel de Firebase n'est pas nul, nous savons que nous pouvons rediriger le navigation vers l'écran suivant comme avant.

Quels seraient les bons commentaires des utilisateurs si l'utilisateur actuel est nul? Avoir un réfléchir et découvrir.

C'est tout pour l'authentification par téléphone; dans la section suivante, nous allons stocker nos faveurs sur le backend Cloud Firestore.

Base de données NoSQL avec Cloud Firestore

Cloud Firestore de Firebase est une base de données cloud NoSQL flexible et évolutive. Cela nous aide dans le développement d'applications temps réel avec des technologies de synchronisation entre clients qui rendent notre application rapide et fonctionnelle.

[265]

Épisode 284

Plugins Firebase

Chapitre 8

Dans ce chapitre, nous allons apporter quelques modifications à notre application Favors. Nous ferons le Suivant:

Transférer notre liste de faveurs vers Firebase

Découvrez comment ajouter des règles afin qu'un utilisateur ne puisse pas accéder aux faveurs d'un autre utilisateur

Envoyer / stocker une demande de faveur à un autre utilisateur / ami dans Cloud Firestore

Activation de Cloud Firestore sur Firebase

La première étape, si vous vous en souvenez, consiste à activer les services nécessaires sur Firebase. Dans ce cas, nous voulons activer la technologie Cloud Firestore sur Firebase:

[266]

Épisode 285

Plugins Firebase

Chapitre 8

Nous l'activons comme n'importe quel autre service Firebase. Une chose importante concernant les données est de faire avec sécurité. Firebase fournit des mécanismes de règles afin que nous puissions configurer le niveau pour accès à toute information stockée dans notre base de données. Dans l'invite de création, c'est le seul chose que nous configurons:

Dans notre application, nous n'allons définir aucune règle de simplicité; c'est pourquoi nous avons choisi mode d'essai. Je vous recommande fortement d'en savoir plus sur ces règles, car elles sont très important pour les applications réelles: <https://firebase.google.com/docs/firestore/security/README.md>

[la sécurité](#) / la st

[267]

Épisode 286

*Plugins Firebase**Chapitre 8*

Après cela, nous pouvons commencer le développement du stockage et du chargement des faveurs sur le Cloud Base de données Firestore.

Cloud Firestore et Flutter

Comme nous l'avons vu précédemment, FlutterFire fournit un ensemble de plugins pour différentes technologies. Cela est également vrai pour le plugin Cloud Firestore. Donc, la première étape consiste à ajouter leur nécessaire dépendances à notre pubspec.yaml comme indiqué ici:

```
dépendances:  
  cloud_firestore: ^0.9.5 # Cloud Firestore
```

Après avoir obtenu les dépendances nécessaires avec les packages de flutter get, nous sommes prêts à changer notre stockage de faveurs.

[268]

Épisode 287

*Plugins Firebase**Chapitre 8*

Chargement des faveurs de Firestore

Nous utilisons Firestore via la classe Firestore de la bibliothèque cloud_firestore Dart.

Dans la fonction initState () de FavorsPageState, nous ajoutons un appel à watchFavorsCollection ().

Les collections ne sont qu'un groupe de documents. Dans notre application, nous avons un seul collection appelée faveurs qui stocke tous les documents de faveur de l'application.
Un document est un enregistrement dans une collection. Ils sont généralement représentés comme Objets JSON.

Dans watchFavorsCollection (), nous commençons à charger les faveurs de Firebase comme indiqué ici:

```
// fait partie de favors_page.dart watchFavorsCollection  
La classe FavorsPageState étend l'état <FavorsPage> {
```

```

@passer outre
void initState () {
    super.initState ();
    ...
    pendingAnswerFavors = Liste ();
    AcceptedFavors = Liste ();
    completeFavors = Liste ();
    refuséFavors = Liste ();
    amis = Set ();
}

watchFavorsCollection ();
}

...
void watchFavorsCollection () async {
    final currentUser = attendre FirebaseAuth.instance.currentUser ();

    Firestore.instance
        .collection ('faveurs') // 1
        .where ('à', isEqualTo: currentUser.phoneNumber) // 2
        .snapshots () // 3
        .listen ((instantané) {}) // 4
    ...
}
}

```

Une requête Firebase typique peut avoir de nombreux formats; celui-ci fait ce qui suit:

1. Il commence par spécifier la collection ciblée - les faveurs.
2. Il ajoute une condition where pour filtrer les faveurs qui ne sont envoyées qu'à l'utilisateur actuel numéro de téléphone.

[269]

Épisode 288

Plugins Firebase

Chapitre 8

3. snapshots () crée un flux de snapshots.
4. listen ((snapshot) {}) est l'endroit où nous écoutons les changements sur les instantanés; cette est, nous souscrivons aux changements de snapshot. À chaque modification de la base de données affecte la requête, la fonction passée à listen () sera appellée. Le rappel le code de la fonction listen () est le suivant:

```

// fait partie de watchFavorsCollection
void watchFavorsCollection () async {
    final currentUser = attendre FirebaseAuth.instance.currentUser ();

    Firestore.instance
        .collection ('faveurs')
        .where ('à', isEqualTo: currentUser.phoneNumber)
        .snapshots ()
        .listen ((instantané) {
            List <Favor> newCompletedFavors = List ();
            List <Favor> newRefusedFavors = List ();
            List <Favor> newAcceptedFavors = List ();
            List <Favor> newPendingAnswerFavors = List ();
            Set <Friend> newFriends = Set ();

            snapshot.documents.forEach ((document) {
                Favor favor = Favor.fromMap (document.documentID,
                    document.data);
                if (favor.isCompleted) {
                    newCompletedFavors.add (favour);
                } else if (favor.isRefused) {
                    newRefusedFavors.add (favour);
                } else if (favor.isDoing) {
                    newAcceptedFavors.add (favour);
                } autre {
                    newPendingAnswerFavors.add (favour);
                }

                newFriends.add (favor.friend);
            });
        });
}

// mettre à jour nos listes
setState (() {

```

```

        this.completedFavors = newCompletedFavors;
        this.pendingAnswerFavors = newPendingAnswerFavors;
        this.refusedFavors = newRefusedFavors;
        this.acceptedFavors = newAcceptedFavors;
        this.friends = newFriends;
    });
}

```

[270]

Épisode 289

Plugins Firebase

Chapitre 8

Comme vous pouvez le voir, chaque fois que la partie de la collection où notre requête est à la recherche change par l'insertion, l'édition, la suppression d'une faveur, le rappel sera appelé et le ce qui suit se produira:

Une nouvelle liste de chaque type de faveur est créée.

Une faveur est créée via un nouveau constructeur défini par fromMap comme indiqué ici:

```

Favor.fromMap (String uid, Map <String, dynamic> data)
    : ce(
        uid: uid,
        description: données ['description'],
        dueDate: DateTime.fromMillisecondesSinceEpoch
            (données ['dueDate']),
        accepté: données ['acceptées'],
        terminé: data ['completed']! = null
            ? DateHeure. De MillisecondesSinceEpoch
                (données ['complétées'])
            : nul,
        ami: Friend.fromMap (données ['ami']),
        à: données ['à'],
    );

```

Le constructeur fromMap reçoit un ID (l'ID de document) et une instance de Map avec le champs correspondants. Comme vous pouvez le voir, c'est une simple utilisation du constructeur par défaut avec paramètres issus des données provenant de Firebase:

La même chose est faite pour l'objet Friend. Découvrez la classe Favor pour cet exemple.

En fonction du statut de faveur, il est inséré dans la liste correspondante.

En plus de cela, un groupe d'amis est créé, et chaque ami de faveur est ajouté à l'ensemble. Comme les ensembles permettent une seule occurrence de chaque objet, pas de répétition des amis seront présents.

Vérifiez la classe Friend. Pour une utilisation correcte dans la collection Set, le, L'opérateur equals (==) et la méthode hashCode ont été remplacés pour le évaluation correcte.

À la fin, les listes de l'instance State sont mises à jour pour provoquer une reconstruction du disposition.

[271]

Épisode 290

Plugins Firebase

Chapitre 8

Mettre à jour les faveurs sur Firebase

Avant, lors de l'utilisation de données fictives, nous avions besoin que de modifier nos listes en mémoire. Maintenant nous avons besoin de mettre à jour nos documents de faveur correspondants sur Firebase afin que cela déclenche notre rappel précédemment défini, ce qui entraînera une reconstruction et une mise à jour de nos mises en page.

Nous créons une nouvelle méthode qui sera utilisée sur chaque faveur changer, `_updateFavorOnFirebase ()`:

```
void _updateFavorOnFirebase (Favoriser) async {
    attendre Firestore.instance
        .collection ('faveurs') // 1
        .document (favor.uuid) // 2
        .setData (favor.toJson ()) // 3
}
```

Le début de l'appel Firestore est presque toujours le même: nous obtenons l'instance Firestore, puis nous complétons les étapes suivantes:

1. Nous allons à la collection des faveurs.
2. Ensuite, nous obtenons la référence du document de faveur que nous voulons mettre à jour.
3. La dernière étape consiste à envoyer les données au format JSON à mettre à jour dans le document correspondant. La méthode `toJson ()` est un simple convertisseur pour stocker sur Firebase.

Consultez le code source joint `hands_on_firebase` pour le code complet de conversions vers et depuis Firebase.

La méthode `_updateFavorOnFirebase` est utilisée sur les méthodes précédemment définies:

compléter, abandonner, accepter à faire et refuser à faire. C'est tout ce dont nous avons besoin pour mettre à jour Firebase et refléter les modifications apportées à la mise en page de l'application.

Enregistrer une faveur sur Firebase

Dans la classe `RequestFavorPageState`, nous devons ajouter le code pour insérer une nouvelle faveur dans notre collection de faveur à Firestore. Ceci est fait avec la méthode `_save ()` précédente, qui, jusqu'à présent, n'a rien enregistré:

```
// fait partie du fichier request_favors_page.dart
void save (contexte BuildContext) async {
    if (_formKey.currentState.validate ()) {
```

[272]

Épisode 291

Plugins Firebase

Chapitre 8

```
_formKey.currentState.save (); // 1
final currentUser = attendre FirebaseAuth.instance.currentUser;
// 2

attendre _saveFavorOnFirebase (
    Favoriser(
        à: _selectedFriend.number,
        description: _description,
        dueDate: _dueDate,
        ami ami(
            nom: currentUser.displayName,
            numéro: currentUser.phoneNumber,
            photoURL: currentUser.photoUrl,
        ),
        ),
    );
); // 3

Navigator.pop (contexte); // 4
}
```

Le processus de sauvegarde est défini comme suit:

1. Nous validons et sauvegardons les champs du formulaire. Autrement dit, nous stockons la valeur des champs de texte

de description, date d'échéance et ami comme variables à utiliser plus tard. Il existe d'autres moyens d'obtenir les valeurs des champs de formulaire; celui-ci est simple et propre.

2. Nous obtenons l'utilisateur actuellement connecté, car nous avons besoin des informations de l'utilisateur actuel pour remplir la demande de faveur, afin que l'ami sollicité sache qui lui demande un favoriser.
3. Nous appelons une nouvelle méthode utilitaire `_saveFavorOnFirebase ()` qui rend le Appel Firebase, avec une nouvelle instance de faveur créée avec les valeurs à venir à partir du formulaire comme indiqué ici:

```
_saveFavorOnFirebase (Favoriser) async {
    attendre Firestore.instance
        .collection ('faveurs')
        .document () // sans passer aucun identifiant de document
        .setData (favor.toJson ());
}
```

[273]

Épisode 292

Plugins Firebase

Chapitre 8

Comme vous pouvez le voir, l'appel est très similaire au code de mise à jour précédent. Le seul chose différente est que nous n'allons pas vers un document spécifique sur l'appel de la méthode `document ()`. Cela amènera Firestore à générer un nouvel unique ID, puis mappez vers un nouveau document dans lequel nous définirons les données ultérieurement.

4. Après l'enregistrement, nous affichons l'itinéraire pour revenir à l'écran précédent.

Peut-être que nous aurions pu traiter les erreurs survenues lors du processus de sauvegarde afin que l'utilisateur puisse réessayer plus tard, qu'en pensez-vous? C'est un bon il est temps de se salir les mains et d'améliorer le code.

Avec ces modifications, nous stockons et récupérons désormais les faveurs de Cloud Firestore, comme indiqué dans la capture d'écran suivante:

Nous n'avons pas écrit de code backend ici, et en prime, nous avons également des modifications en temps réel reflétée dans notre application, ce qui la rend idéale pour les contextes impliquant plusieurs utilisateurs.

[274]

Épisode 293

*Plugins Firebase**Chapitre 8*

Cloud Storage avec Firebase Storage

Firebase **Storage** est une excellente plateforme pour stocker des fichiers sur le cloud. Les cas d'utilisation les plus courants stockent des photos ou des vidéos d'utilisateurs, mais il n'y a aucune limitation; vous pouvez stocker n'importe quel type de données nécessaires à votre application. Les besoins de l'application sont pris en compte avec ce mécanisme de stockage puissant.

Présentation du stockage Firebase

Comme les services précédents, Firebase **Storage** a une étape d'introduction où il explique le besoin pour sécuriser les données, comme indiqué ici:

[275]

Épisode 294

*Plugins Firebase**Chapitre 8*

Le service de stockage est activé avec une définition de règle par défaut, où uniquement authentifié les requêtes peuvent effectuer des appels en écriture et en lecture. Cela suffit pour notre application.

Encore une fois, pour les applications du monde réel, il est recommandé de créer le meilleur les règles que vous pouvez faire pour aider à protéger les données spécifiques à

[google.com](#)

Après cette étape d'introduction, nous pouvons ajouter des bibliothèques spécifiques à Flutter et démarrer le développement étape.

Ajout de dépendances de stockage Flutter

En plus des plugins précédents, FlutterFire fournit un plugin pour Firebase Storage. nous besoin d'ajouter la dépendance à notre pubspec.yaml, comme indiqué dans le code suivant:

```
dépendances:  
  firebase_storage: ^2.1.0 # Cloud Firestore
```

Après avoir obtenu les dépendances avec les packages de flutter get, nous sommes prêts à utiliser Firebase Stockage dans notre projet.

Téléchargement de fichiers sur Firebase

Nous allons ajouter la fonctionnalité de téléchargement de fichiers sur Firebase Storage à nos faveurs app. Dans la section **Profil** du processus de connexion, une fois que l'utilisateur s'est connecté avec succès, nous pouvons ajouter une fonctionnalité afin que l'utilisateur puisse ajouter une image à son profil.

[276]

Épisode 295

Plugins Firebase

Chapitre 8

Vous pouvez consulter cette section de l'application dans la dernière partie de l'écran de connexion, comme indiqué dans le capture d'écran suivante:

[277]

Épisode 296

Plugins Firebase

Chapitre 8

Nous avons également ajouté une autre bibliothèque utile, `image_picker`, aux dépendances, afin que nous puissions obtenir une image de la caméra et la télécharger sur Firebase Storage pour l'utiliser comme profil utilisateur image.

Pour vérifier en détail l'utilisation de la caméra et le plugin `image_picker`,
lis [C Chaque](#) *ractéristiques du flutter*
App , en *ation de la caméra du téléphone .*

Nous devons changer notre méthode `_saveProfile ()` dans l'écran de connexion. Ici, nous ajoutons le code nécessaire pour télécharger l'image sélectionnée sur Firebase Storage, et après cela, nous stockons le URL dans les informations de profil de l'utilisateur comme suit:

```
// fait partie de login_page.dart

void _saveProfile () async {
    setState (() {
        _showProgress = true;
    });
}

utilisateur final = attendre FirebaseAuth.instance.currentUser ();

mise à jour finale = UserUpdateInfo ();
updateInfo.displayName = _displayName;
updateInfo.photoUrl = attendre uploadPicture (user.uid);

attendre user.updateProfile (updateInfo);

Navigator.of (contexte) .pushReplacement (
    MaterialPageRoute (
        constructeur: (contexte) => FavorsPage (),
    ),
);
}
```

Comme vous pouvez le voir, la seule chose a été la modification de l'objet `updateInfo` en utilisant sa propriété `photoUrl`. La partie de sauvegarde est toujours la même. `uploadPicture ()` est le partie intéressante:

```
uploadPicture (String userUid) async {
    StorageReference ref = FirebaseStorage.instance
        .ref ()
        .child ('profils')
        .child ('profile_ $ userUid'); // 1

    StorageUploadTask uploadTask = ref.putFile (_imageFile,
```

[278]

Épisode 297

Plugins Firebase

Chapitre 8

```
StorageMetadata (contentType: 'image / png'); // 2
StorageTaskSnapshot lastSnapshot = attendre uploadTask.onComplete; // 3
return wait lastSnapshot.ref.getDownloadURL (); // 4
}
```

La tâche de téléchargement vers Firebase Storage est divisée en petites étapes suivantes:

1. Tout d'abord, nous créons une référence à un nouvel objet sur le **stockage**. Comme vous pouvez le voir, nous chaîner les appels child (), créer un dossier appelé Profiles et un fichier avec l'ID utilisateur en son nom.
2. Après cela, nous créons une tâche de téléchargement de stockage qui initialisera le téléchargement vers Firebase. Notez le paramètre StorageMetadata; nous créons un contenu d'image tapez car c'est une image qui est stockée.
3. Ici, nous attendons la référence future de la tâche de téléchargement, obtenant le dernier instantané de la tâche (le résultat).
4. À la fin, nous obtenons l'URL du fichier; il s'agit d'une URL de téléchargement du fichier Firebase donc que nous pouvons accéder au fichier à partir du stockage.

La liste des fichiers est accessible dans la console Firebase comme indiqué ici:

[279]

Épisode 298

Plugins Firebase

Chapitre 8

Dans la page Faveurs, rien ne change. Comme auparavant, la photo de profil est chargée dans CircleAvatar avec NetworkImage, uniquement si la propriété photoURL de l'ami est donnée (pas nul):

```
// fait partie de la page Favors Classe FavorCardItem
CircleAvatar (
    backgroundImage: favor.friend.photoURL! = null
        ? NetworkImage (
            favor.friend.photoURL,
        )
        : AssetImage ('assets / default_avatar.png'),
),
```

Comme vous pouvez le voir, nous avons une solution de rechange pour le cas d'un utilisateur sans photo de profil. C'est ça pour le stockage dans notre application Favors. Il y a beaucoup de capacités qui doivent encore être explorées.

Dans la section suivante, nous allons explorer le plug-in Firebase AdMob.

Annonces avec Firebase AdMob

Google AdMob est une technologie de publicité mobile pour générer des revenus. Ajout d'annonces à applications est une méthode courante de monétisation et une bonne solution pour les applications gratuites.

Nous pouvons facilement intégrer AdMob dans notre application grâce à l'utilisation des plug-ins FlutterFire. L'enregistrement et l'utilisation d'AdMob sont légèrement différents de ceux des plugins précédents que nous avons vu; nous devons créer un autre compte pour cela.

[280]

Épisode 299

Plugins Firebase

Chapitre 8

Compte AdMob

En réalité, AdMob est séparé de la console Firebase. Bien que nous ayons un AdMob dans la console, nous n'avons que des liens vers des documentations AdMob et page de démarrage:

[281]

Épisode 300

Plugins Firebase

Chapitre 8

dans le apps.admob.com , nous pouvons créer et gérer toutes nos applications.

Notez que les projets Firebase et les applications AdMob ne sont pas explicitement connecté jusqu'à ce que vous liez l'application et le projet / l'application Firebase manuellement. Cela peut changer au moment où vous lisez ce livre. Droite maintenant, tout est séparé: les applications d'AdMob sont enregistrés séparément de Firebase et nous devons les lier manuellement.

Créer un compte AdMob

Dans le lien précédent, nous avons la possibilité de créer notre compte AdSense et AdMob. Vous pouvez suivre les étapes de la page pour créer un nouveau compte comme suit:

[282]

Épisode 301

Après cela, nous sommes prêts à gérer nos applications. Dans le cas de Flutter, nous créons deux applications: une pour Android et une pour iOS:

Nous gérons nos applications et nous obtenons un identifiant d'application unique pour chacune des applications.

La création des applications dans le portail AdMob se fait simplement par en suivant les étapes de configuration. Assurez-vous de créer une application pour chaque Plate-forme.

[283]

Épisode 302

Vous obtiendrez la fenêtre suivante après avoir ajouté avec succès votre application à AdMob:

Nous utiliserons ces identifiants d'application pour afficher des bannières dans notre application.

Après avoir créé l'application AdMob, nous pouvons l'associer dans le portail Google AdMob en tant que montré ici:

[284]

Épisode 303

Plugins Firebase

Chapitre 8

Suivez simplement la procédure de la boîte de dialogue et associez l'application AdMob pour iOS / Android à l'application Firebase correspondante dans le projet, comme illustré dans la capture d'écran suivante:

Cela signifie que les données d'analyse collectées sur Firebase aideront votre AdMob. Cela permet votre flux de données Analytics vers AdMob pour améliorer les fonctionnalités du produit et la monétisation.

AdMob dans Flutter

Comme pour les plugins FlutterFire précédents, nous devons ajouter la dépendance d'AdMob à notre pubspec.yaml, comme suit:

```
dépendances:  
  firebase_admob: ^0.8.0 + 4 # AdMob
```

Après avoir obtenu les dépendances avec les packages de flutter get, nous sommes prêts à utiliser Firebase AdMob dans notre projet.

La classe FirebaseAdMob est notre point de départ pour ajouter des bannières à l'application. contrairement à plugins Firebase déjà vus qui obtiennent toutes les informations nécessaires pour s'exécuter à partir du fichiers google-services.json (Android) et GoogleService-info.plist (iOS), dans ce cas, nous avons besoin d'un paramètre supplémentaire avant de pouvoir utiliser le plugin efficacement.

[285]

Épisode 304

Plugins Firebase

Chapitre 8

Nous devons initialiser manuellement le plugin avec nos ID d'application. Cela peut être fait à tout point. Dans notre application Favors, par exemple, nous pouvons le faire dans la méthode principale comme indiqué ici:

```
void main () {
    FirebaseAdMob.instance.initialize (
        appId: Platform.isAndroid
            ? 'ca-app-pub-3940256099942544 ~ 3347511713' // remplacez par votre
        ID d'application Android
            : 'ca-app-pub-3940256099942544 ~ 1458002511', // remplacez par votre
        ID d'application iOS
    );
    runApp (MyApp ());
}
```

Comme vous pouvez le voir, nous initialisons le plugin en fournissant notre identifiant d'application enregistré (important pour Libération). Dans l'exemple précédent, nous n'utilisons que des ID de test. C'est la même valeur que présent dans la propriété FirebaseAdMob.testAppId de la bibliothèque. Nous pouvons tester nos bannières dans les deux manières suivantes:

En utilisant des annonces de test fournies par Google. Avec cela, nous utilisons un ensemble de publicités simulées, avec pas de trafic réel dans nos annonces applicatives.

Ce paramètre est vraiment important, car il génère un trafic non valide vers nos applications peut entraîner l'invalidation du compte. Assurez-vous donc d'utiliser des annonces de test pendant le développement: en savoir plus ici:<https://developers.google.com/test-firebase/>

[com/](https://developers.google.com/test-firebase/) ^ changer réelle ID d'appareil de test dispo

En ajoutant des appareils de test avec nos vrais ID. C'est l'option préférée, car elle signifie que nous avons le vrai look des publicités.

Lorsque vous utilisez des émulateurs Android ou des simulateurs iOS, ils sont automatiquement configurés comme dispositifs de test. Pour les vrais appareils, la première fois que vous exécutez une application AdMob correctement configurée, l'ID de l'appareil de test apparaîtra dans **LogCat** (Android) ou journal de la **console** (iOS). Utilisez cet identifiant pour marquer votre appareil comme test dispositif. Découvrez plus ici:<https://developers.google.com/test-firebase/>

[ios/](https://developers.google.com/test-firebase/) TF [ps/](https://developers.google.com/test-firebase/) / développement
[TEST-](https://developers.google.com/test-firebase/)

[286]

Épisode 305

Plugins Firebase

Chapitre 8

Note d'accompagnement sur Android

Sous Android, il y a une étape supplémentaire. Nous devons ajouter le même ID d'application AdMob que celui utilisé pour

initialisez le plugin FirebaseAdMob dans AndroidManifest.xml avec le code suivant:

```
<! - AndroidManifest.xml ->
<application>
    <méta-données>
        android: name = "com.google.android.gms.ads.APPLICATION_ID"
        android: value = "ca-app-pub-3940256099942544 ~ 3347511713" />
    </application>
```

Cela se fait en ajoutant la valeur <meta-data> contenant le même ID d'application précédemment configuré.

Note d'accompagnement sur iOS

Dans iOS, nous devons également ajouter le même ID d'application AdMob que celui utilisé pour initialiser FirebaseAdMob plugin dans le fichier Info.plist avec le code suivant:

```
<! - Info.plist ->
<plist version = "1.0">
<dict>
    ...
        <key> GADApplicationIdentifier </key>
        <string> ca-app-pub-3940256099942544 ~ 1458002511 </string> // remplacer par
                                            // votre application iOS
                                            // id
    ...
</dict>
```

Cela se fait en ajoutant une entrée à la section <dict> contenant le même ID d'application qui était précédemment configuré pour iOS.

Affichage des annonces dans Flutter

Après avoir correctement configuré l'initialisation du plug-in AdMob, nous pouvons commencer à afficher différents types d'annonces, comme des **bannières** par exemple. Contrairement à de nombreuses vues Flutter, les annonces sont affiché d'une manière différente des widgets. Ils n'ont pas de noeud dans l'arborescence.

Nous allons modifier notre RequestFavorPageState pour afficher des publicités. Nous afficherons un BannerAd en bas de l'écran et une annonce interstitielle en plein écran après l'enregistrement d'un demande.

[287]

Épisode 306

Plugins Firebase

Chapitre 8

Nous devons conserver une référence aux publicités lorsque nous leur montrons que nous sommes en mesure de les éliminer plus tard. Donc, nous les ajoutons d'abord en tant que champs dans notre état avec le code suivant:

```
// Classe RequestFavorPageState

InterstitialAd _interstitialAd;
BannerAd _bannerAd;
```

Dans la fonction initState (), nous préparons les annonces comme suit:

```
_bannerAd = BannerAd (
    adUnitId: BannerAd.testAdUnitId,
    taille: AdSize.banner,
)
.. load ()
.. show ();

_interstitialAd = Annonce interstitielle (
    adUnitId: InterstitialAd.testAdUnitId,
) .. load ();
```

Vous pouvez vérifier plusieurs types d'annonces ici: <https://paquets.firebaseio/>

Nous avons quelques éléments à prendre en compte lors de la définition des publicités; regarde ça:

adUnitId est la propriété principale d'une annonce, issue de la documentation AdMob:

"Un bloc d'annonces est une ou plusieurs annonces Google affichées à la suite d'un élément du Code d'annonce AdSense."

Nous utilisons testAdUnitId des classes Ad pour créer des publicités simulées; C'est, annonces de test simples. Vous pouvez créer / configurer des blocs d'annonces sur le portail AdMob.

La fonction load () est l'appel de démarrage des annonces; cela rendra l'annonce prête pour affichage.

La fonction show () rend l'annonce visible (en attente si le chargement n'est pas terminé).

Une autre propriété importante est le ciblageInfo; cela nous aide à cibler les publicités. Vérifier la classe MobileAdTargetingInfo pour plus d'informations. Dans cette classe, nous pouvons aussi définir des appareils de test (précédemment mentionnés dans *AdMob dans la section Flutter*).

[288]

Épisode 307

Plugins Firebase

Chapitre 8

Comme vous pouvez le voir, nous affichons la bannière publicitaire au début juste après son chargement. Plus tard dans le save (), l'annonce interstitielle est également affichée avec le code suivant:

```
// méthode de sauvegarde
attendre_interstitialAd.show();
```

Comme vous pouvez le voir, les annonces sont affichées avec une note de test; vous pouvez utiliser de vraies annonces en créant des annonces unités et en utilisant des appareils de test:

Dans la section suivante, nous couvrirons une autre technologie, Firebase ML Kit, qui aide nous pour intégrer des outils d'apprentissage automatique dans nos applications.

[289]

Épisode 308

*Plugins Firebase**Chapitre 8*

ML avec Firebase ML Kit

Firebase ML Kit permet d'ajouter des fonctionnalités ML à notre application sans avoir besoin d'une expérience ML il. Il n'est pas nécessaire d'avoir une connaissance approfondie des réseaux de neurones ou de l'optimisation des modèles pour Commencer.

Firebase ML Kit fournit plusieurs outils, qui sont les suivants:

Reconnaissance de texte (OCR) : reconnaître le texte sur les photos. Disponible sur appareil et fonctionnalité basée sur le cloud.

Détection de visage : détectez les visages sur une image, identifiez les principales caractéristiques du visage et contours des visages détectés. Disponible en tant que fonctionnalité sur l'appareil.

Numérisation de codes-barres : numérissez plusieurs types de codes-barres. Disponible sur appareil.

Étiquetage d'image : reconnaissiez les entités d'une image. Disponible sur appareil et fonctionnalité basée sur le cloud.

Reconnaissance de repère : Reconnaître des repères bien connus dans une image. Disponible en tant que fonctionnalité basée sur le cloud.

Identification de la langue : détermine la langue d'une chaîne de texte. Disponible en fonctionnalité sur l'appareil.

Inférence de modèle personnalisé : Utilisez une commande tensorflow Lite (<https://www.tensorflow.org/Lite>). Disponible en tant que fonctionnalité sur l'appareil.

Les outils sur l'appareil sont des API qui s'exécutent hors ligne et traitent rapidement les données. API basées sur le cloud, d'autre part, comptez sur Google Cloud Platform pour fournir des résultats avec une grande précision.

Ajout du kit ML à Flutter

Comme pour les plugins FlutterFire précédents, nous devons ajouter la dépendance pour ML Kit à notre pubspec.yaml comme suit:

```
dépendances:  
  firebase_ml_vision: ^ 0.6.0 # ML Vision
```

Après avoir obtenu les dépendances avec flutterpackages get, nous sommes prêts à utiliser Firebase Kit ML dans notre projet.

[290]

Épisode 309

*Plugins Firebase**Chapitre 8*

Utilisation du détecteur d'étiquettes dans Flutter

Comme nous l'avons vu, nous avons plusieurs outils fournis par Firebase ML Kit; dans cet exemple, nous exécutera le détecteur d'étiquettes sur l'image, de sorte que l'image sera interprétée et le La bibliothèque nous donnera des informations sur ce que l'image pourrait être. Cela peut être utile pour prétraitement et filtrage des images.

En fonction du service que nous voulons utiliser, nous devons ajouter des bibliothèques spécifiques au niveau du système.

Pour l'étiquetage d'image, nous devons ajouter une bibliothèque d'étiquetage (OCR) au niveau natif de notre projet.

Sous Android, cela se fait dans le fichier android / app / build.gradle, essentiellement par télécharger le code natif qui permet la résolution d'entité dans une image comme suit:

```
dépendances {
    ...
    api 'com.google.firebaseio: firebase-ml-vision-image-label-model: 16.2.0'
}
```

C'est une autre étape, mais elle est facultative. Nous pouvons ajouter ceci à AndroidManifest.xml comme ceci:

```
<application ...>
    ...
    <méta-données
        android: name = "com.google.firebaseio.ml.vision.DEPENDENCIES"
        android: valeur = "ocr" />
    <! - Pour utiliser plusieurs modèles: android: value = "ocr, label, barcode, face" ->
</application>
```

Dans iOS, la base est la même, nous ajoutons cela via des pods (les pods sont équivalents aux plugins dans Battement).

Dans le répertoire ios, exécutez pod init si vous ne disposez pas d'un fichier Podfile.

Remarque: Podfile existerait probablement si vous essayez d'exécuter l'application Flutter sur iOS, comme lors de la construction, il obtiendra les pods correspondants pour les plugins Flutter. Ainsi, Podfile peut déjà avoir du contenu.

Ajoutez ensuite la dépendance pour l'étiquetage d'image dans Podfile avec le code suivant:

```
pod 'Firebase / MLVisionLabelModel'
```

Ensuite, exécutez avec la commande suivante:

installation du pod

[291]

Épisode 310

Plugins Firebase

Chapitre 8

Toute la configuration nécessaire pour chaque technologie peut être vue en détail sur la page du plugin: <https://.. / pub da vision>.

Une fois les dépendances ajoutées, nous pouvons détecter des entités dans une image.

Dans un cas simple, nous détecterons les étiquettes de l'image du profil utilisateur. Ceci est fait par changer le comportement du bouton de capture; après avoir capturé l'image, nous exécutons le code de _labelImage () .

La méthode _labelImage () ressemble à ceci:

```
// fait partie de login_page.dart

_labelImage () async {
    if (_imageFile == null) return;

    setState () {
        _labeling = vrai;
    });

    Final FirebaseVisionImage visionImage =
        FirebaseVisionImage.fromFile (_imageFile); //1

    LabelDetector final labelDetector =
        FirebaseVision.instance.labelDetector (); //2

    List <Label> labels = attendre labelDetector.detectInImage (visionImage);
    //3
```

```
setState () {
    _labels = étiquettes;
    _labeling = faux;
);
}
```

Pour effectuer la détection des entités, nous exécutons quelques étapes:

1. Nous instancions FirebaseVisionImage à partir de l'image capturée
2. Ensuite, nous instancions un Firebase LabelDetector
3. Nous traitons l'image avec LabelDetector; cela renverra une collection de Étiqueter les objets qui seront affichés plus tard

[292]

Épisode 311

Plugins Firebase

Chapitre 8

N'oubliez pas que toutes les informations traitées ont une valeur de confiance associée à il.

Capturer une image simple à partir de l'application d'appareil photo de l'émulateur Android avec une pièce et quelques meubles, nous obtenons quelques étiquettes, comme indiqué ici:

[293]

Épisode 312

Plugins Firebase

Chapitre 8

Comme vous pouvez le voir, il détecte de nombreuses entités de l'image avec une valeur de confiance élevée. Il s'agit d'informations importantes dans l'apprentissage automatique; toutes les valeurs calculées ont une valeur de confiance.

Avec cela, nous concluons l'intégration de l'étiquetage d'image dans notre application.

Sommaire

Dans ce chapitre, nous avons vu les excellents outils Firebase qui nous aident à développer des fonctionnalités complètes applications avec des technologies avancées. Nous avons ajouté l'authentification par téléphone avec code SMS validation sur notre application en utilisant le plugin d'authentification Firebase. Plus tard, nous avons changé la liste des faveurs et fait en sorte que les demandes soient envoyées au service Cloud Firestore. Le stockage Firebase Le plugin a été utilisé pour envoyer des images de profil utilisateur au backend Firebase Storage, où nous pouvons stocker tout type de fichiers à utiliser dans nos applications. En prime, nous avons eu une introduction à le service AdMob avec le plug-in Firebase AdMob et vers ML Kit via Firebase Plugin de vision ML. Nous avons vu comment configurer et gérer nos applications dans Firebase console et le portail AdMob.

Nous pouvons également créer nos propres plugins à utiliser dans nos applications Flutter. Dans le chapitre suivant, nous allons vérifier le processus de création du plugin, de l'implémentation à la publication dans le dépôt de pub.

[294]

Épisode 313

9 Développer votre propre flottement Brancher

Tout comme l'utilisation de plugins de communauté, un développeur peut souhaiter partager du code modulaire utilisable avec la communauté ou l'avoir dans sa propre boîte à outils. De cette façon, la création et le partage de packages est totalement facilité avec le framework Flutter. Dans ce chapitre, vous apprendrez comment créer un petit projet de plugin pour apprendre les bases du processus, en ajouter documentation et publiez-la pour contribuer à la communauté.

Les sujets suivants seront traités dans ce chapitre:

- Créer un projet de package / plugin
- Structure du projet de plug-in
- Documentation dans les packages
- Publier un package
- Recommandations de développement de plugins

Créer un projet de package / plugin

Comme nous l'avons vu, le développement d'applications Flutter complètes repose sur l'utilisation d'un ou plusieurs paquets partagés par la communauté dans les écosystèmes **Flutter / Dart**. Tout développer à partir de zéro ne serait pas pratique pour la plupart des applications, car nous aurions à développer à plusieurs reprises du code spécifique à la plateforme, ce qui allonge le cycle de développement et Ralentissez.

Les écosystèmes Flutter et Dart fournissent des outils pour aider cette contribution à se produire sans des difficultés. Le processus de développement et de publication d'un package se fait dans le Flutter environnement.

Épisode 314

Développer votre propre plugin Flutter

Chapitre 9

Dans ce chapitre, nous allons générer un projet de plugin Flutter simple et analyser son structure. Le plugin généré contient un exemple Flutter qui a une seule méthode pour obtenir la version de la plate-forme, c'est-à-dire la version du système d'exploitation en cours d'exécution. C'est un plugin simple qui n'a rien de spécial, mais est une bonne introduction au plugin projets.

Paquets Flutter par rapport aux paquets Dart

Dans [Chapitre 2](#), *Programmation Dart intermédiaire*, nous avons vu à quoi ressemblent les packages Dart et comment ils sont gérés par l'outil pub. Dans Flutter, ce n'est pas différent; Les packages Flutter sont rien de plus que des packages Dart qui peuvent contenir des fonctionnalités spécifiques à Flutter et donc ont une dépendance sur le framework Flutter.

Il existe deux types de packages Flutter:

Packages Dart : Il existe des packages Dart simples qui peuvent fournir des bibliothèques utiles qui ne dépendent pas du framework Flutter et peuvent donc être utilisés dans n'importe quel Environnement Dart: Web, bureau, serveur, etc. Des packages spécifiques à Flutter avoir une dépendance sur le framework Flutter ne peut être utilisé que dans un contexte Flutter.

Packages de plugins : il existe des packages qui contiennent des implémentations (Java / Kotlin sous Android et ObjC / Swift sous iOS) de ses fonctionnalités, et la partie Dart n'est rien de plus qu'une API qui traduit les appels au Flutter niveau de l'application. Si vous inspectez les emballages utilisés dans nos Faveurs application, comme les packages Firebase ou image_picker, vous verrez sont des packages de plugins qui contiennent des implémentations natives de la plateforme avec une API écrit en Dart.

Démarrer un projet de package Dart

Pour créer un package Dart dans Flutter, nous utiliserons l'outil de création Flutter bien connu. Un des les arguments de cet outil (`--template`) déterminent le type de package que nous sommes

création: un package d'application, un package Dart ou un package de plugin. Nous utilisons le --template argument pour créer un nouveau package Dart:

```
flutter create --template = package simple_package
```

[296]

Épisode 315

Développer votre propre plugin Flutter

Chapitre 9

Cela générera un projet appelé simple_package qui contient un simple package Dart projet. La structure de projet générée est aussi simple qu'un package Dart et n'a pas tout ce qui est spécifique à Flutter:

Comme vous pouvez le voir, il ne contient pas les dossiers Android et iOS typiques, car nous n'avons pas besoin les pour de simples packages Dart.

Même pubspec.yaml n'a rien de spécial, à part le sdk Flutter dépendance:

```
nom: simple_package
description: Un nouveau projet de package Flutter.
version: 0.0.1
auteur:
page d'accueil:

environnement:
  sdk: ">= 2.1.0 <3.0.0"

dépendances:
  battement:
    sdk: scintillement

  dev_dependencies:
    flutter_test:
      sdk: scintillement

battement:
...
...
```

[297]

Épisode 316

Développer votre propre plugin Flutter

Chapitre 9

Si nous voulions que le package ne soit pas spécifique à Flutter, nous pourrions supprimer le Flutter et travaillez dessus comme un package Dart. Comme le dépendance flutter_test, par exemple, n'est pas vraiment nécessaire pour les packages Dart uniquement.

Rappelez -vous : pour de simples paquets Dart, vous pouvez utiliser le <https://GitHub.com/dsim>. Donc, en écrivant des packages spécifiques à Flutter, et nous utilisons l'outil de création Flutter.

Nous n'entrerons pas dans les détails concernant la mise en œuvre de ce type de package, car c'est un simple package Dart.

Parlons maintenant des packages de plugins.

Démarrer un package de plugins Flutter

Pour créer un package de plugin dans Flutter, nous utilisons à nouveau l'outil de création Flutter avec le plugin modèle cette fois:

```
flutter create --template = plugin hands_on_platform_version -a kotlin -i rapide
```

Par défaut, le modèle de plugin utilise ObjC pour iOS et Java pour Android. N'oubliez pas: pour passer à Swift ou Kotlin, vous pouvez spécifier la langue iOS en utilisant l'argument -i et le langage Android en utilisant -a.

Cela générera un projet appelé hands_on_platform_version qui contient un Flutter projet de package. La structure de projet générée est similaire à un package d'application Flutter.

Une structure de projet de plugin

Dans la section précédente, nous avons généré un projet de plugin pour commencer l'analyse. Maintenant, prenons un regarder des parties spécifiques de celui-ci. Le projet est l'exemple de plugin par défaut de Flutter; le seul il renvoie la version du système d'exploitation de la plate-forme du périphérique en cours d'exécution.

[298]

Épisode 317

Développer votre propre plugin Flutter

Chapitre 9

Il y a cependant quelques différences:

Le contenu des dossiers ios / et android / ne contient pas de applications qui démarrent le runtime Flutter. Au lieu de cela, il contient simplement natif classes qui sont des points d'entrée vers des implémentations natives spécifiques. Nous vérifierons ceci en détail plus tard.
Le répertoire example / est un simple package d'application Flutter - oui, un sous-package dans le package du plugin.

lib / hands_on_show_toast.dart est une API Dart pour le plugin:

```
// pubspec.yaml

nom: hands_on_platform_version
description: Un nouveau projet de plugin Flutter.
version: 0.0.1
auteur:
page d'accueil:

environnement:
sdk: ">= 2.1.0 <3.0.0"
```

```
dépendances:  
  battement:  
    sdk: scintillement  
  
dev_dependencies:  
  flutter_test:  
    sdk: scintillement  
  
battement:  
  brancher:  
    androidPackage: com.example.hands_on_platform_version  
    pluginClass: HandsOnPlatformVersionPlugin
```

Comme vous pouvez le voir, le fichier pubspec est également similaire à un simple package d'application Flutter. La différence se trouve dans la section plugin à l'intérieur de la section flutter. Cette partie définit le package comme un package de plugin identifiant le code natif qui composera le réel mise en œuvre dans un contexte de plateforme spécifique.

[299]

Épisode 318

Développer votre propre plugin Flutter

Chapitre 9

MethodChannel

Communication flottante entre le client (Flutter) et l'application hôte (native) se produit via les canaux de la plate-forme. La classe MethodChannel est responsable de l'envoi messages (**invocations de méthode**) côté plateforme. Côté plateforme, Activation de MethodChannel sur Android (API) et FlutterMethodChannel sur iOS (API) recevoir des appels de méthode et renvoyer un résultat:

La technique du canal de plate-forme permet le découplage du code UI de la plate-forme.

code spécifique. L'hôte écoute sur le canal de la plateforme et reçoit un message. Il peut utiliser des API de plateforme pour faire l'implémentation de la logique et renvoyer une réponse au client, la partie Flutter de l'application.

[300]

Épisode 319

Développer votre propre plugin Flutter

Chapitre 9

Pour comprendre comment se déroule l'échange de messages, vous pouvez consulter le <https://.../flutter>

Platform- canaux
types de mess:

Implémentation du plugin Android

Comme nous l'avons vu, le modèle de projet par défaut génère un petit code qui récupère la plateforme version. Jetons un coup d'œil au code généré dans `HandsOnPlatformVersionPlugin.kt`, qui se trouve dans le sous-projet Android `com.example.hands_on_platform_version` paquet. Ce fichier unique est le point d'entrée du plugin:

```
// HandsOnPlatformVersionPlugin.kt

class HandsOnPlatformVersionPlugin: MethodCallHandler {
    object compagnon {
        fun registerWith(registre: registre) {// 1
            val channel = MethodChannel(registre.messenger(),
                "hands_on_platform_version")
            channel.setMethodCallHandler(HandsOnPlatformVersionPlugin())
        }
    }

    override fun onMethodCall(appel: MethodCall, résultatat: Résultat) {// 2
        if (call.method == "getPlatformVersion") {// 3
            result.success("Android ${android.os.Build.VERSION.RELEASE}")
            // 4
        } autre {
            result.notImplemented() // 5
        }
    }
}
```

L'appel d'une méthode de plugin se déroule comme suit:

1. Cette première méthode statique est utilisée par le framework Flutter pour préparer le plugin à être accessible à partir d'un contexte Dart. Il crée essentiellement une instance MethodChannel et définit le gestionnaire de méthode comme classe actuelle. En résumé, il met en place le lien entre Dart et code natif.

[301]

Épisode 320

Développer votre propre plugin Flutter

Chapitre 9

Nous vérifierons le type MethodChannel en détail au [chapitre 13](#), *Améliorer l'Expérience utilisateur*, où nous verrons comment ajouter des codes natifs à l'application projets, pas seulement des packages de plugins.

2. La méthode `onMethodCall` est appelée chaque fois que l'API Dart correspondante a besoin du code natif pour s'exécuter; c'est-à-dire que du côté de Dart, il demandera le framework pour exécuter un code natif avec un nom et des paramètres enregistrés spécifiques.
- Il y a deux arguments dans la méthode:

`MethodCall`: décrit la demande

Résultat: renvoie les résultats à un contexte Dart

3. La première étape pour exécuter un code spécifique est de vérifier ce que l'appelant veut être réalisé. Dans ce cas, il y a une vérification par le nom de la méthode. Un plugin pourrait avoir de nombreuses méthodes; c'est pourquoi il est nécessaire.
 4. En utilisant l'objet `Result`, nous livrons le résultat de la méthode, en utilisant le rappel de succès pour renvoyer la valeur demandée.
 5. Le rappel `notImplemented()`, également de la classe `Result`, peut être utilisé pour informer l'appelant que la méthode demandée n'a pas de correspondant
- la mise en oeuvre. De la même manière, il y a le rappel d'erreur pour la gestion des erreurs.

Implémentation du plugin iOS

Du côté iOS, le code Swift ressemble au code Kotlin:

```
// SwiftHandsOnPlatformVersionPlugin.swift

classe publique SwiftHandsOnPlatformVersionPlugin: NSObject, FlutterPlugin {
    registre public de func statique (avec registrar: FlutterPluginRegistrar) {
        // 1
        let channel = FlutterMethodChannel (nom: "hands_on_platform_version",
                                           binaryMessenger: registrar.messenger ())
        let instance = SwiftHandsOnPlatformVersionPlugin ()
        registrar.addMethodCallDelegate (instance, canal: canal)
    }

    public func handle (_ appel: FlutterMethodCall, résultat: @escaping
                        FlutterResult) {
        résultat ("iOS" + UIDevice.current.systemVersion)
    }
}
```

[302]

Épisode 321

Développer votre propre plugin Flutter

Chapitre 9

Le processus ressemble à Android, mais il existe quelques petites différences:

La méthode `handle` est l'équivalent iOS de `onMethodCall` dans Kotlin. Notez que il ne vérifie pas l'appel de méthode à partir de l'argument `FlutterMethodCall`. Bien que ce soit bien pour un plugin de méthode unique, il est toujours bon de vérifier le caller pour clarifier ce que cela gère.

`FlutterResult` est utilisé pour renvoyer des données au contexte Dart. Il y a aussi types constants pour erreur équivalente et cas non implémentés: `FlutterError` et `FlutterMethodNotImplemented`.

L'API Dart

Donc, maintenant que nous avons vérifié l'implémentation native du plugin, nous devons comprendre comment Flutter communique avec lui à partir du contexte Dart. Le Dart généré Le fichier API `lib / hands_on_platform_version.dart` est le point d'entrée des applications client. Les packages consommateurs importeront cette bibliothèque pour utiliser le plugin. Vérifions le fichier API:

```
// hands_on_platform_version.dart

class HandsOnPlatformVersion {
    statique const MethodChannel _channel =
        const MethodChannel ('hands_on_platform_version'); // 1
```

```

static Future<String> getPlatformVersion() async { // 2
    _channel.invokeMethod('getPlatformVersion'); // 3
    // retourner la version;
}
}

```

La classe HandsOnPlatformVersion est publique, comme vous pouvez le voir, et elle contient un seul méthode qui expose les implémentations natives:

1. La première chose qui est créée est MethodChannel, le pont entre Dart et code de plateforme natif.
2. La méthode platformVersion est exposée aux consommateurs.
3. L'invokeMethod () de MethodChannel est utilisé pour appeler une méthode spécifique par name, getPlatformVersion dans ce cas. Cette méthode se résout en un Avenir avec le résultat du code natif.

[303]

Épisode 322

Développer votre propre plugin Flutter

Chapitre 9

Un exemple de package de plugin

Le répertoire exemple / contient une application Flutter simple qui dépend de la branche. Consultez le fichier pubspec.yaml:

```

// exemple / pubspec.yaml

nom: hands_on_platform_version_example
description: montre comment utiliser le plugin hands_on_platform_version.
publish_to: 'aucun'

environnement:
  sdk: ">= 2.1.0 <3.0.0"

dépendances:
  battement:
    sdk: scintillement

cupertino_icons: ^0.1.2

dev_dependencies:
  flutter_test:
    sdk: scintillement

hands_on_platform_version:
  chemin: ../

battement:
  utilise-matière-conception: vrai

```

Il s'agit d'un fichier pubspec.yaml de l'application Flutter commun, à l'exception du dernier élément sur la liste dev_dependencies. Il y a une dépendance sur le plugin hands_on_platform_version avec la variante de spécification de chemin.

Comme nous l'avons vu au [chapitre 2, Programmation intermédiaire de fléchettes](#), rappelez-vous que vous pouvez spécifier une dépendance de plugin à partir du référentiel pub, des chemins, ou référentiels sources.

Utilisation du plugin

Pour utiliser le package du plugin, nous commençons par l'importer dans nos bibliothèques Dart, comme n'importe quel autre branche:

```
import 'package: hands_on_platform_version / hands_on_platform_version.dart';
```

Épisode 323

Développer votre propre plugin Flutter

Chapitre 9

L'utilisation suit avec l'invocation de la méthode du plugin:

```
attendre HandsOnPlatformVersion.platformVersion;
```

L'exemple complet conserve la version de la plateforme dans le champ `_platformVersion` et appelle le code natif dans la méthode `initPlatformState ()`:

```
Futur <void> initPlatformState () async {  
    String platformVersion;  
    essayez {  
        platformVersion = attendre HandsOnPlatformVersion.platformVersion;  
    } sur PlatformException {  
        platformVersion = 'Impossible d\' obtenir la version de la plate-forme.';  
    }  
  
    if (! monté) return;  
    setState () {  
        _platformVersion = platformVersion; // 4  
    };  
}
```

Nous pouvons souligner quelques points ici:

L'appel de la méthode est asynchrone car les messages de la plateforme sont asynchrones

Les messages de la plate-forme peuvent échouer, nous utilisons donc une exception de plate-forme try / catch qui aide à inspecter les erreurs

Cette vérification permet de supprimer le résultat de la plate-forme si le widget est supprimé de l'arbre d'ici là

L'état est mis à jour afin que le widget soit reconstruit et affiche la plate-forme récupérée version du plugin

Ajout de documentation au package

Les plugins Flutter sont des éléments importants dans le développement d'applications. L'écosystème Flutter se développe et, jour après jour, de tout nouveaux plugins utiles sont partagés avec la communauté. cependant, les plugins utiles doivent clairement décrire comment ils doivent être utilisés correctement. Ceci est fait avec documentation concrète.

Épisode 324

Développer votre propre plugin Flutter

Chapitre 9

Fichiers de documentation

Si vous consultez le site du référentiel pub (pub.dev), vous verrez des informations importantes sur le paquet. Ces informations sont collectées à partir de fichiers spécifiques présents dans le projet:

`pubspec.yaml`: ce fichier contient des détails sur le package:

```
nom: hands_on_platform_version_example  
description: montre comment utiliser la version hands_on_platform_version  
brancher.
```

version: 0.0.1
 auteur: Alessandro Biessek <alessandrobiessek@gmail.com>
 # homepage: la page d'accueil du plugin

Cette information est utile pour que les clients de la bibliothèque sachent qui l'a créée et ce qu'elle Est-ce que.

README.md: Ceci est une courte documentation sur l'utilisation du package et choses importantes

LICENCE: Ceci est la licence pour l'utilisation du package

CHANGELOG.md: Cela enregistre les modifications dans chaque version du package

exemple /: Ceci est un exemple pratique sur la façon d'utiliser le package

Documentation de la bibliothèque

Un autre élément important de la documentation du package est au niveau de Dart. Le consommateur a besoin de connaître chaque méthode disponible, ses arguments et les types de retour pour savoir comment prenez le maximum de la bibliothèque.

Nous allons créer la documentation de la bibliothèque dans les API Dart en ajoutant des commentaires de documentation dans directives de bibliothèques (voir [chapitre 2, Intermediate Dart Programming](#)) avec la syntaxe //:

// Ceci est un commentaire doc et peut être ajouté à n'importe quel membre d'une bibliothèque.

[306]

Épisode 325

Développer votre propre plugin Flutter

Chapitre 9

Cela peut également être appliquée aux membres de la bibliothèque, tels que les méthodes, les variables et les classes. Même *les membres privés* peuvent avoir des commentaires sur la documentation qui peuvent être utiles pour comprendre différentes pièces de la bibliothèque.

Consultez les conseils officiels pour rédiger une bonne documentation pour votre Flutter paquets: <https://.../www.defléchette/> documentation.

Générer de la documentation

Lorsque vous publiez un package, la documentation de l'API est automatiquement générée (tant que vous utilisez le type de commentaire mentionné précédemment) et publié sur [dartdocs.org](#). Vous pouvez, si nécessaire, générer la documentation de l'API localement.

Tout d'abord, configurez l'environnement racine Flutter de la manière suivante:

```
export FLUTTER_ROOT = ~/dev/flutter (sous macOS ou Linux)
```

```
set FLUTTER_ROOT = ~/dev/flutter (sous Windows)
```

Vous pouvez générer une documentation de package en exécutant ce qui suit:

```
cd ~/dev/mypackage
```

```
$ FLUTTER_ROOT/bin/cache/dart-sdk/bin/dartdoc (sous macOS ou Linux)
```

```
% FLUTTER_ROOT%\bin\cache\dart-sdk\bin\dartdoc (sous Windows)
```

Lors de la rédaction de ce livre, il y a un problème ouvert concernant la commande précédente sous Windows: ietez un œil: <https://GitHub.com>

[com/ d](#)

Par défaut, la documentation est générée sous le répertoire doc / api en HTML statique des dossiers.

[307]

Épisode 326

Développer votre propre plugin Flutter

Chapitre 9

Publier un package

La publication d'un package est la dernière étape pour le rendre disponible à la communauté Flutter. Tous les paquets de flutter publication de pub --dry-run

L'argument --dry-run fonctionne comme une étape de pré-publication, où l'outil pub fera un processus de validation mais ne télécharge pas réellement le package. Après que tout va bien, nous peut supprimer la partie --dry-run:

```
flutter packages pub publish
```

Cela publiera efficacement le package sur le site de publication afin que chaque code source soit publié dans le référentiel de pub. Seuls les fichiers cachés et ignorés (en cas d'utilisation de Git) ne sont pas téléchargés.

Vous pouvez en savoir plus sur la commande de publication ici: <https://www.dartlang.org/tools/pub/publish>

Développement de projets de plugins recommandations

Les plugins Flutter sont parfaits pour accélérer le développement d'applications. Contribuer à la communauté en partageant un plugin, c'est aussi super; cependant, il y a peu de points à considérer lors de la planification pour publier un plugin pour le rendre vraiment utile et accepté:

Prise en charge de plusieurs plates - formes : les plugins qui ciblent une seule plate-forme tournent mal le début. Puisque Flutter est un framework multiplateforme, nous devons penser ceci façon, puisque les plugins seront utilisés pour créer des applications qui fonctionneront sur plusieurs plates-formes.

Rédigez une bonne documentation : Flutter fournit des outils pour faciliter la création et publier un package avec toute la documentation; la seule tâche requise est d'écrire ce document.

Recherchez d'abord les plugins existants: vous envisagez peut - être d'en développer un autre plugin sur Flutter, mais vous devriez d'abord chercher dans pub pour vérifier si c'est déjà développé par d'autres développeurs afin que vous puissiez l'utiliser et même y contribuer:

[308]

Épisode 327*Développer votre propre plugin Flutter**Chapitre 9*

Ecrire un bon plugin ciblé peut être très utile aux autres développeurs. N'hésitez pas à consulter le code source du plugin existant et apprenez à créer d'excellents outils pour la communauté.

Sommaire

Dans ce chapitre, nous avons vu à quoi ressemble un package de plugin Flutter et en quoi il est différent que l'application Flutter et les packages Dart simples. Nous avons vu que les plugins Flutter vont jusqu'au code natif en utilisant MethodChannels, qui fournissent de bons mécanismes pour interopérer directement avec le système.

Nous avons vu comment démarrer un projet de package de plugins dans Flutter et comment le documenter correctement le rendre utile et compréhensible pour la communauté. Et enfin, nous avons appris à faire un package public sur le dépôt pub afin que d'autres développeurs puissent l'utiliser.

Dans le chapitre suivant, nous allons continuer à plonger dans le code de plate-forme spécifique en intégrer différentes fonctionnalités propres à chaque système, comme l'importation d'un contact, à l'aide de l'appareil photo et à la gestion des autorisations des applications.

[309]

Épisode 328

dix

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Les applications mobiles ne vivent pas seules dans le contexte de l'appareil et de l'utilisateur, et cela est vrai pour tous les niveaux d'applications, des applications simples à usage unique aux plus complexes

ceux. Une application peut avoir besoin d'accéder à des fonctionnalités matérielles telles que Bluetooth, l'appareil photo, importer un contact pour permettre à l'utilisateur d'interagir avec des amis ou partager du contenu avec d'autres applications et utilisateurs. Ainsi, un développeur doit rendre l'application consciente de l'utilisateur et de l'appareil.

Dans ce chapitre, vous apprendrez à intégrer une application au contexte utilisateur, comme comme l'affichage et le lancement d'une URL, la gestion des autorisations de la plateforme, le lancement d'un téléphone caméra et importation d'un contact.

Les sujets suivants seront traités dans ce chapitre:

- Lancer une URL depuis l'application
- Gérer les autorisations des applications
- Importer un contact depuis un téléphone
- Intégration de la caméra du téléphone

Lancer une URL depuis l'application

Jusqu'à présent, nous avons vu comment nous pouvons utiliser les plugins Flutter pour ajouter des fonctionnalités spécifiques aux applications. Dans l'application Favors, pour l'image du profil de l'utilisateur, par exemple, nous avons utilisé un plugin qui lance l'application caméra et attend un fichier image: le plugin image_picker. Ce plugin agit comme un pont pour nous, et l'application caméra est indépendante du sous-jacent système, car nous n'avons pas besoin de savoir comment lancer l'application appareil photo et comment prendre le image, nous lui demandons juste de faire le travail dur pour nous.

Épisode 329

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Prendre une photo de profil est une bonne utilisation d'un plugin, car dans une future version de l'application, nous pourrions permettre à l'utilisateur d'importer l'image de la galerie et de l'utiliser de la même manière. Le plugin image_picker utilisé fait également ce travail.

Imaginons maintenant un autre cas d'utilisation: un utilisateur demande une faveur à un autre utilisateur qui implique accéder à une URL pour obtenir plus de contexte sur la faveur. Par exemple, si quelqu'un vous demande de acheter un produit sur un site e-commerce, il est bon de partager le lien vers le produit afin que il n'y a pas de malentendu.

L'ajout de la fonctionnalité de *lien ouvert* à l'application peut être effectué à l'aide d'un plugin, url_launcher. Le fait est que pour de nombreuses fonctionnalités de nos applications, nous n'avons pas besoin de savoir comment la plate-forme fonctionne sous le capot, car il existe de nombreux plugins Flutter utiles à notre disposition.

Consultez le code pour lancer les URL depuis l'application sur GitHub dans le répertoire Chapter11 | hands_on_url_handler.

Afficher un lien

Tout d'abord, l'utilisateur doit identifier le lien dans un texte sur lequel cliquer. Dans un contexte mobile, nous avons besoin pour rendre les choses aussi simples que possible, donc, comme vous le savez peut-être, il ne convient pas d'ajouter un autre champ à la demande de faveurs d'ajouter un lien à la faveur. Jetez un œil à une application de chat utilise peut-être en ce moment; vous pouvez y saisir une URL, et lorsque vous l'envoyez à un autre utilisateur, il apparaît automatiquement sous forme de texte cliquable et vous n'avez pas besoin d'effectuer d'action; tu tapez simplement.

La meilleure façon de faire dans l'application est que les liens URL ajoutés à une description de faveur peuvent être transformé en liens cliquables dans les cartes faveurs. Vous pensez peut-être écrire le code avec cette fonctionnalité, car il ne serait pas difficile de faire ce qui suit:

- Analysez la description de la faveur pour les liens trouvés.
- Créez plusieurs TextSpans pour changer son style.
- Manipulez les robinets avec des gestes Flutter.

TextSpans peut être utilisé lorsque nous voulons appliquer différents styles à des parties de texte. Consultez la documentation du widget TextSpan pour en savoir plus
détails: https://api.flutter.dev/flutter/text_selection/TextSpan-class.html.

[311]

Épisode 330

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Bien que ce soit simple, le codage prendra du temps que vous pourriez investir dans l'application. C'est pourquoi il est bon d'utiliser des plugins autant que possible: cela augmente la *productivité*.

Le plugin flutter_linkify

Il existe, bien sûr, un plugin qui fait le travail de styliser les liens dans le texte pour nous, flutter_linkify. Il fait le travail décrit dans la section précédente et nous le présente via le widget Linkify. Il analyse un texte à la recherche de liens et utilise des intervalles pour faire la différence entre le texte simple et les liens et, en prime, il expose des fonctionnalités utiles:

- La propriété onOpen, qui attend un rappel pour gérer un clic sur un lien
- La propriété humanisante, qui affiche un lien sans HTTP / HTTPS

Nous avons modifié notre application Favors pour afficher les liens de la description de la demande dans les cartes de faveur.

La partie requête ne nécessite aucune modification, car l'utilisateur tape le lien normalement dans le texte.

Les modifications pour que les liens apparaissent et soient cliquables sont minimales:

1. Tout d'abord, nous ajoutons le plugin en tant que dépendance:

dépendances:
flutter_linkify: ^2.1.0 # Plug-in Flutter Linkify

2. Après cela, dans le widget FavorCardItem, nous échangeons sa description Text child en le nouveau widget Linkify

Voici à quoi cela ressemblait avant:

```
// dans la méthode de construction de la classe FavorCardItem, favorisez la description
Text(
  favor.description,
  style: bodyStyle,
),
```

Voici à quoi cela ressemblait maintenant:

```
import 'package: flutter_linkify / flutter_linkify.dart'; // importer le plugin
bibliothèque

// dans la méthode de construction de la classe FavorCardItem, favorisez la description
```

[312]

Épisode 331

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

```
Linkify(texte: favor.description,
        humaniser: vrai,
        ),
```

Cela rendra le lien cliquable et avec un style distinct:

Le texte commençant par `http://` ou `https://` apparaît sous forme de lien et est cliquable. La prochaine étape consiste à gérer le clic pour ouvrir l'URL cible.

[313]

Épisode 332

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Lancer une URL

Maintenant que nous avons les liens affichés et exploitables dans l'application, nous devons les rendre travaille correctement. Si vous êtes un développeur Android ou iOS, vous savez peut-être comment lancer un URL, les schémas valides et comment y parvenir. Dans Flutter, comme vous pouvez vous y attendre, ce la fonctionnalité doit être gérée de manière spécifique à la plate-forme.

Vous pouvez consulter les schémas d'URL pris en charge pour chaque plate-forme pour iOS
à: <https://.../dev>
[iPhoneURLSchemas](#)
Android sur: <https://.../dev>
[intents](#) commi

Encore une fois, grâce au travail de la communauté Flutter, nous pouvons faire ce niveau d'intégration à l'aide du plugin `url_launcher` présenté précédemment.

Le plugin `url_launcher`

Le plugin `url_launcher` agit comme un pont vers les gestionnaires de liens natifs de la plate-forme afin que nous vous n'avez pas à vous soucier des détails au niveau de la plate-forme.

L'utilisation du plugin est réduite à quelques fonctions, launch (url) étant la principale. Le La fonction de lancement récupère une URL comme argument et s'occupe du lancement c'est propre à chaque système.

Sous Android, il créera une intention que le système gère via une application de navigateur (ou afficher une vue Web pour les schémas Web si forceWebView est défini sur true). Sous iOS, schéma Web Les URL sont gérées par défaut dans un contrôleur de vue appartenant à l'application.

Nous intégrons le plugin dans la fonction handleLinkClick FavorCardItem, où nous appelez simplement la fonction de lancement (url), en passant l'URL qui provient de Linkify rappeler:

```
// élément de description
Linkify (
    texte: favor.description,
    humaniser: vrai,
    onOpen: handleLinkClick,
),
...
// gestion des clics
handleLinkClick (lien LinkableElement) async {
    if (attendre canLaunch (link.url)) { // 1
```

[314]

Épisode 333

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

```
        attendre le lancement (link.url); // 2
    }
}
```

Comme vous pouvez le voir, le plugin résume une grande partie du travail pour nous. Nous avons juste besoin d'appeler son fonction avec le bon argument:

1. Tout d'abord, nous vérifions si l'appareil est capable de lancer l'URL avec le fonction canLaunch. Cela confirmera que l'appareil a une application installée qui est capable de gérer le schéma d'URL.
2. Enfin, et si possible, nous lançons l'URL; cela enverra l'intention à la plate-forme correspondante.

Pour avoir une idée de ce qui est mis en œuvre sous le capot pour chaque système, je vous recommandons de jeter un œil à la partie native du code source du plugin.

Gérer les autorisations des applications

Les systèmes Android et iOS ont leurs propres politiques de sécurité impliquant des informations utilisateur ou matériel de l'appareil. Le but de l'autorisation est de protéger la vie privée d'un utilisateur. Une application, qu'il soit natif ou non, doit demander l'autorisation d'accéder aux données utilisateur, comme la caméra, pour exemple.

Dans les versions récentes d'iOS, vous devez inclure la description d'utilisation dans le Clés de fichier ios / Runner / Info.plist pour les types de données auxquelles l'application doit accéder, ou crash. Pour accéder à la caméra, par exemple, elle doit inclure NSCameraUsageDescription.

Vous pouvez consulter les autorisations disponibles pour iOS ici: https://developer.apple.com/library/ios/qa/qa1755/_index.html
[développeur](#), Appl
[Références](#) / Info.plist
[ref/](#) do

Sous Android, le fichier android / app / src / main / AndroidManifest.xml est où les autorisations sont répertoriées. Android a le concept des autorisations système en plus de celles des utilisateurs; pour que votre application accède à Internet, par exemple (pour récupérer des données depuis Firebase), elle doit avoir android.permission.INTERNET ajouté par défaut sur le modèle Flutter.

[315]

Épisode 334

*Accéder aux fonctionnalités de l'appareil depuis l'application Flutter**Chapitre 10*

Consultez le guide Android officiel sur les autorisations pour en savoir plus sur comment ils fonctionnent ici sur le système: <https://developer.android.com/guide/topics/permissions/>

[Guide / thé](#)

Ainsi, la principale différence est que, dans Android, chaque ressource basée sur l'utilisateur dispose d'une autorisation, et vous doit l'ajouter au fichier manifeste et également demander l'autorisation en utilisant le système API fournies. Dans iOS, vous devez ajouter une description à chaque ressource sensible à l'utilisateur dans Info.plist afin qu'une invite soit affichée par le système pour que l'utilisateur l'accepte ou la refuse.

Gérer les autorisations sur Flutter

Comme les deux systèmes ont leur propre gestion des autorisations, nous devons en tenir compte lors de l'utilisation de ressources protégées. Dans Flutter, nous devons descendre à la plate-forme level pour demander les autorisations nécessaires.

Comme vous l'avez vu dans notre application Favors, nous ne nous sommes pas inquiétés des autorisations jusqu'à présent; la seul le paramètre existant à ce sujet est le lien dans le fichier AndroidManifest.xml:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hanson">
    <uses-permission android: name = "android.permission.INTERNET" />
    ...
</manifest>
```

L'autorisation Internet n'est pas ajoutée par défaut dans le fichier AndroidManifest fichier. Le framework Flutter l'utilise pour le débogage et le recharge à chaud.

Grâce à la communauté Flutter, nous avons quelques plugins pour nous aider dans cette tâche. Un bien l'exemple est le plugin permission_handler.

[316]

Épisode 335

*Accéder aux fonctionnalités de l'appareil depuis l'application Flutter**Chapitre 10*

Utilisation du plugin permission_handler

Le plugin permission_handler fournit une API de haut niveau pour demander et vérifier le état des autorisations. Le plugin expose un ensemble d'autorisations dans l'énumération PermissionGroup et fait une simplification sur chacun sur ses Plate-forme. Chaque groupe d'autorisations est mappé à l'autorisation correspondante dans le

système. Les principales méthodes fournies par le plugin sont les suivantes:

- requestPermissions: pour demander l'accès à une ressource particulière
- checkPermissionStatus: pour vérifier l'état d'accès à une ressource particulière
- openAppSettings: pour ouvrir les paramètres de l'application afin que l'utilisateur puisse voir / modifier un ressource particulière

Pour Android, il existe également la méthode shouldShowRequestPermissionRationale.

Vous pouvez consulter les méthodes disponibles et la carte des autorisations dans la page de plugin ici: <https://pub.dartlang.org/gestionnaire>.

Importer un contact depuis le téléphone

Du point de vue de l'utilisateur, insérer manuellement un numéro de téléphone pour faire une demande de faveur est pas la méthode préférée, car elle est sujette à des erreurs.

L'importation d'un contact depuis le téléphone de l'utilisateur est une tâche spécifique à la plate-forme qui, bien sûr, similitudes. Le dernier point est de lancer le sélecteur de contact de la plateforme et d'obtenir un seul contact de celui-ci.

Le référentiel pub contient un ensemble de plugins qui vous aident dans cette tâche. Certains d'entre eux sont aussi suivit:

- contact_picker: prend en charge la sélection d'un numéro de téléphone à partir du contact du téléphone liste
- contacts_service: Fournit une API qui nous permet de choisir un contact et aussi les gérer

Si vous vous en souvenez, notre application Favors permet à l'utilisateur de demander une faveur à un autre utilisateur en en ajoutant le numéro de téléphone de l'ami ciblé. Importer le contact depuis le téléphone la liste de contacts est la meilleure façon de le faire.

[317]

Épisode 336

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Importer un contact avec contact_picker

Le plugin contact_picker convient à la tâche, et nous l'utiliserons pour importer contact dans la phase de demande de faveur.

La première étape consiste à inclure le plugin en tant que dépendance dans le fichier pubspec.yaml et à exécuter les packages flutter reçoivent également la commande:

```
dépendances:  
  contact_picker: ^0.0.2
```

Ensuite, nous devons changer l'écran **Demandeur une faveur**. Nous ajoutons un bouton **Importer** à droite côté de la liste déroulante de l'ami:

[318]

Épisode 337

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Dans l'action onPressed du bouton d'importation, nous allons rediriger l'utilisateur vers le l'écran des contacts pour qu'ils puissent sélectionner un contact.

Jetons un coup d'œil au code. Tout d'abord, nous ajoutons deux champs à RequestFavorPageState classe:

```
// request_favors_page.dart
```

```
La classe RequestFavorPageState étend l'état <RequestFavorPage> {
  Final ContactPicker _contactPicker = ContactPicker ();
  Friend _importedFriend;
  ...
}
```

Voici en quoi les deux champs nous aideront:

```
_contactPicker fournit la fonctionnalité du plugin.
_importedFriend stocke l'ami importé à partir des contacts, le cas échéant.
```

Avec cela, nous pourrons importer un contact facilement. Après cela, nous ajoutons le onPressed callback pour le bouton **Importer contact**:

```
 onPressed: () {
    _importContact ();
},
```

Ensuite, nous importons un contact en utilisant la méthode _importContact ():

```
void _importContact () async {
  Contact contact = await _contactPicker.selectContact (); // 1
  if (contact! == null) {
    setState ( () {
      _importedFriend = Ami (
        nom: contact.fullName,
        numéro: contact.phoneNumber.number,
      ); // 2
    });
  }
}
```

L'importation d'un contact s'effectue en quelques étapes:

1. Tout d'abord, nous lançons le sélecteur de contact en utilisant la méthode selectContact de la classe ContactPicker du plugin.
2. Après avoir vérifié que l'utilisateur a sélectionné un contact (contact! = Null), nous créons une nouvelle instance Friend basée sur les informations de contact sélectionnées.

[319]

Épisode 338

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

La dernière étape consiste à gérer la sauvegarde de la faveur où nous devons obtenir les informations de l'ami de `_importedFriend`, comme nous l'avons fait avec `_selectedFriend` dans la liste déroulante de l'ami liste:

```
void save (context BuildContext) async {
    ...
    attendre _saveFavorOnFirebase (
        Favoriser(
            à: _importedFriend? .number ?? _selectedFriend? .number,
            ...
        )
    )
    ...
}
```

La seule modification nécessaire était dans la propriété 'to' de la nouvelle Faveur qui pointera vers la valeur `_importedFriend` ou `_selectedFriend`.

Comme vous le pensez peut-être, les contacts téléphoniques sont une ressource utilisateur et sont donc protégés information. L'utilisateur doit autoriser l'application à lire ou à écrire des contacts.

Autorisation de contact avec permission_handler

Bien que les informations sur les contacts soient une ressource protégée par l'utilisateur, nous n'avons pas besoin de permission d'importer un contact en utilisant le plugin `contact_picker` car nous ne sommes pas le lire directement, mais via des API spécifiques à la plate-forme.

Nous allons voir, cependant, comment demander l'autorisation d'utiliser les contacts, car cela peut être utile A l'avenir.

Si vous vous en souvenez, chaque plateforme a sa propre façon de gérer les autorisations, et nous devons implémenter les demandes d'autorisation basées sur cela.

Autorisation de contact sur Android

Dans Android, nous devons ajouter la demande d'autorisation de contact dans le fichier `AndroidManifest`, donc修改ons le fichier `android / app / src / (main | debug | profile) / AndroidManifest.xml` fichier:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hanson">
    ...
    <uses-permission android: name = "android.permission.READ_CONTACTS" />
```

[320]

Épisode 339

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

```
<uses-permission android: name = "android.permission.WRITE_CONTACTS" />
</manifest>
```

En ajoutant l'autorisation `READ_CONTACTS`, nous déclarons au système Android que nous devons accéder à la liste de contacts des utilisateurs; `WRITE_CONTACTS`, comme vous l'avez peut-être déduit, déclare le besoin d'écrire de nouveaux contacts dans le système.

Le comportement de cet enregistrement dépend de la version du système de l'application est installé sur [Découvrez ceci ici](https://developer.android.com/develop/ui/views/available-display-sizes) / développeur andr formation / autor

Autorisation de contact sur iOS

Sous iOS, nous devons fournir une description appropriée du fichier Info.plist afin que le utilisateur sait pourquoi l'application a besoin de l'autorisation demandée. Ceci est fait dans le Fichier ios / Runner / Info.plist:

```
<dict>
...
<key> NSContactsUsageDescription </key>
<string> Vous pouvez importer un ami à partir d'une liste de contacts. </string>
</dict>
```

Lorsque l'application essaie d'accéder aux contacts dans iOS, le système demande l'autorisation à l'utilisateur montrant la description fournie pour aider à l'acceptation.

Vérification et demande d'autorisation dans Flutter (permission_handler)

Supposons que notre application ait besoin d'une autorisation pour accéder aux contacts afin de faire une demande de faveur (qui c'est-à-dire que cela serait vrai si nous voulions afficher tous les contacts de notre application pour que l'utilisateur puisse sélectionnez-en un). Nous créons la fonction _checkPermissions pour vérifier et demander le autorisation, si nécessaire, et procédez comme suit:

1. Tout d'abord, nous obtenons le statut de l'autorisation de l'API:

```
void _checkPermissions () async {
    Statut PermissionStatus = attendre PermissionHandler ()
        .checkPermissionStatus (PermissionGroup.contacts);
```

[321]

Épisode 340

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

2. Ensuite, nous testons si le statut est différent de celui accordé, c'est-à-dire qu'il a pas déjà accordé par l'utilisateur:

```
if (status! = PermissionStatus.granted)
```

3. Enfin, si l'autorisation n'est pas accordée (statut! = PermissionStatus.granted), nous le demandons:

```
attendre
PermissionHandler (). RequestPermissions ([PermissionGroup.contacts]);
}
```

En résumé, _checkPermissions obtiendra le statut d'autorisation actuel, et si ce n'est pas le cas accordé, il le demandera. Un endroit approprié pour appeler cette fonction est dans le **Contact** bouton d' **importation** , avant d'importer le contact:

```
void _importContact () async {
    attendre _checkPermissions ();
    ...
}
```

Dans notre cas, le résultat de _checkPermissions () n'est qu'illustratif, car nous n'avons pas besoin la permission.

Intégration de la caméra du téléphone

La fonction de caméra est présente dans de nombreuses applications et son intégration peut être effectuée en quelques façons. Nous pourrions, par exemple, implémenter le code nous-mêmes, mais grâce au communauté, Flutter fournit plusieurs plugins pour accéder à la caméra. Certains des meilleurs les plugins connus sont les suivants:

caméra: Avec ce plugin, nous pouvons afficher l'aperçu de la caméra directement sur Flutter,

prendre des photos ou enregistrer des vidéos.

`image_picker`: Ce plugin essaie de simplifier beaucoup la tâche; on lui demande seulement de donner nous une photo de la caméra ou de la galerie, et il s'occupe du reste.

Si vous vous souvenez, dans [Chapitre 8](#), *Plugins Firebase*, nous avons réussi à envoyer une photo de profil utilisateur à Firebase Storage, et nous avons utilisé le plugin `image_picker` pour obtenir un fichier image depuis l'appareil photo. Alors, passons en revue comment cela fonctionne en détail.

[322]

Épisode 341

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Prendre des photos avec `image_picker`

Flutter ne communique pas directement avec l'API de la caméra, car il s'agit d'un niveau de plate-forme Ressource. Le plugin `image_picker`, comme son nom l'indique, aide à choisir une image. Il permet d'importer des fichiers image depuis la galerie et de prendre de nouvelles photos à l'aide de l'appareil photo.

Tout d'abord, nous ajoutons la dépendance au fichier `pubspec.yaml` et l'obtenons avec le flutter les packages reçoivent la commande:

```
dépendances:  
  image_picker: ^0.5.0 # Sélecteur d'images
```

Nous contrôlons la sélection d'images au même endroit lorsque l'utilisateur entre son nom d'affichage après login, dans la dernière étape du widget Stepper. Lorsque l'utilisateur appuie sur le petit avatar image, l'appareil photo s'ouvre pour prendre une photo:

```
// login_page.dart  
  
// fait partie de la classe LoginScreenState  
void _importImage () async {  
    image finale = attendre ImagePicker.pickImage (source: ImageSource.camera);  
    setState ((){  
        _imageFile = image;  
    });  
}
```

Ceci est fait avec la classe `ImagePicker`. Nous utilisons sa méthode `pickImage()` pour démarrer le appareil photo et prenez une photo (le tout géré par le plugin) qui résout l'image capturée en un fichier pour notre usage.

Vous pouvez trouver le code source de `login_page.dart` sur GitHub pour une exemple d'utilisation du plugin `image_picker`. Aussi, il est important que vous après la documentation à du plugin https://pub.dartlang.org/packages/image_picker.

[org/_pa](#) écessite une configuration pour fo.....

Autorisation de la caméra avec `permission_handler`

Le plugin gère lui-même les demandes d'autorisation, mais, dans ce cas, nous utiliserons à nouveau le plugin `permission_handler` pour vérifier et demander la permission à la caméra.

[323]

Épisode 342

Autorisation de la caméra sur Android

Sous Android, nous devons déclarer l'autorisation de la caméra dans le fichier AndroidManifest comme nous l'avons fait pour les contacts, donc on change le fichier android / app / src / (main | debug | profile) /AndroidManifest.xml:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hanson">
    ...
    <uses-permission android: name = "android.permission.CAMERA" />
</manifest>
```

En ajoutant l'autorisation CAMERA, nous déclarons au système Android que nous devons accéder l'appareil photo. De plus, nous pouvons utiliser une autre balise de manifeste Android:

```
<manifest ...>
    <utilise-fonctionnalité
        android: name = "android.hardware.camera"
        android: requis = "faux" />
</manifest>
```

La balise uses-feature déclarera que notre application a besoin de la caméra pour fonctionner correctement (encore une fois, dans notre cas, ce n'est pas vraiment vrai. L'argument requis peut être défini sur true si nécessaire). Si vrai, l'application ne sera disponible que pour les appareils contenant une caméra disponible.

Autorisation de la caméra sur iOS

Comme nous l'avons fait pour les contacts, dans iOS, nous devons fournir une description appropriée dans le fichier Info.plist afin que l'utilisateur sache pourquoi l'application a besoin de l'autorisation de la caméra. Reportez-vous au code du fichier ios / Runner / Info.plist:

```
<dict>
    ...
    <key> NSCameraUsageDescription </key>
    <string> Vous pouvez ajouter une photo de profil directement depuis la caméra </string>
    <key> NSPhotoLibraryUsageDescription </key>
    <string> Cette application nécessite l'accès à la photothèque. </string>
    <key> NSMicrophoneUsageDescription </key>
    <string> Cette application ne nécessite pas l'accès au microphone. </string>
</dict>
```

[324]

Épisode 343

Lorsque l'application essaie d'accéder à la caméra sous iOS, le système demande l'autorisation des utilisations, montrant la description fournie afin que l'utilisateur donne la permission d'accéder au caméra.

Demander l'autorisation de la caméra dans Flutter (`permission_handler`)

Dans la configuration du profil après la connexion, nous pouvons ajouter une photo de profil. Pour demander l'autorisation de accéder à la caméra, le processus est très similaire à la demande d'accès aux contacts de l'utilisateur.

Nous créons une fonction pour vérifier et demander l'autorisation, si nécessaire:

```
void _checkPermissions () async {
    Statut PermissionStatus = attendre PermissionHandler ()
        .checkPermissionStatus (PermissionGroup.camera); // 1
    if (status! = PermissionStatus.granted) { // 2
```

```

attendre PermissionHandler ().requestPermissions ([PermissionGroup.camera]);
// 3
}
}

```

La méthode est très similaire à la vérification effectuée dans l'exemple d'importation de contact:

1. Nous obtenons le statut de l'autorisation de la caméra de l'API.
2. Nous testons si le statut est différent de celui accordé (si l'autorisation était déjà accordé par l'utilisateur).
3. Enfin, si l'autorisation n'est pas accordée, nous la demandons.

Un endroit approprié pour appeler cette fonction est dans l'image de profil en choisissant une scène, à l'intérieur la méthode `_importImage ()`:

```

void _importImage () async {
  attendre _checkPermissions ();
  ...
}

```

Bien que nous ayons besoin de l'autorisation de la caméra, le plugin `image_picker` a déjà demandé cela pour nous, donc cela fonctionnera aussi.

[325]

Épisode 344

Accéder aux fonctionnalités de l'appareil depuis l'application Flutter

Chapitre 10

Sommaire

Dans ce chapitre, nous avons vu comment utiliser des plugins pour utiliser les fonctionnalités du téléphone telles que l'appareil photo, contacts et lancer une URL. Nous avons vu que la communauté Flutter fournit un ensemble de plugins pour toutes les fonctionnalités nécessaires.

Nous avons utilisé les plugins `url_launcher` et `flutter_linkify` pour afficher un lien vers l'utilisateur dans la description de l'application Favor. Après cela, nous avons ajouté le plugin `permission_handler` à gérer les autorisations des applications. Nous avons également utilisé le plugin `contact_picker` pour importer un contact à partir de la liste de contacts de l'utilisateur, et, en utilisant le plugin `permission_handler`, nous avons ajouté une vérification et demande d'autorisation de contact.

Plus tard, le plugin `image_picker` a été utilisé de la même manière pour récupérer le profil de l'utilisateur image lors de la connexion, et, encore une fois, nous avons utilisé le plugin `permission_handler` pour vérifier et demander l'autorisation de la caméra.

Dans [Chapitre 11 , Vues de la plateforme et intégration de la carte](#) , nous continuerons d'intégrer Flutter plugins. Cette fois, nous verrons comment utiliser les cartes, alors continuez à lire pour consulter la carte intégration avec les applications Flutter.

[326]

Épisode 345

11

Vues et carte de la plate-forme L'intégration

L'affichage de cartes est une fonctionnalité qui apparaît fréquemment dans les applications de nos jours car de nombreux les applications mobiles reposent sur le positionnement de l'utilisateur et la localisation des lieux pour fournir des outils aider à de nombreuses tâches et activités, telles que la recherche de lieux spécifiques, la conduite, le cyclisme, le transport et le transit. Dans ce chapitre, vous apprendrez à intégrer Google Maps dans Flutter applications afin d'ajouter des marqueurs et des interactions pour les rendre pleinement interagissant avec des lieux intéressants à l'aide de l'API Google Places.

Les sujets suivants seront traités dans ce chapitre:

- Afficher une carte
- Ajouter un marqueur à la carte
- Ajout d'interactions cartographiques
- Utilisation de l'API Google Places

Épisode 346

Vues de la plateforme et intégration de la carte

Chapitre 11

Afficher une carte

L'affichage d'une carte dans l'application est la première étape pour en faire une application basée sur une carte, donc Commençons par créer une application qui affiche une carte et nous y ajouterons plus tard des fonctionnalités. Le framework Flutter ne contient pas de widget de carte directement dans son SDK principal; c'est pris en charge à la place avec le plugin officiel `google_maps_flutter`, que nous utiliserons pour afficher une carte comme celle-ci:

[328]

Épisode 347

Vues de la plateforme et intégration de la carte

Chapitre 11

Au moment de la rédaction de ce livre, `google_maps_flutter` est en préversion pour les *développeurs*; C'est, le plugin *s'appuie* sur le nouveau mécanisme de Flutter pour intégrer les vues Android et iOS et, comme ce mécanisme est actuellement dans l'aperçu des dévelopeurs, ce plugin devrait également être pris en compte dans l'aperçu des dévelopeurs.

L'affichage d'une carte dans les applications Flutter nécessite quelques ajustements de l'application par défaut. Donc, commençons par comprendre quels sont ces ajustements, puis ajoutons le support à la plate-forme vues.

Vues de la plateforme

PlatformView de Flutter est un widget qui intègre une vue native Android / iOS et l'intègre dans l'arborescence des widgets Flutter. Les vues de plate-forme sont des widgets avec état qui contrôlent ressources associées à la vue native de la plateforme. Quant à l'incorporation, ce type de vue est une tâche coûteuse, il doit donc être utilisé avec prudence, et seulement si cela est vraiment nécessaire. Vous pouvez utiliser-le pour afficher des cartes, par exemple, car Flutter n'a pas de widget équivalent qui affiche une carte seule.

Les vues de plate-forme sont des éléments importants dans les frameworks tels que Flutter, car ils vous permettent de combler certaines lacunes au cours de l'évolution du cadre. Cependant, il y a quelques points associés à ceci que vous devrez peut-être prendre en compte avant de l'utiliser:

Sur Android, il nécessite le niveau d'API 20 ou supérieur

Sur iOS, certaines étapes supplémentaires sont nécessaires pour configurer la fonctionnalité (*voir ce qui suit sections*)

Encore une fois, l'incorporation de vues est coûteuse pour le cadre et doit être évitée dès que possible

PlatformView remplit tout l'espace disponible du parent, comme le widget de conteneur

PlatformView participe à l'arborescence des widgets comme n'importe quel autre widget

Cette fonctionnalité a été présentée lors de la version Flutter 1.0 et, à l'époque d'écriture, évolue toujours sur les plates-formes Android et iOS, donc continuez à suivre son statut sur les problèmes liés au référentiel Flutter: <https://github.com/flutter/flutter/issues>

[GitHub](https://github.com/flutter/flutter/issues). com

[329]

Épisode 348

Vues de la plateforme et intégration de la carte

Chapitre 11

Activation des vues de plate-forme sur iOS

Dans les premières versions des fonctionnalités d'affichage de la plate-forme, il n'était pris en charge que sur Android. À au moment de la rédaction de ce livre, l'implémentation iOS de l'incorporation d'UIKitView est toujours en cours *aperçu de la version*. Nous devons donc modifier le fichier ios / Runner / Info.plist de l'application et ajouter un paramètre spécifique:

```
<plist version = "1.0">
<dict>
    ...
    <key> io.flutter.embedded_views_preview </key>
    <string> OUI </string>
</dict>
</plist>
```

Cela activera la fonctionnalité pour les applications iOS afin que nous puissions utiliser la fonction de prévisualisation dans notre application.

Une liste des problèmes en suspens concernant l'incorporation de vues iOS est disponible sur GitHub: <https://github.com/flutter/flutter/issues>

Créer un widget de vue de plate-forme

Lorsque nous créons un widget de vue de plate-forme, nous créons essentiellement un wrapper Flutter d'un natif Vue iOS / Android. Le processus de création d'une vue de plate-forme est similaire aux plugins et nécessite l'ajout de code natif à une application.

Pour garder les choses simples, nous créons un projet de plugin; voir [Chapitre 9, Développer le vôtre Flutter Plugin](#), pour se rappeler comment créer un projet de plugin. Dans ce projet, nous définissons un nouveau view, HandsOnTextView, qui est une vue d'affichage de texte natif; TextView sur Android et UITextView sur iOS.

Vérifiez, le fichier hands_on_platform_views sur GitHub pour le code du plugin.

[330]

Épisode 349

Vues de la plateforme et intégration de la carte

Chapitre 11

Pour commencer, après la création du projet de plugin, nous définissons l'API Dart. C'est le code qui fait le pont de Dart au code natif. Nous créons un widget HandsOnTextView.

Comme vous pouvez le voir, sa méthode de construction comprend les parties importantes suivantes:

Selon le type de plateforme, Theme.of(context).platform, nous instanciez un widget AndroidView ou UIKitView.

Leurs propriétés sont similaires, et nous définissons le widget viewType que nous voulons créer, ses paramètres (creationParams) et les paramètres codec (creationParamsCodec):

viewType: un type de vue est utilisé par la vue de la plateforme Flutter système pour indiquer la vue native que nous avons l'intention d'utiliser, similaire à un système de plugins.

creationParams: ce sont les arguments que nous voulons passer jusqu'à la création de la vue native - le texte à afficher, dans notre Cas.

creationParamsCodec: Ceci définit quelle méthode de paramètre le transfert de données se produira lors de l'envoi de creationParams au code natif.

Tout cela pour le côté Dart de la vue de la plate-forme. Nous devons maintenant définir la vue dans plates-formes correspondantes.

Au [chapitre 13, Amélioration de l'expérience utilisateur](#), nous vérifierons comment ajouter des code à l'application. Vous pouvez également y trouver des informations utiles pour vous aider à comprendre le fonctionnement de la vue de la plateforme.

Créer une vue Android

La création et l'enregistrement des vues de plate-forme sur chaque plate-forme est un processus très similaire; nous venons de gérer les différences dans les langues et les API de vue native. Le moyen le plus simple de démarrer la création de vues de plate-forme consiste à l'enregistrer dans le registre des vues de plate-forme, très similaire à ce qui est fait lors de la création d'un plugin Flutter. De plus, comme nous avons affaire à un projet de plugin, cela se fait avec l'enregistrement du plugin:

```
class HandsOnPlatformViewsPlugin {
    objet compagnon {
        @JvmStatic
        fun registerWith(registraire: registraire) {
            greffier
            .platformViewRegistry()
```

[331]

Épisode 350

Vues de la plateforme et intégration de la carte

Chapitre 11

```
.registerViewFactory(
    "com.example.hanson / textView",
    HandsOnTextViewFactory());
```

Nous enregistrons une fabrique de vues en l'identifiant avec un *type / clé*, de sorte que, lors de l'instanciation d'un

vue plate-forme, le moteur Flutter est capable de trouver l'usine correspondante et de déléguer la vue sa création. La fabrique de vues, en passant, est responsable de l'instanciation des vues depuis types spécifiques. Comme vous pouvez le voir, nous avons enregistré une fabrique de vues pour le com.example.hanson / type textView. Nous obtenons l'instance PlatformViewRegistry avec la méthode platformViewRegistry (), et à travers elle, nous avons ajouté notre usine au registre donc quand quelqu'un demande le type enregistré, la construction sera déléguée à cette instance d'usine HandsOnTextViewFactory.

HandsOnTextViewFactory se présente comme suit:

```
classe HandsOnTextViewFactory:  
PlatformViewFactory (StandardMessageCodec.INSTANCE) {  
  
    override fun create (context: Context, id: Int, args: Any): PlatformView  
{  
    val params = args as Map <String, Any> // 1  
  
    val text = if (params.containsKey ("text")) {// 2  
        params ["text"] comme chaîne? ?: ""  
    } autre ""  
  
    return HandsOnTextView (contexte, texte) // 3  
}  
}
```

La classe de fabrique doit étendre PlatformViewFactory et implémenter la méthode de création. Cette méthode est responsable de la création du type de vue spécifié, qui va comme suit:

1. Il reçoit des arguments en tant que paramètre et peut l'utiliser pour configurer la vue
2. Il obtient la valeur de texte d'une carte reçue dans le paramètre
3. Enfin, il renvoie une instance HandsOnTextView

Notez la valeur StandardMessageCodec.INSTANCE transmise à la classe parente du usine. Cela doit avoir le même type de creationParamsCodec défini dans Dart, donc le framework est capable de transférer les arguments de Dart-side vers natif.

[332]

Épisode 351

Vues de la plateforme et intégration de la carte

Chapitre 11

La classe HandsOnTextView est la classe de vue native:

```
classe constructeur interne HandsOnTextView (contexte: contexte, texte: chaîne)  
: PlatformView {  
    valeur privée textView: TextView = TextView (contexte)  
  
    init {  
        textView.text = texte  
    }  
  
    remplacer fun getView (): View {  
        return textView  
    }  
  
    remplacer fun dispose () {}  
}
```

Comme vous pouvez le voir, il doit implémenter l'interface PlatformView du framework. L'interface nécessite deux méthodes, getView et dispose:

getView () doit renvoyer une vue Android à incorporer dans le contexte Flutter.
La méthode dispose () est appelée lorsque la vue est détachée du Flutter
le contexte. Nous pouvons l'utiliser pour effacer n'importe quelle ressource ou référence afin d'éviter la mémoire fuites.

Créer une vue iOS

Sous iOS, le processus est très similaire à Android, mais il y a quelques points sur la syntaxe qui

```
différer. Nous enregistrons l'usine comme nous l'avons fait précédemment, dans la section Créer une vue Android :
classe publique SwiftHandsOnPlatformViewsPlugin: NSObject, FlutterPlugin {
    registre public de func statique (avec registrar: FlutterPluginRegistrar) {
        laissez viewFactory = HandsOnTextViewFactory ()
        registrar.register (viewFactory, withId: "com.example.handsontextview")
    }
}
```

Ensuite, nous rendons la classe HandsOnTextViewFactory capable de renvoyer la version iOS du vue:

```
classe publique HandsOnTextViewFactory: NSObject, FlutterPlatformViewFactory {

    public func create (
        avec cadre: CGRect,
        viewIdentifier viewId: Int64,
```

[333]

Épisode 352

Vues de la plateforme et intégration de la carte

Chapitre 11

```
arguments args: Tout?
) -> FlutterPlatformView {
    return HandsOnTextView (frame, viewId: viewId, args: args)
}

public func createArgsCodec () -> FlutterMessageCodec & NSObjectProtocol
{
    retourne FlutterStandardMessageCodec.sharedInstance ()
}
}
```

Ici, l'usine doit implémenter le protocole FlutterPlatformViewFactory, avec les deux les méthodes create et createArgsCodec:

create () doit renvoyer une vue iOS à incorporer dans le contexte Flutter, comme getView () de la version Android.

createArgsCodec () doit renvoyer le version creationParamsCodec correspondante. Comme nous l'avons fait plus tôt, nous utilisons le codec standard, FlutterStandardMessageCodec.sharedInstance (), sous iOS.

Dans notre cas, car nous ne passons qu'une chaîne du côté natif. Nous pourrions avoir utilisé StringCodec comme codec de message mais, pour le bien de notre exemple, nous avons utilisé le codec standard à la place.

Vérifiez le document codec de message. https://../.docs/flutter/services/Media_CODEC

Voyons maintenant comment utiliser le widget de plate-forme.

Utilisation d'un widget de vue de plate-forme

L'utilisation d'un widget de plateforme est aussi simple que d'utiliser un widget ordinaire. Séparé de configuration spécifique préalablement prévue pour la plateforme iOS, il n'y a rien de plus nécessaire. Nous l'utilisons simplement comme un widget normal:

```
@passer autre
Construction du widget (contexte BuildContext) {
    retour MaterialApp (
        accueil: Container (
            alignment: Alignment.center,
            couleur: Colors.red,
            enfant: SizedBox (
```

[334]

Épisode 353

Vues de la plateforme et intégration de la carte

Chapitre 11

```
hauteur: 100,
enfant: HandsOnTextView (
    texte: "Texte de la vue Plate-forme",
),
),
);
}
```

Consultez l'exemple `hands_on_platform_views` sur GitHub pour code complet du plugin.

Après cela, le widget de la plate-forme ressemble à n'importe quel autre widget:

[335]

Épisode 354

Vues de la plateforme et intégration de la carte

Chapitre 11

L'emballage de la vue de la plate-forme dans `SizedBox` limite ses dimensions; sinon, il faudrait tout l'espace disponible. Cependant, ce n'est pas obligatoire; `AndroidView` et

Les classes `UIKitView` sont chargées de rendre les vues de la plateforme présentes dans le widget hiérarchie dans d'autres widgets.

Il est important de noter que l'intégration de vues de plate-forme est une opération coûteuse fonctionnement car le moteur Flutter doit gérer les ressources requises par chacune d'entre elles. Par conséquent, l'utilisation des vues de plate-forme doit être évitée lorsqu'un Flutter l'équivalent est possible.

Premiers pas avec `google_maps_flutter`

brancher

Comme dit précédemment, le plugin google_maps_flutter s'appuie sur les vues de la plateforme pour afficher cartes sur les applications Flutter, comme vous l'avez vu dans la section précédente.

Comme la fonctionnalité de vues de plate-forme, ce plugin est toujours en évolution active donc vous devrez peut-être vérifier les changements dans la page du plugin: https://pub.dev/packages/google_maps_flutter

dartlang.org/p/

Le plugin expose le widget GoogleMap, et c'est tout ce dont il s'agit. En plus de cela, le widget expose les fonctionnalités de carte communes qui sont importantes pour le rendre pleinement personnalisable et interactif. Les principaux sont les suivants:

mapType: Ceci permet de changer le style des tuiles de la carte à afficher, pour exemple, MapType.normal affiche des informations sur le trafic et le terrain et MapType.Satellite affiche des photos aériennes.

Vérifiez tous les types disponibles dans la documentation MapType

page: https://pub.dev/packages/google_maps_flutter

[flutter/latest/](https://flutter.dev/docs/development/packages-and-plugins/introduction)

[336]

Épisode 355

Vues de la plateforme et intégration de la carte

Chapitre 11

marqueurs: cela nous permet d'ajouter des marqueurs en haut de la carte (voir la rubrique *Ajout de marqueurs de la section de la carte*).

myLocationEnabled: Ceci permet d'activer la couche **My Location** sur la carte. Il permet également d'afficher un indicateur à l'emplacement actuel de l'appareil comme un bouton **Ma position** pour que l'utilisateur puisse se concentrer sur le courant emplacement connu, si possible.

L'activation de **Ma position** nous oblige à ajouter également des autorisations de localisation à les deux plates-formes natives de notre application. Consultez le chapitre précédent *Gestion des autorisations de l'application* pour vous rappeler comment le faire.

initialCameraPosition: il s'agit de configurer la partie visible initiale du carte.

cameraTargetBounds: Ceci permet de changer le cadre de délimitation géographique pour le cible de la caméra, c'est-à-dire la partie focalisée de la carte.

rotateGesturesEnabled, scrollGesturesEnabled, tiltGesturesEnabled , et zoomGesturesEnabled: ils activent / désactivent les gestes correspondants.

Ce plugin expose également des rappels pour que nous puissions répondre à des événements de carte spécifiques:

onMapCreated: appelé lorsque la carte est structurellement prête

onTap: appelé lorsqu'un tap se produit sur la carte

onCameraMoveStarted, onCameraMove et onCameraIdle: appelé le événements de caméra correspondants

Vous pouvez vérifier toutes les propriétés disponibles de la classe GoogleMap sur https://pub.dev/packages/google_maps_flutter

[latest/googl](https://flutter.dev/docs/development/packages-and-plugins/introduction)

[337]

Épisode 356

Vues de la plateforme et intégration de la carte

Chapitre 11

Afficher une carte avec le `google_maps_flutter` brancher

Le plugin GoogleMaps peut être d'afficher une carte dans Flutter, comme ceci:

La première étape nécessaire est d'ajouter la dépendance du plugin dans le fichier `pubspec.yaml` et d'installer avec les packages de flutter get commande:

```
dépendances:  
...  
google_maps_flutter: ^0.5.3
```

[338]

Épisode 357

Activation de l'API Maps sur Google Cloud Console

Avant d'utiliser le widget GoogleMap, nous devons obtenir une clé API Maps valide de Google Plateforme de cartes. Le processus est effectué dans la plate-forme Maps sur Google Cloud Console,

<https://.../cl>

Voyons comment fonctionne le processus:

1. Sélectionnez l' option **MISE EN ROUTE** . Nous sommes guidés à travers le processus de activer l'API. Tout d'abord, nous sélectionnons les API que nous voulons activer:

[339]

Épisode 358

2. Et puis, nous sélectionnons le projet pour lequel nous voulons activer l'API Maps:

3. Après cela, nous devons activer la facturation pour le projet. Google Maps Platform est gratuit à utiliser mais nécessite un compte de facturation pour être lié au projet. Après création / activation du compte de facturation pour le projet, nous activons l'API:

4. Et enfin, nous obtenons la clé API à utiliser dans notre application mobile:

[340]

Épisode 359

Vues de la plateforme et intégration de la carte

Chapitre 11

La clé API est accessible ultérieurement, sur l'explorateur de l'API sur Google Cloud Console.

Cette clé est utilisée pour initialiser le plugin de carte sur les deux plates-formes, de la même manière que nous l'a déjà fait pour AdMob et Firebase.

Intégration de l'API Google Maps sur Android

Pour la plate-forme Android, nous devons changer le

android / src / main / AndroidManifest.xml et ajoutez une balise de métadonnées contenant le Clé API que nous avons obtenue de la console Maps:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hands_on_maps">
    <application ...>
        <meta-data android: name = "com.google.android.geo.API_KEY"
            android: value = "VOTRE CLÉ ICI" />
    </application>
</manifest>
```

Intégration de l'API Google Maps sur iOS

Sous iOS, nous modifions le fichier ios / Runner / AppDelegate.swift en ajoutant le code responsable de la configuration de la clé API sur le plugin:

```
import UIKit
import Flutter
import GoogleMaps

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
    override func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        GeneratedPluginRegistrant.register(avec: self)
        GMSServices.provideAPIKey("VOTRE CLÉ ICI")
    }
}
```

```

    renvoie super.application (application, didFinishLaunchingWithOptions:
        Options de lancement)
    }
}

```

[341]

Épisode 360

*Vues de la plateforme et intégration de la carte**Chapitre 11*

N'oubliez pas que sur iOS, nous devons activer l'aperçu des vues intégrées version en ajoutant le paramètre spécifique sur le fichier Info.plist (voir la précédente section des *vues de la plateforme*).

Afficher une carte sur Flutter

Après avoir correctement initialisé le plugin sur l'une ou l'autre des plates-formes, nous pouvons utiliser le widget GoogleMap sur notre application. Dans une implémentation minimale, il suffit de l'ajouter à notre mise en page:

```

// fait partie du widget MapPage
@passer outre
Construction du widget (contexte BuildContext) {
    ...
    retourner GoogleMap (
        initialCameraPosition: CameraPosition (
            cible: LatLng (51.178883, -1.826215),
            zoom: 10,0
        ),
    );
    ...
}
...

```

La seule propriété obligatoire à définir dans le widget GoogleMap est initialCameraPosition, qui positionnera la visualisation de la carte dans un emplacement cible défini dans l'instance CameraPosition. La classe CameraPosition prend également en charge les propriétés de zoom, d'inclinaison et de relèvement.

[342]

Épisode 361

*Vues de la plateforme et intégration de la carte**Chapitre 11*

Avec cette configuration, nous pouvons voir GoogleMap en action:

Comme vous pouvez le voir à nouveau, le widget remplit tout l'espace disponible, un comportement défini par PlatformView. De plus, par défaut, les interactions cartographiques telles que le zoom et le déplacement sont activées. Nous pouvons les modifier avec le widget GoogleMap lié aux gestes précédemment vu Propriétés.

[343]

Épisode 362

Vues de la plateforme et intégration de la carte

Chapitre 11

Ajout de marqueurs à la carte

L'affichage d'une carte dans l'application n'est que le point de départ de la création d'une application basée sur une carte. L'ajout d'informations sur les lieux, par exemple, est l'une des tâches les plus courantes lorsque travailler avec des cartes. Voyons comment nous pouvons ajouter des marqueurs à la carte précédemment créée en utilisant la classe Marker fournie par le plugin.

La classe Marker

Le marqueur, comme mentionné dans la documentation, marque simplement un emplacement géographique sur la carte. Il ajoute des informations de contexte sur la carte, telles que l'identification d'un lieu, d'un point de contrôle ou point d'intérêt.

Les marqueurs sont généralement définis avec une icône et une ou plusieurs actions dans son événement de clic. Les propriétés suivantes sont parmi les plus utilisées lors de l'ajout de marqueurs à une carte:

position: Bien que non obligatoire par le plugin lui-même, il identifie le emplacement géographique du marqueur sur la carte, il est donc presque toujours nécessaire icône: il s'agit d'une icône de marqueur au format BitmapDescriptor

Consultez plus d'informations sur la classe BitmapDescriptor sur le page de documentation du plugin: <https://.. / pub da>

```
google\_ma
classe.htm
```

markerId: il s'agit d'un identifiant unique du marqueur sur la carte

infoWindow: il s'agit de la fenêtre d'informations de Google Maps qui s'affiche lorsque le marqueur est touché

Note de la documentation:

"Une icône de marqueur est dessinée orientée contre l'écran de l'appareil plutôt que contre la carte surface, c'est-à-dire qu'elle ne changera pas nécessairement d'orientation en raison des rotations, de l'inclinaison ou zoom."

[344]

Épisode 363

Vues de la plateforme et intégration de la carte

Chapitre 11

Ajout de marqueurs dans le widget GoogleMap

Comme nous l'avons vu précédemment, le widget GoogleMap expose la propriété markers, qui s'attend à ce qu'une collection Set d'instances de Marker lui soit transmise. Voyons comment ajouter markers en définissant la propriété markers:

1. Tout d'abord, nous ajoutons un champ _markers à la classe MapPage pour contenir un ensemble aléatoire de marqueurs (instances de marqueur):

```
La classe MapPage étend StatelessWidget {
  _markers finaux = {
    Marqueur(
      position: LatLng (51.178883, -1.826215),
      markerId: MarkerId ('1'),
      infoWindow: InfoWindow (titre: 'Stonehenge'),
      icône: BitmapDescriptor.defaultMarker
    ),
    Marqueur(
      position: LatLng (41.890209, 12.492231),
      markerId: MarkerId ('2'),
      infoWindow: InfoWindow (titre: 'Colisée'),
      icône: BitmapDescriptor.defaultMarker
    ),
    Marqueur(
      position: LatLng (36.106964, -112.112999),
      markerId: MarkerId ('3'),
      infoWindow: InfoWindow (titre: 'Grand Canyon'),
      icône: BitmapDescriptor.defaultMarker
    ),
  };
  ...
}
```

2. Et puis, il nous suffit de définir la propriété markers sur le widget GoogleMap:

```
@passer autre
Construction du widget (contexte BuildContext) {
  retourner GoogleMap (
    initialCameraPosition:
      CameraPosition (cible: LatLng (51.178883, -1.826215),
      zoom: 10,0),
    marqueurs: _markers,
  );
}
```

[345]

Épisode 364

*Vues de la plateforme et intégration de la carte**Chapitre 11*

Si nous tapons sur un marqueur, l'objet InfoWindow correspondant est affiché avec le jeu de *titres* :

Comme vous l'avez vu, ajouter des marqueurs au widget GoogleMap est aussi simple que d'afficher la carte lui-même, car il suit le paradigme Flutter de la reconstruction du widget avec la description fourni dans sa construction (c'est-à-dire des marqueurs).

Remarque pour les curieux: ces marqueurs font partie des 17 endroits visitez [ces Google Maps trouvés sur Facebook avec leurs coordonnées](#) [sur org/_art](#) [cartes.l...](#)

[346]

Épisode 365

*Vues de la plateforme et intégration de la carte**Chapitre 11*

Ajout d'interactions cartographiques

L'ajout de marqueurs à la carte permet d'enrichir les informations contextuelles qui y sont impliquées; cependant, c'est loin d'être suffisant pour une vraie application basée sur la carte. Gestion des événements ou modification la carte en fonction des besoins des utilisateurs est également fondamentale. Voyons comment nous pouvons ajouter des marqueurs dynamiquement sur la carte et utilisez la classe GoogleMapController pour interagir avec la carte caméra par programmation.

Ajouter des marqueurs dynamiquement

Comme dit précédemment, nous devons passer les marqueurs lors de la construction du widget GoogleMap, donc la première étape consiste à faire de notre widget MapPage un widget StatefulWidget et à reconstruire son sous-arbre chaque fois que nous voulons ajouter un nouveau marqueur.

Après cela, nous avons ajouté un bouton à la mise en page afin que nous puissions ajouter le marqueur après le construction initiale. Le bouton onPressed callback appelle _addMarkerOnCameraCenter, qui fonctionne comme suit:

```
void _addMarkerOnCameraCenter () {
    setState (() {
        _markers.add (Marqueur (
            markerId: MarkerId ("$ { _markers.length + 1}"),
            infoWindow: InfoWindow (titre: "Marqueur ajouté"),
            icône: BitmapDescriptor.defaultMarker,
            position: _cameraCenter,
        )));
    });
}
```

Comme vous pouvez le voir, il utilise la méthode setState pour provoquer une reconstruction du widget et ajoute Marqueur de l'ensemble _markers. La seule nouveauté ici est la position: _cameraCenter affectation sur Marker.

La valeur _cameraCenter est une propriété dans l'état qui suit l'emplacement central du caméra dans le widget GoogleMap. Il est récupéré à l'aide du rappel onCameraMove du widget, comme suit:

```
Google Map(
    ...
    onCameraMove: _cameraMove,
),
)
```

[347]

Épisode 366

Vues de la plateforme et intégration de la carte

Chapitre 11

Et la valeur est simplement stockée, comme mentionné précédemment:

```
void _cameraMove (position CameraPosition) {
    _cameraCenter = position.target;
}
```

De cette façon, chaque fois que l'utilisateur appuie sur le bouton, un marqueur est ajouté à la cible centrale emplacement sur la carte. Bien que ce ne soit pas un cas d'utilisation réel, il s'agit d'un point de départ pratique d'interagir avec la carte.

Jetez un œil à l'exemple hands_on_maps sur GitHub pour vérifier MapPage comme code de widget avec état et les petits changements de mise en page afficher un bouton.

GoogleMapController

Un autre niveau d'interaction que nous pouvons faire est fourni par la classe GoogleMapController, qui fonctionne de manière très similaire aux contrôleurs bien connus, tels comme TextEditingController.

La classe GoogleMapController vise à exposer les méthodes de contrôle de GoogleMap widget. À l'heure actuelle, les seules méthodes disponibles sont les suivantes:

```
animateCamera: Ceci démarre un changement animé de la position de la caméra de la carte
moveCamera: Cela change la position de la caméra de la carte sans animer
```

Obtenir GoogleMapController

Contrairement aux autres widgets contrôlables, nous ne fournissons pas de contrôleur au

Widget GoogleMap par nous-mêmes. Au lieu de cela, cela nous sera fourni par le rappel onMapCreated vu précédemment. Il suffit donc de le stocker, comme suit:

```
Google Map()
...
onMapCreated: (contrôleur) {
    _mapController = contrôleur;
},
),
```

_mapController est un champ d'instance du widget MapPage que nous utiliserons pour interagir avec la caméra de la carte.

[348]

Épisode 367

Vues de la plateforme et intégration de la carte

Chapitre 11

Animation d'une caméra cartographique vers un emplacement

Nous avons ajouté une rangée de boutons sur lesquels l'utilisateur peut appuyer pour se concentrer sur un endroit spécifique. Par en appuyant sur l'un de ces boutons, une nouvelle méthode sera appelée, _animateMapCameraTo, pour Stonehenge, par exemple:

```
RaisedButton (
    enfant: Texte ("Stonehenge"),
    onPressed: () {
        _animateMapCameraTo (_stonehengePosition);
    },
),
```

La nouvelle méthode est responsable de la demande de mise à jour de la caméra:

```
void _animateMapCameraTo (position LatLng) {
    _mapController.animateCamera (CameraUpdate.newLatLang (position));
}
```

Comme vous pouvez le voir, grâce à l'instance GoogleMapController récupérée auparavant, nous pouvons envoyer une animation de caméra à un nouvel emplacement sur la carte.

Le code des autres boutons est très similaire. Encore une fois, vérifiez hands_on_maps sur GitHub pour plus de détails sur l'intégration de la carte exemple.

Utilisation de l'API Google Places

Depuis le site officiel (<https://developers.google.com/places/>) nous pouvons voir ce qui suit:

«L'API Places est un service qui renvoie des informations sur les lieux à l'aide de requêtes HTTP. Les lieux sont définis dans cette API comme des établissements, des emplacements géographiques ou des points d'intérêts.»

Ce service peut être utilisé de plusieurs manières pour obtenir des informations sur les lieux:

Obtenir une liste de lieux en fonction de l'emplacement d'un utilisateur ou d'une chaîne de recherche
Obtenez des informations détaillées sur un lieu spécifique, y compris des avis d'utilisateurs

[349]

Épisode 368

Vues de la plateforme et intégration de la carte

Chapitre 11

Accès aux millions de photos liées au lieu stockées dans la base de données Google Place

Service de prédiction de requêtes pour les recherches géographiques textuelles, renvoyant une suggestion
requêtes au fur et à mesure que les utilisateurs saisissent et remplissent automatiquement le nom et / ou l'adresse d'un
placer en tant que type d'utilisateurs

Dans cette section, nous utiliserons l'API pour obtenir des informations détaillées (c'est-à-dire le nom) d'un
lieu ajouté par l'utilisateur via notre bouton de **marqueur de lieu** créé précédemment .

Activer l'API Google Places

À l'instar du SDK Google Maps, l'API Places doit être activée sur le développeur Google
Console, <https://.. / conso>

googleapis.com :

Vérifiez que vous êtes dans le bon projet et cliquez sur le bouton **ACTIVER**. Cela rendra le
Place l'API disponible via la même clé API utilisée auparavant.

[350]

Épisode 369

Vues de la plateforme et intégration de la carte

Chapitre 11

Premiers pas avec le plugin google_maps_webservice

Le plugin google_maps_webservice est un plugin communautaire Dart qui offre à un client le
API Google Places. Avec ce plugin, nous pouvons passer des appels vers le service Web de Google sans
la nécessité de créer les demandes par nous-mêmes.

Le plugin expose les appels en tant que méthodes de sa classe GoogleMapsPlaces. Cette classe offre
des méthodes telles que getDetailsByPlaceId, par exemple, qui appelle le point de terminaison des détails
du service Web et encapsule la réponse dans une classe PlacesDetailsResponse.

Consultez la page du plugin pour en savoir plus sur toutes les méthodes disponibles du
service Web: <https://.. / pub da>

[Webservice .](#)

Obtenir une adresse de lieu à l'aide de le plugin google_maps_webservice

Tout d'abord, nous devons ajouter le plugin en tant que dépendance dans le fichier pubspec.yaml de notre projet et obtenez-le avec les packages de flutter get command:

dépendances:
google_maps_webservice: ^ 0.0.12

Après cela, nous pouvons commencer à utiliser le plugin. La première chose à faire est de créer une instance de classe GoogleMapsPlaces afin que nous ayons accès aux méthodes fournies:

```
@passer outre
void initState () {
    super.initState ();

    _googleMapsPlaces = GoogleMapsPlaces (
        apiKey: 'API_KEY',
    );
}
```

Nous faisons cela dans la méthode initState afin de pouvoir l'utiliser juste après l'affichage de la carte à l'utilisateur. `_googleMapsPlaces` est un champ dans l'état du widget MapPage.

[351]

Épisode 370

Vues de la plateforme et intégration de la carte

Chapitre 11

Ensuite, nous définissons une méthode qui interrogera un nom de lieu en fonction d'une latitude / longitude paire:

```
Future <PlacesSearchResponse> _queryLatLngNearbyPlaces (position LatLng)
async {
    return wait _googleMapsPlaces.searchNearbyWithRadius (
        Localisation (position.latitude, position.longitude),
        1000,
    );
}
```

La méthode utilise la méthode `searchNearbyWithRadius` de la classe `GoogleMapsPlaces` qui interroge sur le service Web Google des lieux à proximité du lieu, classé par importance / importance, c'est-à-dire que les lieux les plus proches viennent en premier.

Pour utiliser la méthode créée, nous changeons notre fonction `_addMarkerOnCameraCenter` pour interroger l'adresse du lieu avant de l'ajouter à la carte:

```
void _addMarkerOnCameraCenter () async {
    lieux finaux = attendre _queryLatLngNearbyPlaces (_cameraCenter);
    final firstMatchName =
        places.results.length > 0? places.results.first.name: "";
    setState (()
        _markers.add (Marqueur (
            markerId: MarkerId ("$ {_markers.length + 1}"),
            infoWindow: InfoWindow (
                title: "Marqueur ajouté - $ firstMatchName"
            ),
            icône: BitmapDescriptor.defaultMarker,
            position: _cameraCenter,
        )));
}
```

Comme vous pouvez le voir, il y a quelques modifications par rapport à la version précédente. Voici les modifications:

La méthode est maintenant asynchrone, car le plugin renvoie un résultat Future et nous voulons l'attends

Nous obtenons la première correspondance de la requête (uniquement son adresse), le cas échéant

Nous ajoutons les informations de nom dans la propriété de titre InfoWindow

[352]

Épisode 371

Vues de la plateforme et intégration de la carte

Chapitre 11

Et, après avoir ajouté un marqueur à la carte, il contient maintenant le nom de l'emplacement:

Il existe de nombreuses autres façons d'intégrer l'API Google Places dans une application: c'était juste un simple. Avec cela, nous terminons l'intégration de la carte sur les applications Flutter. Continuez à suivre mises à jour du plugin car cette fonctionnalité évolue toujours avec le framework.

[353]

Épisode 372

Vues de la plateforme et intégration de la carte

Chapitre 11

Sommaire

Dans ce chapitre, nous avons vu les bases de l'utilisation des cartes dans Flutter avec le grand plugin `google_maps_flutter`. Nous avons vu qu'il s'appuie sur la fonctionnalité d'affichage de la plateforme cela nous permet d'afficher des vues natives dans le contexte Flutter. Nous avons vu comment nous pouvons créer ces vues par nous-mêmes en utilisant la structure du cadre.

Nous avons vu les propriétés disponibles du widget `GoogleMap` et comment le manipuler pour afficher des marqueurs dessus et déplacer la caméra à l'aide de la classe `GoogleMapController`.

Enfin, nous avons utilisé l'API Google Places pour obtenir des informations sur un emplacement et l'afficher sur le marqueur à l'aide d'une classe `InfoWindow`.

Dans le chapitre suivant, nous examinerons les outils disponibles de Flutter pour l'application avancée développement.

[354]

Épisode 373

4

Section 4: Flutter avancé - Ressources pour les applications complexes

Les applications complexes et uniques impliquent des fonctionnalités dont le développeur a besoin pour comprendre comment réaliser, comme l'écriture de code natif de la plateforme et la personnalisation des ressources du framework selon leurs besoins.

Les chapitres suivants sont inclus dans cette section:

[Chapitre 12, Test, débogage et déploiement](#)

[Chapitre 13](#), Amélioration de l'expérience utilisateur
[Chapitre 14](#), Manipulations graphiques des widgets
[Chapitre 15](#), Animations

Épisode 374

12

Test, débogage et Déploiement

Flutter fournit d'excellents outils pour aider le développeur à atteindre ses objectifs sur le plate-forme, de l'API de test aux outils et plugins IDE. Dans ce chapitre, vous apprendrez à ajouter tests pour créer une application sans bogue, déboguer pour trouver et résoudre des problèmes spécifiques, profiler votre application performances pour trouver les goulots d'étranglement et inspecter les widgets de l'interface utilisateur. En outre, vous apprendrez à préparer l'application pour le déploiement sur l'App Store et Google Play.

Les sujets suivants seront traités dans ce chapitre:

- Test des widgets Flutter
- Débogage des applications Flutter
- Profilage des performances des applications Flutter
- Inspection de l'arborescence des widgets Flutter
- Préparation de l'application pour le déploiement

Test de flottement - tests unitaires et widgets

Tester manuellement les applications mobiles est fondamental tant que nous devons ajouter des fonctionnalités à une application en continu. Il existe plusieurs façons de tester une application Flutter, chacune avec un certain niveau de avantages impliqués et ils ne diffèrent pas trop des tests d'autres applications logicielles.

Les tests unitaires et d'intégration bien connus sont possibles avec Flutter. De plus, nous pouvons écrire des tests de widget pour tester les widgets de manière isolée. Voyons comment nous pouvons écrire un widget et des tests d'intégration pour vous assurer que nos applications fonctionnent correctement.

Vous pouvez consulter le [chapitre 2](#), *Programmation intermédiaire des fléchettes, unité d'écriture*

section *tests* , car les tests unitaires Flutter ne sont rien de plus que des tests unitaires Dart.

Épisode 375

Test, débogage et déploiement

Chapitre 12

Tests de widgets

Les tests de widgets sont utilisés pour valider les widgets de manière isolée. Ils ressemblent beaucoup à l'unité tests mais concentrez-vous sur les widgets.

L'objectif principal est de vérifier les interactions des widgets et si les widgets se présentent comme prévu. Comme les widgets vivent dans l'arborescence des widgets à l'intérieur du contexte Flutter, les tests de widgets nécessitent environnement cadre à exécuter. C'est pourquoi Flutter fournit des outils d'écriture widget teste via le package flutter_test.

Le package flutter_test

Le package flutter_test est livré avec le SDK Flutter, est construit en plus du test package, et fournit un ensemble d'outils pour nous aider à écrire et exécuter des tests de widgets.

Comme indiqué précédemment, les tests de widget doivent être exécutés dans l'environnement de widget et Flutter aide avec cette tâche avec la classe WidgetTester. Cette classe résume la logique pour nous de construire et interagir avec le widget testé et l'environnement Flutter.

Nous n'avons pas besoin d'instancier cette classe par nous-mêmes car le framework fournit le fonction testWidgets (). La fonction testWidgets () est similaire au test Dart () fonction vue auparavant au [chapitre 2, Programmation intermédiaire de fléchettes](#) , section de *tests unitaires d'écriture* . La différence est le contexte Flutter, cette fonction configure une instance WidgetTester pour interagir avec l'environnement, comme mentionné précédemment.

La fonction testWidgets

Cette fonction est le point d'entrée de tout test de widget dans Flutter:

```
void testWidgets (Description de la chaîne, rappel WidgetTesterCallback, {bool skip: false, timeout timeout})
```

Nous allons le vérifier en action en quelques étapes. Commençons par vérifier sa signature:

```
description: cela aide à documenter le test; autrement dit, il décrit quel widget les fonctionnalités sont en cours de test.  
callback: il s'agit de WidgetTesterCallback. Ce rappel reçoit un WidgetTester afin que nous puissions interagir avec le widget et rendre notre validations. C'est le corps du test, où nous écrivons notre logique de test.  
skip: nous pouvons ignorer le test lors de l'exécution de plusieurs tests en définissant cet indicateur.  
timeout: il s'agit de la durée maximale pendant laquelle le rappel de test peut s'exécuter.
```

[357]

Épisode 376

Test, débogage et déploiement

Chapitre 12

Exemple de test de widget

Lorsque nous générions un projet Flutter, nous avons la dépendance du package flutter_test ajouté pour nous automatiquement et un exemple de test est généré dans le répertoire test /. Vérifions ça en dehors.

Tout d'abord, dans pubspec.yaml, il y a la dépendance de package flutter_test ajoutée:

```
dev_dependencies:
  flutter_test:
    sdk: scintillement
```

Notez que la version du package n'est pas spécifiée. En outre, l'origine est configuré en tant que SDK Flutter.

Ensuite, nous pouvons vérifier le test de widget de base dans le fichier test / widget_test.dart:

```
void main () {
  testWidgets ('Counter incrémenté le test de fumée', (WidgetTester tester) async
  {
    attendre tester.pumpWidget (MyApp ());
    expect (find.text ('0'), findOneWidget);
    expect (find.text ('1'), findNothing);

    attendre tester.tap (find.byIcon (Icons.add));
    attendre tester.pump ();

    expect (find.text ('0'), findNothing);
    expect (find.text ('1'), findOneWidget);
  });
}
```

Cet exemple de test de widget valide le comportement de la célèbre application de compteur Flutter. Le test va comme suit:

Le test est défini avec une description et le vu la propriété WidgetTesterCallback. Notez également que le rappel à la fonction async modificateur, comme les méthodes WidgetTester car il renvoie un type Future.
Tout commence par un widget; attendre tester.pumpWidget (MyApp()); rend l'interface utilisateur du widget donné - MyApp, dans ce cas.
Si nous avons besoin de reconstruire le widget à un moment donné, nous pouvons utiliser le tester.pump () méthode.

[358]

Épisode 377

Test, débogage et déploiement

Chapitre 12

Dans les tests de widgets, deux éléments supplémentaires sont importants et très courants, trouver et attendre():

La classe Finder est ce qui nous permet de rechercher des widgets spécifiques dans l'arbre. La constante de recherche nous fournit des outils (Finders) pour rechercher et regarder dans l'arborescence des widgets des widgets spécifiques.

Vérifiez tous les Finders disponibles fournis par find: <https://api.flutter.dev/flutter/widgets/finders.html>

La méthode expect () est utilisée avec Matchers pour faire des affirmations sur les widgets trouvés à l'aide des Finders.
Matcher aide à valider la caractéristique de widget trouvée avec un valeur attendue.

Analysons les assertions de test de widget précédentes:

1. Au début, il y a une assertion pour la présence d'un seul widget avec le

0 texte et aucun avec 1:

```
expect (find.text ('0'), findOneWidget);
expect (find.text ('1'), findNothing);
```

2. Ensuite, tap () est exécuté, suivi d'une requête pump (). Le robinet se produit sur un

widget qui contient l'icône Icons.add:

```
attendre tester.tap (find.byIcon (Icons.add));
attendre tester.pump ()
```

3. La dernière étape consiste à vérifier que le texte correct est à nouveau affiché. Mais cette fois, la constante findOneWidget est utilisée pour vérifier que seul le texte, 1, est visible:

```
expect (find.text ('0'), findNothing);
expect (find.text ('1'), findOneWidget);
```

[359]

Épisode 378

Test, débogage et déploiement

Chapitre 12

Comme la constante find, plusieurs Matchers sont disponibles; ne trouve rien et findOneWidget ne sont que quelques-uns d'entre eux.

Vérifiez tous les Matchers disponibles dans la bibliothèque flutter test
Documentation: <https://api.flutter.dev/flutter/test/test.Binder-class.html>

[flutter_bibliot](#)

Débogage des applications Flutter

Le débogage est un élément important du développement logiciel. Petites erreurs, étranges comportements et les bogues complexes peuvent être résolus à l'aide du débogage. Avec ça, on peut procéder comme suit:

- Faire des affirmations logiques
- Déterminer les améliorations nécessaires
- Trouver des fuites de mémoire
- Faire une analyse de flux

Flutter fournit également plusieurs outils pour vous aider dans cette tâche. Comme nous l'avons vu précédemment en [chapitre 1](#), *Une introduction à Dart*, Dart contient un ensemble d'outils pour aider à la travail de développeur.

Nous n'évaluons pas un IDE spécifique pour le développement de Flutter et vous pouvez deviner que le débogage n'est pas possible sans lui. Cependant, l'outil Dart est également préparé pour cela.

Observatoire

Le débogage Flutter est basé sur l'outil **Dart Observatory**. L'Observatoire de Dart est présent dans le Dart SDK et aide au profilage et au débogage des applications Dart telles que les applications Flutter.

[360]

Épisode 379

Test, débogage et déploiement

Chapitre 12

Lorsqu'une application Flutter est démarrée en mode débogage (rappelez-vous la compilation JIT du [chapitre 1](#), *An Introduction to Dart*), cet outil est automatiquement exécuté, permettant le débogage et le profilage sur l'application. En utilisant la commande flutter run, vous aurez l'adresse: port partie de la sortie après le message Hot Reload. Cette adresse est l'adresse de l'**interface utilisateur de l'Observatoire** ; nous peut y accéder via de nombreux navigateurs Web * et voici à quoi il ressemble:

Il existe certains navigateurs Web avec des limitations sur l'affichage du Outil d'observation. Veuillez vérifier le problème existant concernant ceci: <https://github.com/flutter/flutter/issues/361>

Il imprime différentes informations sur l'application en cours d'exécution, telles que la version Flutter, utilisée mémoire, hiérarchie de classes et journaux. En outre, un outil supplémentaire important peut être utilisé, le outil de débogage:

[361]

Épisode 380

Test, débogage et déploiement

Chapitre 12

Dans cette page, comme vous pouvez le voir, nous avons accès à toutes les fonctionnalités de débogage, telles que le suivant:

- Ajout et suppression de points d'arrêt
- Exécuter étape par étape, ligne par ligne
- Commutez et gérez les isolats

Vérifiez toutes les fonctionnalités de l'interface utilisateur de l'Observatoire disponibles et un didacticiel d'utilisation complet à <https://.../dart>

Lorsque vous utilisez un IDE comme Visual Studio Code ou Android Studio / IntelliJ, vous ne serez pas en utilisant directement des outils tels que l'interface utilisateur de l'Observatoire. Les IDE utilisent Dart Observatory sous le capot pour exposer ses fonctionnalités via l'interface IDE.

[362]

Épisode 381

Test, débogage et déploiement

Chapitre 12

Fonctionnalités de débogage supplémentaires

Dart fournit des fonctionnalités supplémentaires pour faciliter le débogage avancé avec des variantes du outils communs qui peuvent rendre le processus de débogage encore plus utile. Ce sont les suivants:

L'instruction debugger (): également appelée points d'arrêt programmatiques, c'est là que nous pouvons ajouter un point d'arrêt uniquement si une condition attendue est vraie:

```
void login (string username, string password) {
    débogueur (quand: mot de passe == null);
    ...
}
```

Dans cet exemple, un point d'arrêt se produira uniquement si la condition dans le Le paramètre est vrai, c'est-à-dire uniquement si l'argument du mot de passe est nul. Disons que c'est une valeur inattendue: suspendre l'exécution à ce stade peut aider à comprendre pourquoi se produit et comment y réagir. Ceci est très utile pour tracer des états inattendus et la logique échoue.

debugPrint () et print (): print () est une méthode pour enregistrer des informations dans le console de journal de flutter. Lorsque nous utilisons la commande flutter run, sa sortie de journal est redirigé vers la console et nous pouvons voir tout ce qui vient de print () et les appels à debugPrint (). La seule différence entre ces appels est que la version debugPrint () évite la suppression des journaux par le noyau Android (Flutter logs ne sont qu'un wrapper pour adb logcat).

Vous pouvez en savoir plus sur les journaux Flutter sur <https://.../f/test/> débogage

`assert; assert()` est utilisé pour interrompre l'exécution de l'application lorsque certaines conditions ne sont pas satisfait. C'est similaire à la méthode `debugger()`, mais au lieu de mettre en pause l'exécution, il interrompt l'exécution en lançant `AssertionError`.

DevTools

Dart DevTools est défini dans la documentation comme suit:

"Une suite d'outils de performance pour Dart et Flutter."

[363]

Épisode 382

Test, débogage et déploiement

Chapitre 12

Il s'agit de la prochaine version des outils de l'Observatoire. Les IDE sont déjà intégrant cette suite dans leurs internes, et c'est similaire à Observatory, comme vous pouvez le voir:

Comme vous pouvez le voir, il dispose de quelques outils qui peuvent aider à l'analyse des performances de Flutter applications, tout comme Observatoire. Vous pouvez l'activer / l'installer en exécutant la commande suivante commande dans un terminal:

`pub global activer devtools`

Ou, vous pouvez exécuter ce qui suit:

`paquets flutter pub global activer devtools`

Après cela, nous pouvons exécuter l'outil avec cette commande:

`pub global run devtools`

Ou, nous pouvons utiliser ce qui suit:

`paquets flutter pub global exécuter devtools`

[364]

Épisode 383

Accédez à la page affichée dans un navigateur Web, et vous aurez quelque chose de similaire au capture d'écran suivante:

Comme vous pouvez le voir, nous devons fournir le port de l'application en cours d'exécution (le port Observatory, comme avant) pour que DevTool puisse inspecter les mesures de l'application.

Consultez la page de documentation de DevTools pour plus de détails sur les étapes d'installation pour différents systèmes d'exploitation et IDE: <https://github.com/flutter/devtools/>

Notez également qu'au moment de la rédaction de ce livre, la suite DevTools est toujours dans l'aperçu de la version et peut changer au moment où vous lisez ceci.

Profilage des applications Flutter

Flutter vise à fournir des applications hautes performances avec une fréquence d'images et une fluidité élevées. Comme débogage qui peut aider à trouver des bogues et plus encore, le profilage est un autre outil utile qui peut aider les développeurs à trouver les goulots d'étranglement dans une application, à éviter les fuites de mémoire ou à améliorer les performances de l'application.

L'outil Observatory est, encore une fois, le pont qui nous permet d'inspecter les performances de l'application Flutter. Comme le débogueur, cette section de l'outil est également intégrée dans les IDE lorsque nous les utilisons.

Le profileur de l'Observatoire

Comme vu précédemment, Observatory expose plusieurs outils au développeur pour mesurer l'application performances et éviter tout problème éventuel qui y serait lié. Ceci est fait avec le exposition de plusieurs métriques, comme vous pouvez le voir:

[365]

Épisode 384

La mémoire, l'utilisation du processeur et d'autres informations sont disponibles via le moniteur afin que nous peut évaluer différents aspects de l'application.

Mode profil

Lorsque nous exécutons notre application Flutter en mode de *débogage* par défaut avec le flutter run commande, nous ne pouvons pas nous attendre aux mêmes performances que le mode *release*. Comme nous déjà sachez que Flutter s'exécute en mode débogage à l'aide du compilateur JIT Dart lorsque l'application s'exécute, contrairement à les modes de version et de profil, où le code de l'application est pré-compilé à l'aide de AOT Dart compilateur.

Pour évaluer les performances, nous devons nous assurer que l'application s'exécute à sa capacité maximale; c'est pourquoi Flutter propose différentes méthodes d'exécution: debug, profil et version.

En mode profil, l'application est compilée d'une manière très similaire au mode release, et ce est clairement compréhensible, car nous devons savoir comment l'application fonctionnera dans des scénarios réels. La seule surcharge ajoutée à l'application est celle requise pour activer le profilage (c'est-à-dire que l'Observatoire peut se connecter au processus de candidature).

Un autre aspect important du profilage est la nécessité d'un appareil physique. Simulateurs et les émulateurs ne reflètent pas les performances réelles des appareils réels. Comme le matériel est différent, les métriques de l'application peuvent être influencées et l'analyse peut être correcte.

[366]

Épisode 385

Test, débogage et déploiement

Chapitre 12

Pour exécuter une application en mode profil, nous devons ajouter l'indicateur --profile à la commande run (rappelez-vous, il n'est disponible que sur de vrais appareils):

flutter run - profile

En cours d'exécution dans ce mode, nous avons toutes les informations nécessaires pour inspecter l'application performances en général. Un autre outil utile que le mode profil permet est la **performance superposition**.

Les IDE proposent également le mode profil via leurs interfaces particulières, donc lorsque vous voyez ce mode dans l'IDE choisi, vous savez ce que cela signifie.

Superposition de performances

La superposition de performances est un retour visuel affiché dans l'application. Il fournit plusieurs statistiques de performance utiles. Plus précisément, il affiche des informations sur le temps de rendu. Ici est un exemple de superposition de performances affiché:

Superposition de performances (il s'agit d'une image de superposition de performances. Les autres informations (superposées) ne sont pas importantes ici.)

[367]

Épisode 386

Test, débogage et déploiement

Chapitre 12

Deux graphiques sont affichés représentant le temps de rendu des images prises par les deux threads, interface utilisateur et GPU. Le cadre actuel est affiché dans une barre verte verticale. Aditionellement, nous pouvons voir les 300 dernières images et avoir une idée des étapes critiques du rendu.

Flutter utilise plusieurs threads pour faire son travail. L'interface utilisateur et le GPU contiennent le travail d'affichage du framework, et c'est pourquoi les deux sont affichés dans la superposition de performances. Le fil de l'interface utilisateur est où notre code Dart est exécuté et la construction de la logique et de la description du widget se produit et où le cadre crée une arborescence de couches pour que le thread GPU fonctionne, où les graphiques prennent vie et où s'exécute la bibliothèque graphique Skia.

De plus, à ces threads, Flutter contient également le thread Platform, où le plugin le code s'exécute et le thread d'E / S, où des tâches d'E / S coûteuses sont exécutées. Les deux threads ne apparaissent sur la superposition de la plate-forme.

Vous pouvez vérifier certaines des améliorations possibles que les performances la superposition peut aider à <https://flutter.dev/perf>

Inspection de l'arborescence des widgets Flutter

Avec le débogage et le profilage, nous pouvons découvrir et résoudre de nombreux problèmes et performances problèmes avant qu'ils ne surviennent en production. De plus, nous pouvons mesurer le coût d'une application de exécution au fur et à mesure du développement.

Les deux outils nous offrent des mesures et, avec cela, nous pouvons inspecter des éléments de code soigneusement, mais qu'en est-il de la mise en page? Nous pouvons, à coup sûr, mesurer le cadre de performance par frame basé sur le temps de rendu de notre arbre de widgets, comme nous l'avons vu auparavant avec l'aide de superposition de performances. Mais que diriez-vous de vérifier si notre arbre prend plus d'espace que nécessaire - c'est-à-dire qu'il a plus de widgets que nécessaire - ou si un widget est en cours de création à le bon moment / niveau.

L'inspecteur Flutter peut vous aider dans cette tâche. Encore une fois, avec les grands DevTools, nous pouvons accéder cette fonctionnalité.

[368]

Épisode 387

Test, débogage et déploiement

Chapitre 12

Inspecteur de widgets

L'inspecteur de widgets est un autre de la grande suite d'outils qui peut aider le développeur avec tâches d'optimisation. Cet outil fournit une visualisation détaillée de l'arborescence des widgets.

L'inspecteur Flutter dans DevTools

Sur les IDE pris en charge, le plugin offre déjà des moyens d'accéder à l'inspecteur de widgets, à l'aide de Outil d'inspecteur de widget Flutter sous le capot. Il est également accessible dans la suite DevTools:

Comme vous pouvez le voir, l'arborescence des widgets est présentée et nous pouvons accéder à tous les détails sur les widgets. Pour développeurs Web, cela ressemblera beaucoup à l'explorateur d'éléments dans les outils de développement Web, comme celui de Chrome, par exemple.

Tout comme les outils de profilage et de débogage, l'exploration détaillée de l'arborescence des widgets peut être extrêmement utile pour découvrir les problèmes de mise en page qui seraient difficiles sans visualisation de l'arborescence.

De plus, en regardant la capture d'écran précédente, nous avons eu un petit indice pour activer le *widget de suivi de création*. Lorsque nous ignorons cet indicateur, l'outil affichera un arbre plus profond que ce à quoi on pourrait s'attendre; ce pourquoi il expose des widgets intermédiaires en plus de ceux que nous définissons dans notre application.

Lorsque nous l'activerons, l'arbre aura l'air beaucoup plus simple:

[369]

Épisode 388

Test, débogage et déploiement

Chapitre 12

Avec cela, nous avons un arbre qui ressemble beaucoup plus à celui défini dans notre code, ce qui le rend plus facile à suivre les problèmes. En outre, nous avons des détails sur les propriétés des widgets qui aident également à trouver de petits

problèmes de mise en page.

Préparation des applications pour le déploiement

Flutter vise à offrir les meilleures ressources possibles au développeur pour travailler, et ainsi de suite telles que différentes versions pour le développement, le profilage et la publication ont du sens.

Lors de la préparation d'une application pour la publication, des éléments tels que la compilation à la volée fournie par Dart JIT n'a pas de sens; au lieu de cela, la meilleure chose est d'avoir un plus petit, optimisé et application performante fournie par le compilateur Dart AOT.

La publication d'une application sur Google Play Store et App Store nécessite des comptes d'éditeur valides. Donc, reportez-vous à la documentation des deux plateformes pour savoir comment publier dans les magasins après créer une version commerciale de votre application.

Google a des frais d'inscription uniques de 25 \$ que vous devez payer avant de pouvoir télécharger une app. Vous pouvez vous connecter à à l'[adr](#)

App Store a des frais d'adhésion de 99 \$ par an. Vous trouverez tous les détails et connectez - vous à l'[adresse https://développeur.Appl](#)

[370]

Épisode 389

Test, débogage et déploiement

Chapitre 12

Mode de libération

En mode version, les informations de débogage sont supprimées de l'application et la compilation est réalisé avec la performance à l'esprit. Rappelez-vous, en mode de libération, comme le profil, le L'application ne peut être exécutée que sur des appareils physiques, pour les mêmes raisons également.

Pour compiler en mode release, il suffit d'ajouter l'indicateur --release au flutter run commande et avoir un appareil physique connecté, et c'est tout. Bien que nous puissions le faire, nous n'utilisiez généralement pas la commande flutter run avec l'indicateur --release. Au lieu de cela, nous utilisons cet indicateur avec la commande flutter build pour avoir un fichier d'application intégré dans la cible Formats Android / iOS pour la distribution.

Publier des applications pour Android

Sous Android, .apk est le format qui devrait être publié sur le Google Play Store. quand nous exécutez les commandes flutter build apk ou flutter build appbundle, nous générions le fichier prêt pour le déploiement.

Le format du bundle d'applications Android est également partiellement pris en charge au moment de écrire ce livre.

Avant de générer le fichier pour le déploiement et la publication dans n'importe quel magasin, nous devons faire assurez-vous que toutes les informations sont correctes (c'est-à-dire le nom et le package), tous les actifs nécessaires sont fourni et effectuer tous les ajustements spécifiques à la plate-forme.

Commençons par préparer notre application Favors pour sa sortie sur Google Play afin que nous puissions tout examiner des dernières étapes de publication d'une application Flutter.

AndroidManifest et build.gradle

Sous Android, les méta-information sur l'application sont fournies dans les fichiers AndroidManifest.xml et build.gradle, nous devons donc examiner et faire quelques ajustements dans les deux.

[371]

Épisode 390

*Test, débogage et déploiement**Chapitre 12*

Pensez également à configurer correctement le projet dans la console Firebase et à ajouter le fichier google-services.json au projet (vous pouvez utiliser le même que celui généré pour [Chapitre 8, Plugins Firebase](#)).

AndroidManifest - autorisations

Une étape importante que nous devons faire est d'examiner les autorisations demandées dans le Fichier AndroidManifest.xml. Demander uniquement les autorisations nécessaires est une bonne et pratique recommandée, car votre application peut être analysée et votre publication peut être révoqué si vous demandez plus que les autorisations vraiment requises.

Dans notre application Favors, voici à quoi ressemblent les autorisations du manifeste:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hanson">

    <uses-permission android: name = "android.permission.INTERNET" />
    <uses-permission android: name = "android.permission.READ_CONTACTS" />
    <uses-permission android: name = "android.permission.WRITE_CONTACTS" />
    <uses-permission android: name = "android.permission.CAMERA" />
    <utilise-fonctionnalité
        android: name = "android.hardware.camera"
        android: requis = "faux" />
    ...
</manifest>
```

Outre les autorisations, il existe également la balise uses-feature (voir [Chapitre 10, Accès au périphérique Fonctionnalités de l'application Flutter](#)), ce qui peut limiter l'installation sur les appareils dotés d'une fonctionnalité spécifique disponible (ce n'est pas notre cas), il est donc important de le revoir également.

L'autorisation android.permission.INTERNET est utilisée par le framework Flutter avec l'outil Observatoire, donc, si votre application fonctionne hors ligne, vous pouvez le supprimer pendant release builds (ce n'est pas notre cas, car nous utilisons les technologies Firebase).

AndroidManifest - balises métas

Une autre étape très importante consiste à examiner les balises métas ajoutées à l'application pour travailler avec des services tels qu'AdMob ou Google Maps. Dans notre application Favors, AdMob était la seule clé ajouté, afin que nous puissions examiner la valeur pour nous assurer que le service fonctionnera avec la bonne clé ainsi que:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hanson">
    ...
<application>
```

[372]

Épisode 391

*Test, débogage et déploiement**Chapitre 12*

```
...
<méta-données
    android: name = "com.google.android.gms.ads.APPLICATION_ID"
    android: value = "ADMOB-KEY" />
</application>
</manifest>
```

N'oubliez pas que dans AdMob, nous pouvons utiliser des clés de test pendant le développement afin que nos tests ne soient pas

évalué comme une mauvaise utilisation de l'API.

AndroidManifest - nom et icône de l'application

Jusqu'à présent, dans nos tests, lorsque nous lançons l'application, vous pouvez voir que l'icône de l'application est un Logo Flutter. Pour la sortie, nous devons l'échanger avec notre superbe icône unique pour nous assurer nos utilisateurs distinguent notre application parmi des millions.

L'icône et le nom sont définis dans la balise d'application du manifeste. Par défaut, l'icône fait référence à l'icône Flutter par défaut, comme vous pouvez le voir:

```
<manifest ...>
  ...
  <application
    android: name = "io.flutter.app.FlutterApplication"
    android: label = "Hands On: Favors app"
    android: icon = "@ mipmap / ic_launcher" >
  ...
</manifest>
```

Nous apportons donc deux modifications à cette balise:

Nous remplaçons la valeur de l'étiquette par le nom final de notre application, le nom par lequel notre les utilisateurs reconnaîtront notre application.

La valeur de l'icône que nous pouvons utiliser pour changer l'icône de l'application (en remplacement de la valeur par défaut Flutter logo):

Sous Android, les ressources d'image telles que l'icône se trouvent dans le répertoire android / app / src / main / res /. Sous ceci répertoire, il existe de nombreux dossiers avec des variantes d'une ressource, pour des régions spécifiques, des tailles d'écran, des versions de système, etc.

L'icône de l'application Favors a été générée dans Android Asset Studio outil. Cela nous aide à suivre les directives Android et à générer plusieurs icônes variétés: <https://romana.org/html/>.

[html](https://romana.org/html/).

[373]

Épisode 392

Test, débogage et déploiement

Chapitre 12

Nous devons remplacer le fichier ic_launcher.png dans chacun des dossiers mipmap-xxxxdpi pour effectuer un remplacement complet de l'application icône.

Vérifiez les directives de Material Design sur les icônes pour vous assurer de créer une icône impressionnante pour votre application: <https://material.io/iconography/>.

Après avoir changé le nom et remplacé l'icône, nous pouvons examiner le fichier build.gradle pour faire les derniers ajustements pour le déploiement.

build.gradle - ID et versions de l'application

La valeur de l'ID d'application est ce qui rend une application unique dans le Play Store et sur Android système. Une bonne pratique consiste à utiliser le domaine de l'organisation comme package et à avoir l'application nom qui le suit. Dans notre cas, nous utilisons com.example.handson comme ID d'application.

Assurez-vous de vérifier cette valeur, car elle ne peut pas être modifiée une fois que vous avez téléchargé l'application sur le boutique.

Vous pouvez trouver ce code dans le fichier android / app / build.gradle, à l'intérieur du Section defaultConfig:

```
defaultConfig {
  applicationId "com.example.handson"
  minSdkVersion 16
  targetSdkVersion 28
```

```

multiDexEnabled true
versionCode flutterVersionCode.toInt()
versionName flutterVersionName
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

```

Comme vous pouvez le voir, nous pouvons modifier plus de paramètres que simplement changer applicationId. Dans Flutter, les versions du SDK sont généralement modifiées dans deux cas:

Si les exigences du cadre changent

Si nous utilisons une bibliothèque qui nécessite une version SDK minimale supérieure

Nous pouvons, à coup sûr, le changer en notre propre valeur requise si nous le voulons, mais assurez-vous de suivre les exigences du cadre.

[374]

Épisode 393

Test, débogage et déploiement

Chapitre 12

build.gradle - signature de l'application

L'étape de signature est l'étape finale mais la plus importante avant de publier une application au public, même si vous ne souhaitez pas publier sur le Google Play Store. C'est la signature qui confirme la propriété de l'application - brièvement, celui qui a la signature est propriétaire de l'application. Tu en ont besoin pour pouvoir publier des mises à jour sur votre application, par exemple.

Commencez par jeter un œil à la section buildTypes du fichier build.gradle:

```

buildTypes {
    Libération {
        signatureConfig signatureConfigs.debug
    }
}

```

Il contient la propriété signatureConfig, pointant vers une configuration de signature par défaut. nous besoin de changer cela en notre propre configuration de signature pour les raisons mentionnées précédemment. nous faites ceci en effectuant ces étapes:

1. Nous générerons notre fichier de keystore de développeur (vous pouvez utiliser le même keystore pour plusieurs applications). Cela se fait avec la commande suivante:

```
keytool -genkey -v -keystore DESTINATION_FILEPATH -keyalg RSA -
keysize 2048 -validité 10000 -alias clé
```

Suivez les invites et cela générera un keystore dans le chemin DESTINATION_FILEPATH, par exemple, <vos utilisateurs dir / my-release-key.keystore>. Vous devez référencer ce fichier dans le build.gradle maintenant.

2. Créez un fichier android / key.properties avec le contenu suivant:

```
storePassword = <mot de passe utilisé pour générer la clé>
keyPassword = <mot de passe utilisé pour générer la clé>
keyAlias = clé
storeFile = chemin du fichier de stockage de clés (c'est-à-dire </ votre répertoire utilisateur / ma-version-key.keystore>)
```

3. Ensuite, dans build.gradle, nous chargeons ce nouveau fichier key.properties et créons un nouvelle classe SigningConfig pour cela:

```

def keystoreProperties = nouvelles propriétés ()
def keystorePropertiesFile = rootProject.file ('key.properties')
if (keystorePropertiesFile.exists ()) {
    keystoreProperties.load (nouveau
    FileInputStream (keystorePropertiesFile))
}

```

[375]

Épisode 394

Test, débogage et déploiement

Chapitre 12

```

        }

    Android{
        ...

        signatureConfigs {
            Libération {
                keyAlias keystoreProperties ['keyAlias']
                keyPassword keystoreProperties ['keyPassword']
                fichier storeFile (keystoreProperties ['storeFile'])
                storePassword keystoreProperties ['storePassword']
            }
        }
    }
}

```

4. Ajoutez simplement l'extrait avant la section android, puis déclarez la signature configuration dans la sous-section signatureConfigs. Enfin, remplacez le SignatureConfig propriété dans l'option release dans les buildTypes précédents section avec le nouveau:

```

Android {
    ...
    buildTypes {
        Libération {
            signatureConfig signatureConfigs.release
        }
    }
}

```

Maintenant, lorsque nous utilisons les commandes flutter build apk ou flutter run --release, le l'application sera signée avec notre propre clé.

Après avoir effectué ces ajustements, nous sommes prêts à créer et à distribuer notre application. Juste une finale étape: vérifiez les valeurs versionCode et versionName de l'application; ils se remplissent automatiquement à partir du fichier pubspec.yaml. Ainsi, l'examen de ce fichier peut également être important.

Après avoir construit un .apk avec la commande flutter build apk, nous pouvons l'installer sur un périphérique physique connecté avec la commande flutter install. En outre, le fichier à être publié dans le Play Store est disponible: build / app / output / apk / app.apk.

Vous pouvez également travailler sur la minification et l'obfuscation du code pour améliorer la taille de l'application et la protection contre l'ingénierie inverse: https://github.com/flutter/flutter/tree/master/packages/flutter_tools/doc/minification.md

[com / fl](https://github.com/flutter/flutter/tree/master/packages/flutter_tools/doc/minification.md)

[376]

Épisode 395

Test, débogage et déploiement

Chapitre 12

Publier des applications pour iOS

La sortie sous iOS a une complexité plus élevée par rapport à Android. Bien que vous peut tester sur votre propre appareil lors du développement, rendre une application publique vous oblige à avoir un compte de développeur Apple valide avec la possibilité de publier sur l'App Store, car c'est le seul canal de publication d'applications pris en charge.

Comme Android, nous devons d'abord examiner certaines informations sur l'application dans le projet Xcode paramètres, comme nous l'avons fait dans AndroidManifest.xml. Et après cela, nous pourrons créer un archive d'application prête à être publiée sur l'App Store.

Vérifiez également la présence du fichier GoogleService-Info.plist dans ios / Runner répertoire (voir [chapitre 8, Firebase Plugins](#), pour vous rappeler comment l'importer dans Xcode).

Connecter l'App Store

Sous Android, nous n'avons pas besoin de configurer quoi que ce soit dans la console Play Store avant d'avoir l'apk prêt pour la publication. Une fois que nous l'avons, nous pouvons créer un registre sur la Play Console; remplir la description, les détails et les paramètres marketing; ensuite, nous téléchargeons notre fichier apk et publier.

N'oubliez pas que vous devez être inscrit à un programme de développement pour pouvoir publier sur l'App Store. (Cela s'applique également à l'enregistrement d'une application sur l'application Store Connect). Consultez également le guide officiel pour en savoir plus information: https://./help/por_dev2cd126805.

Sous iOS, le processus est différent. Le téléchargement et la publication sont gérés dans Xcode, donc pour télécharger l'application, nous créons d'abord un enregistrement sur l'App Store Connect, remplissons les descriptions et puis, sur Xcode, nous construisons et téléchargeons notre application iOS. Pour enregistrer l'application, procédez comme suit:

1. Chaque application iOS est associée à un Bundle ID, un identifiant unique enregistré auprès d'Apple. Tout d'abord, nous créons un enregistrement dans les ID d'application (https://idmsa.apple.com/IDMSWebAuth/signin?apldkKey=_891bd3417a7776362562d2197f89480a8547b108fd934911bcbea0110d07f757&path=%2Faccount%2Fresources%2F&rv=1), en remplissant l' **ID du bundle** , qui est l'iOS équivalent à applicationId d'Android.
2. Ensuite, nous créons une application dans le portail App Store Connect en sélectionnant l' **ID du bundle** nous nous sommes enregistrés à l'étape précédente. (Pour notre application Favors, c'est presque la même chose value comme applicationId d'Android).

Après avoir réalisé ces étapes dans App Store Connect, nous terminons le processus dans Xcode.

[377]

Épisode 396

Test, débogage et déploiement

Chapitre 12

Xcode

Dans Xcode, nous devons apporter quelques modifications pour rendre l'application prête au déploiement. Nous devons le faire modifiez l'icône de l'application, le nom public et l'ID du bundle. C'est très similaire à ce que nous fait dans Android.

Xcode - détails de l'application et ID du bundle

Dans l' onglet **Général** du projet **Runner** , nous pouvons modifier le **nom d'affichage de l'application** , qui est le nom de notre application. Nous avons également défini le nom **Android Hands On: Favors app** et définissez l'ID du bundle sur **com.biessek.hanson.favorsapp**.

Notez également les valeurs Version et Build; ils sont similaires au *nom de la version* et au *code de version* dans Android respectivement. Pour chaque téléchargement sur l'App Store, que ce soit Store ou TestFlight, nous devons augmentez la valeur de la version dans le fichier **pubspec.yaml**.

Dans la **cible de déploiement** , nous pouvons définir la version iOS minimale requise, 8.0 par défaut: la version minimale prise en charge par Flutter.

Xcode - AdMob

Contrairement à la configuration du fichier **AndroidManifest.xml**, nous n'avons pas besoin de mettre à jour notre ID AdMob dans iOS. Dans ce cas, la valeur de l'ID est extraite de celle transmise au Initialisation du SDK **FirebaseAdMob** dans Dart lui-même:

```
FirebaseAdMob.instance.initialize (
  appId: 'VOTRE_ADMOB_APP_ID'
);
```

Xcode - signature de l'application

Comme sur Android, nous avons besoin d'un moyen de revendiquer la propriété de l'application. Xcode gère ça pour nous; nous n'avons pas besoin de toucher directement un fichier. Lorsque nous nous enregistrons en tant que développeur Apple

et inscrivez-vous au programme Apple Developer Program, nous avons tout cela prêt.

Après ces paramètres, nous pouvons créer une version iOS de l'application comme nous l'avons fait pour Android, avec la commande flutter build ios. Ensuite, nous avons besoin d'une dernière étape dans Xcode pour publier notre application:

1. Dans Xcode, sélectionnez **Produit | Archiver** pour produire une archive de construction.
2. Ensuite, sélectionnez l'archive de build que vous venez de produire avec le flutter build ios commander.

[378]

Épisode 397

Test, débogage et déploiement

Chapitre 12

3. Cliquez sur le bouton **Valider...**. Si des problèmes sont signalés, résolvez-les et produire une autre version.
4. Une fois l'archive validée, cliquez sur **Télécharger sur l'App Store....**.

Après cela, nous avons une application iOS prête à être publiée. Nous pouvons soit le publier sur TestFlight (une application de test privée avec des utilisateurs de confiance) ou dans l'App Store.

Lisez la documentation officielle pour savoir lequel choisir: <https://aide. por>

Sommaire

Dans ce chapitre, nous avons vu une introduction aux tests de widget Flutter. Nous avons vu comment ils peuvent être utilisé pour tester les widgets individuellement et comment ils sont structurés avec le WidgetTester class dans la fonction testWidgets.

Nous avons également vu comment nous pouvons utiliser les outils Flutter pour explorer les performances des applications en détail et les outils disponibles pour inspecter la mémoire et l'utilisation du processeur tels que l'interface utilisateur de l'Observatoire et superposition de performances. Ensuite, nous avons vu l'évolution des outils avec le tout nouveau DevTools suite.

Enfin, nous avons exploré les étapes pour préparer notre application au déploiement en vérifiant informations et détails, modification de l'icône de l'application visible par l'utilisateur et exécution étapes spécifiques à la plateforme pour créer une application prête à être publiée.

Dans le prochain chapitre, nous passerons en revue quelques sujets importants liés au code natif avec canaux de la plateforme et vérifiez comment préparer votre application pour l'internationalisation.

[379]

Épisode 398

13

Amélioration de l'expérience utilisateur

Si vous souhaitez que votre application atteigne un niveau élevé, vous devez la garder ouverte en interaction avec le contexte utilisateur même s'il n'est pas en cours d'exécution. En plus, développer une application internationalisée et entièrement accessible lui permet de se développer progressivement. Dans ce chapitre, vous apprendrez à créer des processus exécutés en arrière-plan, à traduire votre application dans la langue cible, et ajoutez des fonctionnalités d'accessibilité qui améliorent la convivialité de l'application.

Les sujets suivants seront traités dans ce chapitre:

- Accessibilité dans Flutter
- Ajouter des traductions aux applications
- Communication entre natif et Flutter avec les canaux de la plateforme
- Création de processus d'arrière-plan
- Ajout de code spécifique à Android pour exécuter du code Dart en arrière-plan
- Ajout de code spécifique à iOS pour exécuter du code Dart en arrière-plan

Accessibilité dans Flutter et ajout de traductions vers des applications

L'ajout de l'internationalisation à une application mobile contribue à la croissance du marché, toucher un public plus large. De la même manière, rendre une application accessible est une étape importante pour atteindre autant de personnes que possible et avec une meilleure expérience. Flutter fournit des moyens de rendre les applications plus accessibles avec des composants axés sur les utilisateurs souffrant d'une forme de handicap.

Épisode 399

Amélioration de l'expérience utilisateur

Chapitre 13

Prise en charge de Flutter pour l'accessibilité

La mise en œuvre correcte de l'accessibilité dans les applications mobiles améliore l'expérience utilisateur et aide à augmenter le nombre d'installations et diminuer le nombre de désinstallations. Flutter a des composants pour fournir un support d'accessibilité:

Contraste : Flutter expose des outils afin que le développeur puisse coloriser les widgets de manière appropriée avec un contraste suffisant.

Vérifiez les spécifications du W3C contraste recommandé: <https://www.w3.org/TR/WCAG20-TECHS/>.

[org/TR/WCAG20-TECHS/](https://www.w3.org/TR/WCAG20-TECHS/)

Grandes polices : dans Flutter, les widgets de texte respectent ce paramètre du système d'exploitation lors de la détermination des tailles de police. Ils sont mis à l'échelle si l'utilisateur le souhaite.

Sous Android et iOS, nous pouvons activer les grandes polices grâce à l'accessibilité

paramètres dans les configurations du système d'exploitation.

Lecteurs d'écran : TalkBack sous Android et VoiceOver sous iOS s'activent visuellement les utilisateurs avec facultés affaiblies pour obtenir des commentaires vocaux sur le contenu de l'écran.

Flutter fournit le widget Sémantique pour que le développeur autorise le description de la signification des widgets pour que les lecteurs d'écran puissent fonctionner correctement. Consultez la documentation widget: <https://api.flutter.dev/>

Internationalisation de Flutter

Flutter fournit des widgets et des classes qui aident à l'internationalisation et au Flutter les bibliothèques elles-mêmes sont internationalisées. Cela se fait à l'aide de trois packages, intl, intl_translation et flutter_localizations. Jetons un œil à ces packages et examiner comment ils contribuent à la tâche d'internationalisation.

[381]

Épisode 400

Amélioration de l'expérience utilisateur

Chapitre 13

Le paquet intl

Le package Dart intl est la base des traductions dans Dart, comme indiqué sur sa page sur pub:

"Ce package fournit des fonctionnalités d'internationalisation et de localisation, y compris la messagerie traduction, pluriels et sexes, formatage et analyse de date / nombre, et bidirectionnel texte."

Avec ce package, nous avons des mécanismes pour charger les traductions à partir de fichiers .arb. Ce format est également pris en charge par la boîte à outils Google Translators. Chaque fichier .arb contient un seul JSON table qui mappe les ID de ressources aux valeurs localisées.

Le package intl_translation

Le package intl_translation est basé sur intl. Il n'est nécessaire que dans le développement phase et contient un outil pour générer et analyser les traductions depuis / vers les fichiers .arb. Avec ça package, nous pouvons traduire nos messages au format .arb puis les importer dans Dart à utiliser avec le package intl.

Le package flutter_localizations

Le package flutter_localizations fournit un ensemble de 52 langues (au moment de écrivant ce livre) à utiliser avec les widgets Flutter. Par défaut, les widgets Flutter sont uniquement fourni avec des localisations anglaises américaines, donc, pour prendre en charge d'autres langues, le Le package flutter_localizations peut être utilisé.

Ajouter des localisations à une application Flutter

La localisation dans Flutter est, comme toute autre chose, un widget. Nous allons utiliser package flutter_localizations pour configurer les traductions d'une application simple qui affiche un message unique, Hello Flutter. Nous allons prendre en charge l'anglais, l'espagnol et l'italien.

[382]

Épisode 401

*Amélioration de l'expérience utilisateur**Chapitre 13*

Dépendances

La première étape consiste à ajouter des dépendances de localisation au fichier pubspec.yaml et à les récupérer avec les packages de flutter, obtenez la commande:

```
dépendances:  
...  
  flutter_localizations:  
    sdk: scintillement  
  dev_dependencies:  
    intl_translation: ^ 0.17.3  
...
```

Comme mentionné précédemment, le premier est le package de localisation Flutter supplémentaire pour utiliser son widgets intégrés, et le second nous donne les outils pour générer du code Dart avec les messages à partir de fichiers .arb.

La classe AppLocalization

L'étape suivante consiste à créer une classe qui encapsule les valeurs localisées de l'application. La classe AppLocalizations, par exemple, qui serait très similaire dans toutes les applications, sauf les ressources de chaîne impliquées. Voici à quoi cela ressemble:

```
// fait partie de app_localization.dart  
import 'l10n / messages_all.dart';  
  
class AppLocalizations {  
  static Future <AppLocalizations> load(Locale locale) {  
    nom final de la chaîne =  
      locale.countryCode == null? locale.languageCode:  
    locale.toString();  
    final String localeName = Intl.canonicalizedLocale(nom);  
  
    retourne initializeMessages (localeName) .then ((bool _) {  
      Intl.defaultLocale = localeName;  
      retourne de nouvelles AppLocalizations ();  
    });  
  }  
  
  AppLocalizations statiques de (contexte BuildContext) {  
    return Localizations.of <AppLocalizations> (contexte, AppLocalizations);  
  }  
  
  String get title {  
    renvoyer Intl.message (  
      «Bonjour Flutter»,
```

[383]

Épisode 402

*Amélioration de l'expérience utilisateur**Chapitre 13*

```
    nom: 'titre',  
    desc: 'Le titre de l'application'  
  );  
}
```

```

String get bonjour {
  return Intl.message ('Bonjour', nom: 'bonjour');
}
}

```

AppLocalizations est utilisé pour encapsuler les ressources. Il peut être divisé en quatre pièces principales:

La fonction de chargement: Cela chargera les ressources de chaîne à partir de la locale souhaitée, comme vous pouvez le voir dans le paramètre.

La fonction de: Ce sera une aide comme pour tout autre InheritedWidget à faciliter l'accès à n'importe quelle chaîne à partir de n'importe quelle partie du code de l'application.

get functions: Ceux-ci répertorieront les ressources disponibles traduites dans notre application. Notez le wrapper Intl.message dans le retour; qui donnera l'apparence de l'outil intl cette classe et remplissez les initializeMessages pour nous avec le traductions.

initializeMessages: Cette méthode sera générée par l'outil intl. Remarque le fichier d'importation "l10n / messages_all.dart" qui sera généré dans le prochain étapes contient la méthode qui charge efficacement les messages traduits.

En plus de cette classe, nous devons créer une autre classe chargée de fournir le

AppLocalizations ressources à l'application. Voici à quoi cela ressemble:

```

classe AppLocalizationsDelegate étend
LocalisationsDelegate <AppLocalizations> {
  const AppLocalizationsDelegate ();

  @passer autre
  bool isSupported (paramètres régionaux) {
    return ['en', 'es', 'it'].contains (locale.languageCode);
  }

  @passer autre
  Charge future de <AppLocalizations> (paramètres régionaux locaux) {
    return AppLocalizations.load (paramètres régionaux);
  }

  @passer autre
  bool shouldReload (LocalizationsDelegate <AppLocalizations> old) {
    retourner faux;
}

```

[384]

Épisode 403

Amélioration de l'expérience utilisateur

Chapitre 13

```

  }
}
```

Il peut également être divisé en trois éléments principaux:

La fonction de chargement: Voici les informations de la documentation:

"La méthode load doit renvoyer un objet contenant une collection de ressources (généralement définies avec une méthode par ressource). "

Nous renvoyons notre classe AppLocalizations.load.

isSupported: comme son nom l'indique, il renvoie true si l'application prend en charge pour receivelocale.

shouldReload: Fondamentalement, si cette méthode retourne true, alors toute l'application les widgets seront reconstruits après le chargement des ressources. Vous voudrez généralement renvoie true si votre application change dynamiquement les paramètres régionaux.

Génération de fichiers .arb avec intl_translation

Après avoir défini ces classes, nous devons créer nos traductions de messages. Comme vous pouvez le voir dans la classe AppLocalizations, il n'y a que deux ressources de chaîne à traduire, titre et Bonjour. Comme indiqué précédemment, le processus de traduction se fait avec des fichiers .arb. Alors, il faut définir

fichiers .arb pour chacune des langues prises en charge (anglais, espagnol et italien, dans notre cas), et ces fichiers doivent contenir les ressources de chaîne traduites dans la langue cible.

La création de chacun de ces fichiers peut être fastidieuse, nous pouvons donc utiliser l'outil `intl_translation`, pour générer ces fichiers. Tout d'abord, nous créons un répertoire pour stocker les nouveaux fichiers — `lib / i10n`, dans ce exemple. Ensuite, nous générerons les fichiers .arb avec la commande suivante:

```
flutter pub run intl_translation: extract_to_arb --output-dir = lib / i10n
lib / app_localization.dart
```

Le dernier paramètre fait référence au fichier contenant la localisation de l'application class — `lib / app_localization.dart`, dans notre cas.

[385]

Épisode 404

Amélioration de l'expérience utilisateur

Chapitre 13

Cette commande générera un fichier appelé fichier `intl_messages.arb` dans `lib / i10n`, et ce fichier sert de modèle pour nos traductions:

```
{
  "@@ last_modified": "2019-04-22T21:32:20.153408",
  "title": "Hello world App",
  "@Titre": {
    "description": "Le titre de l'application",
    "type": "texte",
    "espaces réservés": {}
  },
  "Bonjour bonjour",
  "@salut": {
    "type": "texte",
    "espaces réservés": {}
  }
}
```

Nous pouvons créer les traductions souhaitées à partir de ce fichier en le copiant, en le renommant il `intl_<language_code>.arb`, et la traduction des ressources requises:

Consultez le GitHub pour le code source de tous les fichiers et un exemple.

Après cela, avec tout traduit, nous devons le rendre prêt à être utilisé dans notre application. Le processus est l'inverse de la génération de fichiers .arb:

```
flutter pub run intl_translation: generate_from_arb --output-
dir = lib / i10n lib / app_localization.dart lib / i10n / intl_en.arb
lib / i10n / intl_es.arb lib / i10n / intl_it.arb
```

Nous avons maintenant le code Dart généré contenant les ressources traduites. Nous n'allons pas touchez ce code directement lorsque nous devons ajouter des ressources; nous en faisons le fichiers `app_localization.dart` et `.arb`.

[386]

Épisode 405

*Amélioration de l'expérience utilisateur**Chapitre 13*

N'oubliez pas que la classe AppLocalization utilise initializeMessages de le fichier messages_all.dart. Maintenant, il est prêt à fournir des ressources localisées à l'application.

Utiliser des ressources traduites

Avec tous les fichiers générés et toutes les ressources traduites et prêtes à l'emploi, nous devons maintenant les utiliser correctement dans l'application. Pour ce faire, nous devons définir quelques propriétés du Classe MaterialApp. Voici à quoi ressemble notre classe d'application:

```
class MyApp étend StatelessWidget {
    // Ce widget est la racine de votre application.
    @passer autre
    Construction du widget (contexte BuildContext) {
        retourner nouveau MaterialApp (
            localisationsDélégués: [
                AppLocalizationsDelegate (),
                GlobalMaterialLocalizations.delegate,
                GlobalWidgetsLocalizations.delegate
            ],
            supportedLocales: [Locale ("en"), Locale ("es"), Locale ("it")],
            onGenerateTitle: (contexte BuildContext) =>
                AppLocalizations.of (context) .title,
            thème: nouveau ThemeData (
                primarySwatch: Colors.blue,
            ),
            home: nouveau MyHomePage (),
        );
    }
}
```

Nous devons définir les propriétés localizationsDelegates et supportedLocales. Tu répétez supportedLocales de vos délégués et définissez le tableau localizationDelegates avec AppLocalizationsDelegate, plus GlobalDelegates de le package flutter_localizations.

Dans la documentation, notez ce qui suit:

"Les éléments de la liste localizationsDelegates sont des usines qui produisent des collections de valeurs localisées. GlobalMaterialLocalizations.delegate fournit des chaînes localisées et d'autres valeurs de la bibliothèque de composants de matériaux. GlobalWidgetsLocalizations.delegate définit la direction du texte par défaut, de gauche à droite ou de droite à gauche, pour la bibliothèque de widgets."

Ainsi, GlobalWidgetsLocalizations et GlobalMaterialLocalizations sont probablement obligatoire si nous voulons rendre notre application complètement localisée.

[387]

Épisode 406

*Amélioration de l'expérience utilisateur**Chapitre 13*

Cette étape charge nos ressources dans notre application. Désormais, pour les utiliser efficacement, nous utilisons la méthode of dans notre classe AppLocalizations:

```
class MyHomePage étend StatelessWidget {
    MyHomePage ({clé clé}): super (clé: clé);
    @passer autre
    Construction du widget (contexte BuildContext) {
        retour échafaudage (
            appBar: AppBar (
                title: Texte ( AppLocalizations.of (context) .title ),

```

```

),
corps: Centre (
enfant: Texte (
  AppLocalizations.of(context).hello,
  style: Theme.of(context).textTheme.display1,
),
),
);
}
}

```

Avec cette méthode, nous avons accès à notre instance et à toutes les ressources que nous définissons avant. C'est tout pour rendre l'application localisée, comme vous pouvez le voir, nous devenons différents messages pour différents paramètres régionaux de l'appareil:

[388]

Épisode 407

Amélioration de l'expérience utilisateur

Chapitre 13

Maintenant que nous avons terminé l'internationalisation de Flutter, passons à la communication entre le code natif et Flutter.

Communication entre natif et Flutter avec canaux de plateforme

Flutter gagne de plus en plus d'adopteurs depuis 2018 avec la sortie de son premier version stable. L'une des principales raisons de cette adoption réside dans les installations pour développer une interface utilisateur belle, dynamique et fluide. Cependant, ce n'est pas seulement une application mobile qui peut avoir besoin; il a également une fonction à exécuter et nous devons nous occuper des différents hôtes de la plateforme, car de nombreuses fonctionnalités dépendent, telles que les suivantes:

- Bluetooth, caméra, capteurs et emplacement
- Autorisations utilisateur
- Notifications
- Stockage des fichiers et des préférences
- Partager des informations avec d'autres applications

L'échange entre le monde Flutter et la plateforme doit être aussi imperceptible que possible pour que le développeur ne se sente pas découragé en utilisant le framework.

Jusqu'à présent, nous avons utilisé des plugins pour implémenter des fonctionnalités qui dépendent d'un système en œuvre de la plate-forme sous-jacente à exécuter. Plugins et même l'application elle-même peuvent avoir besoin de communiquer d'une manière ou d'une autre avec le code de la plate-forme pour que tout cela fonctionne. Tous ceci est géré par le moteur Flutter, donc pour communiquer le code de nos applications Flutter avec natif Swift / Objective-C et Kotlin / Java, nous traiterons des **canaux de plateforme**.

Dans [Chapitre 9](#), *Développer votre propre plugin Flutter*, quand nous avons vu comment développer le notre Flutter plugin, nous avons eu une introduction aux **canaux de méthode**. Les canaux de méthode sont, par définition, une spécialisation d'un canal de plate-forme Flutter. Alors, voyons en détail comment tout cela fonctionne et examinez les canaux de méthode comme base pour les quelques sections suivantes.

[389]

Épisode 408

Amélioration de l'expérience utilisateur

Chapitre 13

Chaîne de plate-forme

Les applications Flutter sont hébergées dans une application native typique, c'est-à-dire que lorsque vous exécutez une application Flutter, il y a une application iOS ou Android native fonctionnant avec des délégations d'interface utilisateur à Flutter. Comme tu le sais déjà, Flutter rend toute l'interface utilisateur par elle-même, et pour que cela fonctionne, la couche native Flutter a tous le code nécessaire pour configurer un Android View ou iOS UIViewController dans lequel le cadre peut fonctionner.

Certains frameworks mobiles reposent sur la génération de code pour effectuer une conversion à partir de certains langage de premier niveau dans le langage natif, où vous écrivez presque toujours du code uniquement dans le langage spécifique au framework qui sera ensuite converti en langage natif (Kotlin / Java et Swift / Objective-C). Cela rend difficile pour le framework de garder son API aussi à jour que les plates-formes des hôtes, et comme Flutter entend être présent sur de nombreuses plates-formes, il serait encore plus difficile pour elle d'accomplir cela et d'évoluer en même temps.

Pour répondre à ce besoin, Flutter s'appuie sur un style de transmission de message flexible, appelé plateforme canal. Passons en revue sa structure:

[390]

Épisode 409

Amélioration de l'expérience utilisateur

Chapitre 13

Il s'agit de la vue de l'architecture des canaux de la plateforme Flutter. Site officiel: <https://flutter.dev/>

[dev/développement](#)

Comme illustré dans ce diagramme, les MethodChannels sont utilisés pour envoyer / recevoir des messages. Le diagramme montre comment les canaux de la plateforme fonctionnent en général:

L'application Flutter envoie des messages à la partie hôte / native (iOS ou Android) de l'application sur un canal de plate-forme.

La partie hôte / native de l'application écoute sur le canal de la plateforme, reçoit un message, et le traite via sa propre implémentation, en utilisant le système fourni les API et, enfin, renvoie un résultat à la partie Flutter appelante de l'application.

Comme les plugins, PlatformViews, vu précédemment dans [Chapitre 11, Plateforme Views and Map Integration](#), repose également sur le mécanisme de canal de plate-forme pour échanger des données.

Codecs de message

Comme nous l'avons vu jusqu'à présent, le MethodChannel est l'exemple principal et le plus couramment utilisé canal de plate-forme, car il fait abstraction de nombreuses complexités de la traduction des données. Dardez dans les langages de programmation natifs et vice versa.

Il existe également d'autres moyens de communication entre natif et Flutter, tels que BasicMessageChannel. Consultez le tutoriel officiel sur canaux de plateforme pour plus de détails: <https://flutter.dev/integration/>

Ceci est rendu possible grâce à l'utilisation des **codecs de message** standard Flutter. Codecs de message sont responsables de la tâche de traduction des données d'une langue à une autre. Il y a un une variété de codes de messages disponibles et nous pouvons, si nécessaire, créer les nôtres. Ils sont comme suit:

BinaryCodec: ce sont des messages binaires non codés représentés en utilisant ByteData. Sur Android, les messages seront représentés par

java.nio.ByteBuffer. Sur iOS, les messages seront représentés à l'aide de NSData.

JSONMessageCodec: il s'agit de messages JSON encodés en UTF-8. Sur Android, les messages sont décodés à l'aide de la bibliothèque org.json. Sur iOS, les messages sont décodé à l'aide de la bibliothèque NSJSONSerialization.

[391]

Épisode 410

Amélioration de l'expérience utilisateur

Chapitre 13

StringCodec: ce sont des messages String encodés en UTF-8. Sur Android, les messages sera représenté à l'aide de java.util.String. Sur iOS, les messages seront représenté à l'aide de NSString.

StandardMessageCodec: Cela utilise le binaire standard Flutter codage. Les valeurs décodées utiliseront List <dynamic> et Map <dynamic, dynamique>, quel que soit le contenu. Les valeurs du message sont traduites à partir de Dart types dans les types Android / iOS.

Consultez la documentation officielle sur le StandardMessageCodec classe pour voir comment les valeurs sont mappées de Dart à native et vice versa: <https://api.flutter.dev/>

[StandardMessageCodec](#)

MethodChannels utilise le StandardMessageCodec fourni par Flutter sous le capot par défaut pour effectuer la serialisation / deserialisation des données lorsque nous envoyons / recevons des messages avec ça.

Création de processus d'arrière-plan

Dans [Chapitre 2](#), *Programmation Dart intermédiaire*, nous avons vu l'approche Dart pour programmation: isolé. Avec cela, nous pouvons créer des travailleurs indépendants similaires à threads, mais ne partagent pas la mémoire et communiquent entre eux uniquement via des messages.

Dans le contexte des applications mobiles, nous devons également nous soucier de la concurrence. Aussi long les opérations peuvent entraîner un retard sur le rendu et ainsi de suite, Flutter fournit un moyen facile de générer un isolat, la fonction compute () .

La fonction Flutter compute ()

La méthode compute () est destinée à être un facilitateur pour la tâche de générer un nouveau isoler, lui envoyer un message et obtenir une réponse. Sa signature est la suivante:

```
Future<R> compute<Q, R>(rappel ComputeCallback<Q, R>, message Q, {  
  Chaîne debugLabel})
```

[392]

Épisode 411

Amélioration de l'expérience utilisateur

Chapitre 13

Quelques paramètres décrivent la demande adressée au nouvel isolat:

callback: Il s'agit d'une fonction de niveau supérieur à exécuter dans le nouvel Isolate.
Remarque ComputeCallback. Il existe des annotations de type générique <Q, R>; la première one, Q, désigne le type d'entrée du rappel, et R désigne le type de résultat du calcul.

La note de la documentation dit:

"L'argument de rappel doit être une fonction de niveau supérieur; pas une fermeture ou un instance ou méthode statique d'une classe."

message: il s'agit de la valeur du paramètre du type Q, qui sera envoyée à rappeler.

La note de la documentation dit:

"Il y a des limites sur les valeurs qui peuvent être envoyées et reçues vers et depuis isolé. Ces limitations contraignent les valeurs de Q et R qui sont possibles."

debugLabel: Ceci peut être utilisé pendant le développement, en donnant un nom à l'isolat pour une meilleure différenciation sur l'outil d'interface utilisateur de l'Observatoire lors du profilage.

La fonction compute () est idéale pour les calculs qui peuvent prendre plus de quelques millisecondes pour terminer, ce qui peut entraîner la perte de certaines images. Il y a aussi alternatives pour les calculs à court terme. N'oubliez pas l'utilisation des contrats à terme [Chapitre 2](#). *programmation de fléchettes intermédiaire* .

SendPort et ReceivePort

Comme indiqué précédemment, le message passé à la fonction compute () et la valeur de retour de lui doit respecter certaines limitations. Ces limitations viennent des isolats couche de communication. Les isolats, comme dit précédemment, communiquent entre eux via le messages. Ces messages sont envoyés et reçus via SendPort et

Instances ReceivePort.

Pour envoyer un message à un port isolé, nous devons d'abord obtenir une instance ReceivePort correspondant. La classe ReceivePort expose un getter sendPort qui est limité à l'isolat, afin que nous puissions lui envoyer des messages. Comment un isolat obtient-il ReceivePort de un autre isolat? Il le fait via la classe IsolateNameServer.

[393]

Épisode 412

Amélioration de l'expérience utilisateur

Chapitre 13

IsolateNameServer

La classe IsolateNameServer est un registre global d'isolats Dart, à partir duquel nous pouvons Inscrivez-vous et recherchez SendPorts et ReceivePorts. Simplement dit, un isolat peut enregistrer son ReceivePort via le IsolateNameServer.registerPortWithName méthode et d'autres isolats peuvent obtenir le SendPort correspondant avec la méthode IsolateNameServer.lookupPortByName () .

Un exemple de compute ()

Comme dit précédemment, pour créer un isolat pour effectuer de longs processus, nous utilisons la fonction compute (). Nous pouvons avoir n'importe quel type d'implémentation dans le callback isolate, qui sera passé à la fonction de calcul. La seule exigence est qu'il doit s'agir d'un fonction de haut niveau. Par exemple, consultez le code suivant:

```
import 'fléchette: io';

void backgroundCompute (args) {
    print ('rappel de calcul en arrière-plan');
    print ('calcul de fibonacci à partir d'un processus d'arrière-plan');

    int premier = 0;
    int seconde = 1;
    pour (var i = 2; i <= 50; i++) {
        var temp = seconde;
        deuxième = premier + deuxième;
        premier = temp;
        sommeil (durée (millisecondes: 200));
        print ("premier: $ premier, deuxième: $ deuxième.");
    }

    print ('fini de calculer fibo');
}
```

Cette méthode calcule les 50 premiers nombres de Fibonacci et les imprime dans les journaux de l'appareil. Comme tu peut voir, il contient un appel de veille, qui est un appel de blocage; cela signifie qu'aucune asynchrone les opérations peuvent être traitées dans l'isolat lorsqu'il est bloqué.

Nous pouvons exécuter un isolat pour exécuter ce rappel n'importe où dans une application Flutter en exécutant le suivant:

```
compute (backgroundCompute, null);
```

[394]

Épisode 413

Amélioration de l'expérience utilisateur

Chapitre 13

Cette fonction très utile résume toute la configuration nécessaire pour exécuter et communiquer avec un nouvel isolat. Nous l'envoyons, avec ou sans paramètres, et récupérons une option réponse en retour.

Un aspect important à noter, cependant, est que le nouvel isolat est un enfant du Flutter principal isoler l'application et ainsi, si l'application est arrêtée (c'est-à-dire lorsque l'utilisateur la fait glisser sur le plateau des applications), l'isolat enfant est également terminé.

Processus d'arrière-plan complet

Bien que très utile, la fonction compute () peut ne pas être ce dont nous avons besoin dans tous les cas.

Comme indiqué précédemment, l'isolat enfant créé par la fonction compute () se termine chaque fois que l'isolat parent se termine.

Dans certaines situations, nous pouvons vouloir exécuter du code totalement indépendant du application, comme dans les exemples suivants:

Nous pouvons le faire lors de la réception de notifications push et de la mise à jour des informations.

Nous n'avons pas besoin que l'application soit en cours d'exécution pour recevoir et traiter le push à distance notifications.

Un autre exemple est lorsque vous écoutez les changements de localisation de l'utilisateur ou que vous geofences.

La récupération des informations du serveur à partir d'un flux est un autre exemple.

Enfin, nous pourrions le faire lors du téléchargement de fichiers sur le serveur. En fonction de la taille des fichiers, les opérations peuvent nécessiter beaucoup de temps pour s'exécuter et bien d'autres.

Pour les cas d'utilisation où nous avons besoin d'un code à exécuter indépendamment de l'interface utilisateur de l'application, nous pouvons créer **des isolats sans tête**, c'est-à-dire un isolat qui n'est pas lié à l'isolat d'application principal et, si l'isolat principal se termine, cela n'affecte pas son exécution.

Jusqu'à ce stade d'écriture de ce livre, il n'y a pas d'API par défaut pour gérer ces cas d'utilisation, donc Les auteurs et développeurs de plugins qui ont besoin de ce type de fonctionnalité dans leur application doivent traiter les fondations de bas niveau du moteur Flutter pour créer un isolat d'arrière-plan et établir la communication entre les couches.

[395]

Épisode 414

Amélioration de l'expérience utilisateur

Chapitre 13

Pour créer un processus d'arrière-plan, nous pouvons diviser les responsabilités en langues et en application couches. Nous devons également vérifier ce que nous pouvons / ne pouvons pas faire avec le cadre et la plateforme sous-jacente. Prenons-le dans l'ordre:

1. Tout d'abord, nous devons définir un point d'entrée d'isolat d'arrière-plan Flutter, similaire au main () de notre application. L'isolat d'arrière-plan doit avoir son fonction de type principal.
2. Avec ce point d'entrée défini, nous pouvons démarrer l'isolat d'arrière-plan. Du perspective de l'application, nous procérons comme suit:
 - Nous envoyons une requête via un appel de méthode au côté natif de notre signalisation de l'application pour lancer le nouvel isolat
 - Du côté natif, nous créons la structure nécessaire et exécutons le nouveau isoler indépendamment de l'application qui a fait la demande
 - Avec l'isolat d'arrière-plan prêt et en cours d'exécution, nous informons le natif côté, donc il sait qu'il peut communiquer avec l'isolat
3. Depuis l'application, nous pouvons commencer à faire des demandes au côté natif qui traiter les éléments liés à la structure Flutter et déléguer à l'arrière-plan isoler.

Ce processus semble être beaucoup plus complexe que nécessaire, et bien que pas simple, le La communauté Flutter vise à améliorer cela dès que possible pour faire la tâche d'arrière-plan traitement dans Dart plus simple.

Jetez un œil aux problèmes de Flutter sur: <https://github.com/flutter/flutter/issues> de mise à jour des alternatives de traitement

[396]

Épisode 415

Amélioration de l'expérience utilisateur

Chapitre 13

La communication peut être simplifiée comme suit:

La communication directe entre les isolats principaux et de fond est facultative et difficile à maintenir car nous devons nous occuper des aspects de fonctionnement. Le moyen le plus simple jusqu'à présent est de faire des demandes ou, même, utiliser un système d'exploitation qui déclenche l'envoi de l'arrière-plan vers isoler indépendamment de l'isolat de l'application principale.

Créons un exemple en utilisant le même algorithme de Fibonacci qu'auparavant. Cette fois, on commence l'isoler de l'application, comme avant, mais si nous terminons l'application (c'est-à-dire en le faisant glisser hors du plateau des applications), les journaux seront toujours imprimés dans les journaux de l'appareil en tant que le processus fonctionnera toujours en arrière-plan.

Initier le calcul

Depuis l'application, lorsque l'on clique sur le bouton de calcul, il faut initialiser le processus nous avons vu avant. La première étape consiste à appeler une méthode via un canal de méthode. nous ont créé l'exemple dans une structure de plugin, il vous sera donc facile de le changer et même l'utiliser dans vos applications. Le plugin sera responsable de l'abstraction de l'isolat processus de création afin qu'il soit transparent pour l'application. Sa seule méthode est CalculateInBackgroundProcess, qui est appelé depuis l'application:

```
HandsOnBackgroundProcess.calculateInBackgroundProcess();
```

Consultez l'exemple hands_on_background_process sur GitHub pour

le code source complet.

[397]

Épisode 416

Amélioration de l'expérience utilisateur

Chapitre 13

Ce code précédent appelle la méthode du plugin qui est responsable du lancement du processus cela va comme suit:

```
const pluginChannel = MethodChannel ('com.example.handson / plugin_channel');

class HandsOnBackgroundProcess {
    static void CalculateInBackgroundProcess () async {
        final callbackHandle = PluginUtilities.getCallbackHandle (
            fondIsolatPrincipal
        );

        attendre pluginChannel.invokeMethod (
            "initBackgroundProcess",
            [callbackHandle.toRawHandle ()]
        );
    }
}
```

Comme vous pouvez le voir, tout d'abord, nous définissons un canal de méthode nommé com.example.handson / plugin_channel pour les appels de plugin; c'est typiquement le premier étape dans les plugins. Ensuite, dans la méthode CalculateInBackgroundProcess (), nous faisons le Suivant:

Nous obtenons une poignée vers le nouveau point d'entrée d'isolat d'arrière-plan. nous utilisons l'utilitaire PluginUtilities.getCallbackHandle fourni par le framework pour obtenir l'identifiant du rappel à passer au côté natif du application. De cette façon, plus tard, du côté natif, nous pouvons récupérer ce rappel et exécutez l'isolat d'arrière-plan avec lui comme point d'entrée.
Après avoir récupéré le handle, nous invoquons la méthode "initBackgroundProcess", en lui passant la poignée. Cette méthode effectuera le travail d'isolement mentionné précédemment.

Jetons d'abord un coup d'œil au point d'entrée Dart de l'isolat, puis vérifions le code nécessaire pour le faire fonctionner correctement.

L'arrière-plan isoler

Le rappel Dart transmis à la partie native du plugin via le handle vu avant est responsable du calcul du Fibonacci, comme avant. Cependant, ce n'est pas exactement le même:

```
void backgroundIsolateMain () {
    print ('isoler le point d'entrée en arrière plan en cours d'exécution');
    const backgroundchannel = MethodChannel (
        'com.example.handson / background_channel'
```

[398]

Épisode 417

Amélioration de l'expérience utilisateur

Chapitre 13

```
);

WidgetsFlutterBinding.ensureInitialized ();

backgroundchannel.setMethodCallHandler ((appel MethodCall) async {
    if (call.method == 'calculer') {
```

```

print ('calcul de fibonacci à partir d'un processus d'arrière-plan');

int premier = 0;
int seconde = 1;
pour (var i = 2; i <= 50; i++) {
    var temp = seconde;
    deuxième = premier + deuxième;
    premier = temp;
    sommeil (durée (millisecondes: 500));
    print ("premier: $ premier, deuxième: $ deuxième.");
}

print ('fini de calculer fibo');
backgroundchannel.invokeMethod ("calculFinished");
}
});
backgroundchannel.invokeMethod ("backgroundIsolateInitialized");
}

```

Comme vous pouvez le voir, il a quelques changements:

1. L'isolat d'arrière-plan commence par configurer un canal de méthode, pas le même que avant. Maintenant, nous en créons un nommé com.example.hanson / background_channel. Il est utilisé pour établir communication avec le code natif exécuté en arrière-plan (service sur Android et exécution en arrière-plan sur iOS).
2. Définissez le gestionnaire de la méthode de calcul, afin que le code natif puisse l'appeler pour démarrer le calcul. Bien que ce ne soit pas vraiment nécessaire dans ce cas (nous pourrions commencer calcul directement dans le corps du point d'entrée), c'est bon pour l'exemplification.
3. Après avoir configuré le canal de méthode, nous informons le côté natif avec un appel à backgroundIsolateInitialized. Après cela, tout est prêt du côté de Dart.

Pour le côté Dart de l'exécution en arrière-plan, nous ne devons implémenter qu'une seule fois. Puis pour chacune des plates-formes (Android / iOS), nous devons mettre en place l'environnement pour cet isolat pour courir.

[399]

Épisode 418

Amélioration de l'expérience utilisateur

Chapitre 13

Ajout de code spécifique à Android pour exécuter Dart code en arrière-plan

Sous Android, il y a le concept de Services, qui est le moyen idéal de faire fonctionner code d'application en arrière-plan, indépendamment de l'exécution principale de l'application. Donc, en gros, nous devons créer une méthode de service, lier le nouvel isolat d'arrière-plan et exécuter.

Lisez la documentation officielle sur les services Android: <https://developer.android.com/>

La classe HandsOnBackgroundProcessPlugin

La première étape consiste à configurer le plugin, comme nous l'avons déjà fait au [chapitre 9](#), *Développer votre Posséder le plugin Flutter*. Cela commence par une implémentation de la méthode static registerWith qui notifie au moteur Flutter l'existence de l'instance de plugin:

```

classe HandsOnBackgroundProcessPlugin (
    contexte de val privé: Contexte
): MethodChannel.MethodCallHandler {
    objet compagnon {
        ...
    }
}
```

```

@JvmStatic
fun registerWith(registraire: PluginRegistry.Registrar) {
    val channel = MethodChannel(
        registrar.messenger(),
        "com.example.handsOn / plugin_channel"
    )
    plugin val = HandsOnBackgroundProcessPlugin (
        registrar.context()
    )
    channel.setMethodCallHandler(plugin)
}
...
}

```

[400]

Épisode 419

Amélioration de l'expérience utilisateur

Chapitre 13

Comme vous pouvez le voir, il configure le canal de méthode, appelé com.example.handsOn / plugin_channel, utilisé pour initialiser le calcul via la méthode initBackgroundProcess:

```

override fun onMethodCall(appel: MethodCall, résultatat: MethodChannel.Result?) {
    val args = call.arguments() as? ArrayList<*>
    if (call.method == "initBackgroundProcess") {
        val callbackHandle = args?.get(0) comme? Long?: Retour
        executeBackgroundIsolate(contexte, callbackHandle)
    }
}

```

Pour gérer la méthode initBackgroundProcess, il récupère le handle de rappel à venir de Dart. Pour le récupérer correctement, il est analysé avec le type Long (int dans Dart) selon le Classe StandardMessageCodec.

L'exécution de l'isolat d'arrière-plan se fait en deux étapes. Le premier se fait en executeBackgroundIsolate (), comme suit:

```

...
fun privé executeBackgroundIsolate(contexte: contexte, callbackHandle:
Longue) {
    préférences val = context.getSharedPreferences (
        SHARED_PREFERENCES_KEY,
        IntentService.MODE_PRIVATE
    )
    preferences.edit().putLong(ARG_CALLBACK_KEY, callbackHandle).apply()

    startBackgroundService(contexte)
}
...

```

Tout d'abord, la méthode stocke la valeur du handle dans un fichier SharedPreferences. Ensuite, il demande l'exécution du service d'arrière-plan via le startBackgroundService () méthode:

Les préférences partagées sont utilisées dans Android pour stocker des données clé-valeur dans un simple et voie privée. Il est utilisé ici car nous ne pouvons pas transmettre de paramètres au service constructeurs car ils ne doivent pas obtenir d'arguments.

[401]

Épisode 420

Amélioration de l'expérience utilisateur

Chapitre 13

Vous pouvez en savoir plus sur les préférences partagées dans le site officiel Documentation: <https://developer.android.com/training/basics/fragments/prefs>

startBackgroundService () fait simplement la demande au système Android pour initialiser le service d'arrière-plan:

```
...
amusement privé startBackgroundService (contexte: Contexte) {
    intention val = intention (
        le contexte,
        BackgroundProcessService :: class.java
    )
    context.startService (intention)
}
...
...
```

La partie restante du travail est effectuée dans la classe BackgroundProcessService.

La classe BackgroundProcessService

La classe BackgroundProcessService est le service Android qui s'exécutera pendant notre isolat est en cours d'exécution. Comme elle est en arrière-plan, l'application peut être fermée et l'isolat fonctionnera normalement.

Encore une fois, il est important de consulter la documentation du service Android mentionné précédemment pour comprendre comment fonctionne le cycle de vie.

L'exécution du service est entièrement gérée par le système Android; nous n'en avons pas le contrôle total, nous devons donc réagir aux événements fournis par le système pour exécuter notre isolat basé sur le État du service.

Tout commence par la méthode onCreate, lorsque le système crée notre méthode Service et nous pouvons mettre en place toutes les ressources nécessaires à son fonctionnement. C'est un bon endroit pour commencer notre isolat de fond:

```
classe BackgroundProcessService: Service (), MethodChannel.MethodCallHandler
{
    outrepasser le plaisir onCreate () {
        super.onCreate ()
    }
}
```

[402]

Épisode 421

Amélioration de l'expérience utilisateur

Chapitre 13

```
createNotification ()
FlutterMain.ensureInitializationComplete (applicationContext, null)
startBackgroundIsolate ()
}
...
}
```

Comme vous pouvez le voir, cela fait plus que simplement initialiser notre isolat. Décomposons-le:

1. Tout d'abord, nous mettons en place une notification via la méthode createNotification (). la notification est placée sur la barre d'état Android et fait fonctionner notre service

le mode de premier plan. Fondamentalement, les services qui s'exécutent en arrière-plan sont plus susceptibles d'être tués par le système en cas de manque de ressources. Premier plan les services, en revanche, ont une priorité plus élevée dans le système et sont moins susceptibles d'être terminé dans ce cas.

2. Ensuite, nous utilisons le FlutterMain.ensureInitializationComplete (applicationContext, null), qui affirme que le moteur Flutter est configuré et que nous pouvons utiliser des choses telles que comme canaux de plate-forme.
3. Enfin, nous démarrons l'isolat avec l'appel startBackgroundIsolate () .

La méthode startBackgroundIsolate () est la méthode principale et la plus complexe de cette classe. Il est responsable de la configuration de la structure nécessaire à l'exécution de l'isolat d'arrière-plan. Cela se passe comme suit:

```
amusement privé startBackgroundIsolate () {
    préférences val = applicationContext.getSharedPreferences (
        SHARED_PREFERENCES_KEY,
        MODE_PRIVATE
    )
    val callbackHandle = preferences.getLong (ARG_CALLBACK_KEY, 0L)
    if (callbackHandle == 0L) retour
    rappel val =
        FlutterCallbackInformation.lookupCallbackInformation (
            callbackHandle
        ) ?: revenir

    sBackgroundFlutterView = FlutterNativeView (ceci, vrai)
    chemin val = FlutterMain.findAppBundlePath (applicationContext)
    val args = FlutterRunArguments ()
    args.bundlePath = chemin
    args.entrypoint = callback.callbackName
    args.libraryPath = callback.callbackLibraryPath

    sBackgroundFlutterView?.runFromBundle (args)
```

[403]

Épisode 422

Amélioration de l'expérience utilisateur

Chapitre 13

```
backgroundChannel = MethodChannel (
    sBackgroundFlutterView,
    "com.example.hanson / background_channel"
)
backgroundChannel?.setMethodCallHandler (ceci)

sPluginRegistrantCallback? .registerWith (
    sBackgroundFlutterView? .pluginRegistry
)
}
```

Cette méthode initialise et enregistre une nouvelle instance de plugin d'arrière-plan dans le Flutter moteur comme dans les applications normales. Le processus est un peu plus délicat, alors voyons comment nous allons procéder:

1. Tout d'abord, nous obtenons le rappel Dart qui est le point d'entrée du nouvel arrière-plan isoler. Pour ce faire, nous obtenons le handle des préférences partagées stockées et utilise la méthode FlutterCallbackInformation.lookupCallbackInformation pour récupérer les informations de rappel nécessaires à son exécution.
2. Ensuite, nous instancions une nouvelle méthode FlutterNativeView. Cette vue est utilisée pour disposer d'un environnement approprié pour l'exécution du nouvel isolat. Sous Android, c'est ainsi que le Le moteur Flutter fonctionne. Rappelez-vous, View est passé à notre côté Dart pour le travail d'application dessus. Notez le deuxième paramètre passé à le constructeur FlutterNativeView, true, ce qui signifie que la vue s'exécutera dans le arrière-plan et n'a pas besoin d'une surface sur laquelle dessiner.
3. Pour enfin exécuter l'isolat, nous utilisons la méthode runFromBundle () de l'instance FlutterNativeView que nous avons vue auparavant. Cette méthode nécessite une instance de FlutterRunArguments pour identifier ce qu'elle va exécuter. Notre variable args contient les informations que nous avons obtenues du callback, telles que son

4. Après avoir exécuté l'isolat d'arrière-plan, nous créons une instance à l'arrière-plan canal de méthode nommé com.example.handsOn / background_channel, juste comme nous l'avons fait du côté de Dart.
5. La dernière étape consiste à enregistrer l'instance de plugin dans le registre Flutter avec le aide de la propriété sPluginRegistrantCallback. Cette propriété doit être transmis manuellement à la classe Service en quelque sorte. Pourquoi? Battement enregistre automatiquement le plugin dans le thread principal lorsque vous l'utilisez (appelez-vous la méthode static registerWith que nous devons implémenter pour notre plugins). PluginRegistrantCallback est la façon dont nous le faisons manuellement. Par cela, nous pouvons enregistrer un plugin n'importe où, comme les endroits où registerWith est pas recherché (notre service, dans ce cas).

[404]

Épisode 423

Amélioration de l'expérience utilisateur

Chapitre 13

Consultez la documentation pour en savoir plus sur les threads sous Android:
<https://./flu>
[devs # comm...](#)

La propriété PluginRegistrantCallback

Nous transmettons l'instance PluginRegistrantCallback à la classe Service dans l'exemple projet. Nous créons un descendant de la classe FlutterApplication, qui servira de notre rappel du registrant au service:

```
Application de classe: FlutterApplication(),
PluginRegistry.PluginRegistrantCallback {
    outrepasser le plaisir onCreate () {
        super.onCreate ()
        Log.w ("CONTEXTE", "application")
        BackgroundProcessService.setPluginRegistrant (ce)
    }

    remplacer fun registerWith (registre: PluginRegistry?) {
        GeneratedPluginRegistrant.registerWith (registre)
    }
}
```

Comme vous pouvez le voir, nous transmettons l'instance d'application à l'instance de service afin qu'elle puisse pour vous inscrire dans le moteur Flutter. Nous devons également définir notre classe d'application dans AndroidManifest.xml pour que cela fonctionne:

```
<manifest xmlns: android = "http://schemas.android.com/apk/res/android"
    package = "com.example.hands_on_background_process_example">
    <application
        android: name = ". Application"
        android: label = "hands_on_background_process_example"
    >
    ...
</manifest>
```

Après avoir configuré le plugin et l'isolat d'arrière-plan, nous devons communiquer avec lui pour commencez les calculs. Tout ce que nous devons faire est de gérer les appels de méthode en arrière-plan canal de méthode que nous avons défini:

```
override fun onMethodCall (appel: MethodCall, résultat: MethodChannel.Result?) {
    if (call.method == "backgroundIsolateInitialized") {
        backgroundChannel?.invokeMethod ("calculer", null)
```

[405]

Épisode 424

*Amélioration de l'expérience utilisateur**Chapitre 13*

```

} else if (call.method == "calculFinished") {
    sBackgroundFlutterView?.destroy()
    sBackgroundFlutterView = null
    shutdownService()
} autre {
} // méthode 'calculer' à partir de ce canal, gérée sur l'isolat Dart.
}

```

Notre instance `BackgroundProcessService` est définie comme le gestionnaire de méthode du appels de canal de méthode d'arrière-plan:

La méthode nommée `backgroundIsolateInitialized` est appelée à partir du `background` isoler quand il est prêt et, en réponse à cela, nous commençons le calcul invoquant `calculer` dans le même canal.

De plus, chaque fois que le calcul se termine et que l'isolat d'arrière-plan Dart appelle le `calculFinished`, notre instance `FlutterNativeView` qui contient l'isolat est détruit et le service arrêté avec un appel au `shutdownService()`, qui supprime simplement la notification définie avant et tue le service.

C'est tout pour la mise en œuvre Android; avec cela, même si nous mettons fin à notre application en le faisant glisser hors de la barre des applications, l'isolat d'arrière-plan fonctionnera jusqu'à ce qu'il finitions.

Ajout de code spécifique à iOS pour exécuter du code Dart dans l'arrière-plan

Les choses sont différentes dans iOS. L'exécution en arrière-plan est beaucoup plus restreinte qu'avec Android. Le concept de service n'existe pas, et nous avons quelques instants que nous pouvons exécuter code en arrière-plan.

La majorité des cas d'utilisation sont couverts par `UIBackgroundModes`, où une application peut définir les modes d'arrière-plan pris en charge, puis est autorisé à exécuter des types spécifiques de exécution en arrière-plan. Nous pouvons, par exemple, faire ce qui suit:

- Avoir le mode d'arrière-plan Audio et AirPlay qui définit l'application comme capable de lecture de contenu audible à l'utilisateur ou enregistrement audio en arrière-plan.
- Recevoir des mises à jour d'emplacement lorsque l'emplacement met à jour le mode d'arrière-plan.
- Kiosque est un mode de téléchargement dans lequel l'application peut télécharger et traiter le contenu d'un magazine ou d'un journal en arrière-plan.

[406]

Épisode 425

*Amélioration de l'expérience utilisateur**Chapitre 13*

Consultez le guide officiel d'exécution en arrière-plan de l'iOS Documentation: <https://developer.apple.com/documentation/backgroundfetching>

[Documentation / iPhone / BackgroundExecution /](#)

Une grande partie du travail est similaire à Android, à l'exception de la partie Service. Alors, commençons par le définition du plugin.

Le SwiftHandsOnBackgroundProcessPlugin classe

L'enregistrement et la configuration du plugin se font de la même manière que la classe `HandsOnBackgroundProcessPlugin`. Cette fois, dans le `register()` static fonction, nous avons ce qui suit:

```
registre public de func statique (avec registrar: FlutterPluginRegistrar) {
    laissez channel = FlutterMethodChannel (
        nom: "com.example.handsOn / plugin_channel",
        binaryMessenger: registrar.messenger ()
    )
    let instance = SwiftHandsOnBackgroundProcessPlugin (
        registrarie: registrarie
    )
    registrar.addMethodCallDelegate (instance, canal: canal)
}
```

Comme dans la version Android, il configure le canal de méthode appelé

com.example.handsOn / plugin_channel, qui sert à initialiser le calcul via la méthode initBackgroundProcess, comme vous pouvez le voir:

```
handle de fonction publique (
    _appel: FlutterMethodCall,
    résultatat: @escaping FlutterResult
) {
    if (call.method == "initBackgroundProcess") {
        guard let args = call.arguments as? NSArray else {
            revenir
        }
        guard let handle = args [0] comme? Int64 else {
            revenir
        }
        executeBackgroundIsolate (handle: handle)
    }
}
```

[407]

Épisode 426

Amélioration de l'expérience utilisateur

Chapitre 13

Dans ce cas, comme nous n'avons pas de séparation en tant que service, nous commençons l'exécution du arrière-plan isoler immédiatement de l'appel.

L'exécution de l'isolat d'arrière-plan dans la méthode executeBackgroundIsolate () va comme suit:

```
func privée executeBackgroundIsolate (handle: Int64) {
    _backgroundRunner = FlutterEngine.init (
        nom: "BackgroundProcess",
        projet: nul,
        allowHeadlessExecution: vrai
    )
    garde let info = FlutterCallbackCache.lookupCallbackInformation (
        manipuler
    ) autre {
        revenir
    }
    laissez entrypoint = info.callbackName
    laissez uri = info.callbackLibraryPath
    _backgroundRunner!.run (
        withEntryPoint: entrée,
        libraryURI: uri
    )

    _backgroundChannel = FlutterMethodChannel (
        nom: "com.example.handsOn / background_channel",
        binaryMessenger: _backgroundRunner!
    )
    _registrar.addMethodCallDelegate (
        soi,
        canal: _backgroundChannel!
    )
    SwiftHandsOnBackgroundProcessPlugin._registerPlugins? (
        _backgroundRunner!
    )
}
```

On peut, encore une fois, décomposer l'exécution en plusieurs étapes:

1. Tout d'abord, nous stockons une instance de la classe FlutterEngine dans le

Propriété `_backgroundRunner`. Cette instance sera notre plugin Flutter qui être le pont, comme `FlutterNativeView` était sur Android.

[408]

Épisode 427

Amélioration de l'expérience utilisateur

Chapitre 13

2. Ensuite, nous obtenons notre point d'entrée de la poignée de rappel via l'utilitaire `FlutterCallbackCache.lookupCallbackInformation()`. Tous les informations sont égales à celles que nous obtenons dans Android. Ici, nous utilisons le point d'entrée et `uri` pour exécuter l'arrière-plan isoler à travers `le _backgroundRunner!.run (withEntrypoint: entrystore, LibraryURI: uri) appel.`
3. Après avoir exécuté l'isolat, la partie finale est très similaire à Android. Nous créons le canal nommé `com.example.handson / background_channel` pour le communication et nous définissons son gestionnaire comme l'instance de plugin elle-même.
4. Enfin, nous enregistrons le plugin en arrière-plan via le `_registerPlugins callback`, tout comme `PluginRegistrantCallback` dans Android.

Cette dernière étape n'est pas vraiment nécessaire sous iOS. Il n'y a *pas d'* autre arrière-plan thread en cours d'exécution à côté de notre application. Il est déplacé vers un état d'arrière-plan mais le plugin est toujours enregistré normalement. Si notre application était exécuté dans une clé `UIBackgroundMode` comme mentionné précédemment, ce l'enregistrement serait toujours important.

Après avoir lancé l'isolat d'arrière-plan, nous pouvons, à nouveau, gérer les appels depuis l'arrière-plan canal:

```
public func handle (_ appel: FlutterMethodCall, résultat: @escaping
FlutterResult) {

    if (call.method == "initBackgroundProcess") {
        // ... vu précédemment
    } else if (call.method == "backgroundIsolateInitialized") {
        self.taskID = UIApplication.shared.beginBackgroundTask {
            self.taskID = .invalid
        }
        _backgroundChannel?.invokeMethod ("calculer", arguments: nil)
    } else if (call.method == "calculFinished") {
        if (self.taskID! == nil && self.taskID! == .invalid) {
            UIApplication.shared.endBackgroundTask (self.taskID!)
            self.taskID = .invalid
        }
        // terminer la tâche d'arrière-plan
    }
}
```

[409]

Épisode 428

Amélioration de l'expérience utilisateur

Chapitre 13

Bien que ce soit différent, l'idée de base de la gestion des méthodes est similaire:

1. Lorsque la méthode nommée `backgroundIsolateInitialized` est appelée, nous invoquer la méthode de calcul correspondante, afin qu'il effectue des calculs et se connecte à la console Flutter. Avant cela, nous enregistrons une tâche d'arrière-plan iOS. Ce informera le système que nous avons besoin d'un peu plus de temps pour conclure nos travaux et l'empêcher de se terminer avant que prévu. Rappelez-vous, iOS est très restrictif dans les tâches d'arrière-plan.
2. Lors d'un appel au calcul Terminé, nous notifions simplement au système que notre tâche est fini avec `UIApplication.shared.endBackgroundTask(self.taskID!)` et est sûr de déplacer notre application à l'état *suspendu*.

Il est fondamental pour vous de comprendre pourquoi et quand cela peut être utilisé.

https://.. / démanaging_y_fond / extending_ ,

Tout comme dans Android, le plugin d'arrière-plan iOS est également enregistré. Nous faisons cela avec le rappel `_registerPlugins`. Il est transmis au plugin la fonction statique `setPluginRegistrantCallback()` qui est appelée dans l'application

Classe `AppDelegate`, très similaire à Android:

```
@UIApplicationMain
@objc classe AppDelegate: FlutterAppDelegate {
    remplacer l'application func (
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?
    ) -> Bool {
        GeneratedPluginRegistrant.register (avec: self)
        SwiftHandsOnBackgroundProcessPlugin.setPluginRegistrantCallback (
            registerPlugins: registerPlugins
        )
        retourne super.application (
            application,
            didFinishLaunchingWithOptions: options de lancement
        )
    }
}
```

[410]

Épisode 429

Amélioration de l'expérience utilisateur

Chapitre 13

Un peu différente d'Android, la fonction `registerPlugins` est une fonction de premier niveau, comme suit:

```
func registerPlugins (registre: FlutterPluginRegistry) {
    GeneratedPluginRegistrant.register (avec: registre)
}
```

Comme vous pouvez le voir, il est similaire à celui défini dans l'application Android, qui sert à enregistrez les plugins via l'utilitaire `GeneratedPluginRegistrant.register`.

Découvrez plus sur le filtre dans iOS: <https:// / Flutter dev / docs / commencé / fil>

Après cela, notre application se comporte de la même manière qu'Android et nous avons imprimé tous nos journaux, même en arrière-plan:

[411]

Épisode 430

Amélioration de l'expérience utilisateur

Chapitre 13

Sommaire

Dans ce chapitre, nous avons vu des méthodes avancées pour rendre notre application plus conviviale et interactif. Nous avons commencé par apprendre les outils disponibles axés sur l'accessibilité des utilisateurs fournis par le cadre Flutter.

Ensuite, nous avons vérifié comment nous pouvons ajouter des traductions aux applications Flutter, en générant des fichiers .arb, créer plusieurs traductions, les importer dans Dart et les appliquer à notre Classe MaterialApp.

Enfin, nous avons examiné les options de traitement en arrière-plan avec Flutter, allant du très fonction compute () utile à un service d'arrière-plan sur Android et modes d'arrière-plan sur iOS. Nous avons également vu les caractéristiques et les limites de chaque plateforme dans cet aspect.

Dans le chapitre suivant, nous allons examiner les manipulations graphiques des widgets et comment nous pouvons transformer des widgets et dessiner des formes personnalisées sur le canevas.

Épisode 431

14

Manipulations graphiques des widgets

Utiliser les widgets tels qu'ils sont par défaut est suffisant pour créer une belle application Flutter, mais étendre les widgets avec des transformations de mise en page, telles que l'opacité, les rotations et décorations, peuvent améliorer encore l'expérience utilisateur. Dans ce chapitre, vous apprendrez à ajouter ces transformations en un widget. En outre, vous apprendrez à modifier un widget en ajoutant des transformations graphiques avec la classe `Transform` et utilisez le canevas pour dessiner un widget personnalisé.

Les sujets suivants seront traités dans ce chapitre:

- Transformer des widgets avec la classe `Transform`
- Explorer les types de transformations
- Ajouter des transformations à vos widgets
- Utilisation de peintres personnalisés et de la toile

Transformer des widgets avec le Transform classe

Parfois, nous devons changer l'apparence d'un widget. En réponse à l'entrée de l'utilisateur ou pour faire des effets sympas dans la mise en page, nous devrons peut-être déplacer le widget sur l'écran, changer sa taille, voire la déformer un peu.

Si vous avez déjà essayé de faire cela dans des langages de programmation natifs, vous avez peut-être trouvé quelques difficultés. Flutter, comme vous vous en souvenez, est fortement axé sur la conception de l'interface utilisateur et propose de faciliter la vie du développeur.

Épisode 432

Le widget Transformer

Le widget `Transform` est l'un des meilleurs exemples de la puissance du framework Flutter et sa cohérence. C'est un widget à usage unique qui applique simplement une transformation graphique à son enfant et rien de plus. Avoir des widgets axés sur un seul objectif est fondamentale pour une meilleure structure de mise en page, et Flutter le fait très bien.

Le widget Transformer, comme son nom l'indique, effectue une seule tâche: il **transforme** son enfant sous-jacent. Bien que sa tâche soit très complexe, elle résume la majeure partie de cette complexité pour le développeur. Jetons un coup d'œil à son constructeur:

```
const Transform ({  
  Clé clé,  
  @ transformation Matrix4 requise,  
  Origine du décalage,  
  Alignement Alignement géométrie,  
  bool transformHitTests: true,  
  Enfant du widget  
})
```

Comme vous pouvez le voir, en plus de la propriété de clé typique, ce widget n'a pas besoin de beaucoup arguments pour faire son travail. Voyons ces arguments:

transform: c'est la seule propriété obligatoire (annotation `@required`) utilisée pour décrire la transformation qui sera appliquée au widget enfant.

Un objet Matrix4, c'est une matrice à quatre dimensions (4D) qui décrit le transformation de manière mathématique. Il y aura plus de détails plus tard.

origine: il s'agit de l'origine du système de coordonnées auquel appliquer le matrice de transformation. L'origine est spécifiée par le type Offset, représentant, dans ce cas, un point (x, y) dans le système cartésien qui est relatif au coin supérieur gauche coin du widget de rendu.

alignement: comme l'origine, il peut être utilisé pour manipuler la position du matrice de transformation appliquée. Nous pouvons l'utiliser pour spécifier l'origine de manière plus flexible façon, comme l'origine nous oblige à utiliser des valeurs de position réelles. Rien ne t'empêche d'utiliser à la fois l'origine et l'alignement.

transformHitTests: Ceci spécifie si les **tests de succès** (c'est-à-dire les taps) sont évalué dans la version transformée du widget.

child: Il s'agit du widget enfant auquel la transformation sera appliquée.

[414]

Épisode 433

Manipulations graphiques des widgets

Chapitre 14

Comprendre la classe Matrix4

Dans le fondement des transformations géométriques, il y a les mathématiques. Dans Flutter, les transformations sont représentées dans une matrice 4D. Outre des méthodes telles que l'ajout de matrice ou multiplication, la classe Matrix4 contient des méthodes qui aident à la construction et manipulation des transformations géométriques. Certains d'entre eux sont les suivants:

rotation: `rotateX()`, `rotateY()` et `rotateZ()` sont quelques exemples de méthodes qui font pivoter la matrice sur un axe spécifique.

scale: `scale()`, avec quelques variantes, est utilisé pour appliquer une échelle sur la matrice en utilisant valeurs doubles des axes correspondants (x, y et z) ou par vecteur représentations avec les classes `Vector3` et `Vector4`.

translation: Comme avant, nous pouvons traduire la matrice en utilisant le `translate()` avec des valeurs x, y ou z spécifiques et `Vector3` et `Vector4` les instances.

skew: Ceci est utilisé pour incliner la matrice autour de l'axe X avec `skewX()` ou axe Y avec `skewY()`.

Consultez la documentation officielle de Matrix4 pour tous les possibilités offertes par cette classe: <https://api.flutter.dev/flutter/math/Matrix4-class.html>

`math/` `cl`
`transfor`

c'est la L...
Transformer.

Explorer les types de transformations

~~Les widgets Flutter 4 et Transform possèdent déjà des constructeurs simples.~~

la classe Transform offre encore plus de fonctionnalités au développeur via son usine constructeurs. Il y en a beaucoup pour chacune des transformations possibles, ce qui en fait extrêmement facile à appliquer une transformation à un widget sans aucune connaissance approfondie de calculs géométriques. Ils sont les suivants:

Transform.rotate (): construit un widget Transform qui fait pivoter son enfant autour de son centre

Transform.scale (): construit un widget Transform qui met à l'échelle son enfant uniformément

Transform.translate (): Construit un widget Transform qui traduit son enfant par un décalage x, y

[415]

Épisode 434

Manipulations graphiques des widgets

Chapitre 14

Faire pivoter la transformation

La transformation de rotation apparaît dans les situations où nous voulons simplement faire notre le widget tourne. En utilisant le constructeur Transform.rotate (), nous pouvons obtenir des effets comme celui-ci:

La variante du constructeur Transofm.rotate peut être utilisée pour y parvenir, voyons donc de quoi il s'agit ressemble à:

```
Transform.rotate ({
  Clé clé,
  @ double angle requis,
  Origine du décalage,
  AlignmentGeometry alignment: Alignment.center,
  bool transformHitTests: true,
  Enfant du widget
})
```

[416]

Épisode 435

Manipulations graphiques des widgets

Chapitre 14

Comme vous pouvez le voir, il ne diffère pas trop du constructeur Transform par défaut. les différences sont les suivantes:

Absence de la propriété transform : nous utilisons la variante rotate () car nous voulons appliquer une rotation, nous n'avons donc pas besoin de spécifier l'ensemble matrice à cela. Nous utilisons simplement la propriété angle à la place.
Angle : Ceci spécifie la rotation souhaitée en radians dans le sens des aiguilles d'une montre.
Origine : par défaut, la rotation est appliquée par rapport au centre de l'enfant. Cependant, nous pouvons utiliser la propriété origin pour manipuler l'origine de rotation, comme si nous traduisions le centre du widget par le décalage d'origine, faire en sorte que la rotation soit relative à un autre point si nous le voulons.

Transformation d'échelle

La transformation d'échelle apparaît dans les situations où nous voulons simplement provoquer notre widget pour changer sa taille, soit en augmentant ou en diminuant son échelle. Nous pouvons obtenir quelque chose comme ce:

[417]

Épisode 436

Manipulations graphiques des widgets

Chapitre 14

Ce type de transformation est généralement effectué en utilisant le constructeur Transform.scale (). Voyons à quoi ça ressemble:

```
Transform.scale ({
    Clé clé,
    @ double échelle requise,
    Origine du décalage,
    AlignmentGeometry alignement: Alignment.center,
    bool transformHitTests: true,
    Enfant du widget
})
```

Comme vous pouvez le voir, tout comme le constructeur d'usine rotate (), cette variante ne diffère pas non plus

beaucoup de celui par défaut:

Absence de la propriété transform : Ici encore, nous utilisons la propriété scale au lieu de toute la matrice de transformation.

Echelle : C'est ce que nous utilisons pour spécifier l'échelle souhaitée au format double, 1.0 étant la taille d'origine du widget. Il représente le scalaire à appliquer à chaque x et axe y .

Alignement : par défaut, l'échelle est appliquée par rapport au centre de l'enfant.

Ici, nous pouvons utiliser la propriété d'alignement pour changer l'origine de l'échelle.

Encore une fois, nous pouvons combiner les propriétés d'alignement et d'origine pour obtenir le résultat.

Traduire la transformation

La transformation de traduction est plus susceptible d'apparaître dans les animations (voir [Chapitre](#)

[15, animations](#)). En utilisant le constructeur `Transform.translate()`, nous déplaçons le widget autour de l'écran:

[418]

Épisode 437

Manipulations graphiques des widgets

Chapitre 14

Et voici à quoi ressemble le constructeur d'usine `Transform.translate()`:

```
Transform.translate ({
  Clé clé,
  @ décalage de décalage requis,
  bool transformHitTests: true,
  Enfant du widget
})
```

[419]

Épisode 438

*Manipulations graphiques des widgets**Chapitre 14*

Ici, nous avons encore moins de propriétés par rapport aux transformations précédentes. le les différences sont les suivantes:

L'absence des propriétés de transformation et d'alignement : la transformation sera appliquée par la valeur de décalage, nous n'avons donc pas besoin de la matrice de transformation.

Offset : Cette fois, offset spécifie simplement la translation à appliquer sur le widget enfant; c'est différent des transformations précédentes, où cela affecte le point d'origine de la transformation appliquée.

Transformations composées

Nous pouvons, et nous le ferons très probablement, combiner un certain nombre des transformations vues précédemment pour obtenir des effets uniques, tels que la rotation en même temps que nous déplaçons et mettons à l'échelle un widget, comme dans l'exemple suivant:

[420]

Épisode 439

La composition des transformations peut être effectuée de deux manières:

- Utilisation du constructeur de widget Transform par défaut et génération de notre transformation en utilisant les méthodes fournies par Matrix4 pour la composer
- Utilisation de plusieurs widgets Transformer de manière imbriquée avec le rotate (), scale () et translate () constructeurs d'usine, obtenant le même effet, mais faire en sorte que notre arbre de widgets soit plus grand que nécessaire

Appliquer des transformations à vos widgets

Comme nous l'avons vu jusqu'à présent, le widget Transformer peut nous aider à modifier le naturel du widget les apparences. L'application de transformations à des widgets est aussi simple que l'ajout d'une transformation widget en tant que parent du widget que nous voulons modifier. Vérifions les alternatives que nous pouvons utiliser pour appliquer des transformations aux widgets.

Rotation des widgets

Comme indiqué précédemment, nous pouvons utiliser le constructeur Transform.rotate () pour ajouter un Transformez le widget en l'arborescence des widgets responsable de la rotation de son enfant. On peut utiliser quelque chose comme ça:

```
Transform.rotate(
    angle: -45 * (math.pi / 180.0),
    enfant: RaisedButton (
        enfant: Texte ("Bouton pivoté"),
        onPressed: () {},
    ),
);
```

Nous ajoutons un widget qui pivote de 315 ° dans le sens des aiguilles d'une montre (le même que -45 ° dans le sens inverse des aiguilles d'une montre). le même résultat est obtenu en utilisant le constructeur par défaut du widget Transform et un

Transformation Matrix4 à la place:

```
Transformer(
    transformée: Matrix4.rotationZ (-45 * (math.pi / 180.0)),
    alignment: Alignment.center,
    enfant: RaisedButton (
        enfant: Texte ("Bouton pivoté"),
        onPressed: () {},
    ),
);
```

[421]

Épisode 440

Les arguments que nous devons fournir pour obtenir le même résultat sont les suivants:

- transformer avec la rotation sur l' axe z
- alignement de la transformation

Mise à l'échelle des widgets

Pour mettre à l'échelle les widgets, nous utilisons le constructeur typique Transform.scale (). Pour faire évoluer un widget, par exemple, nous pouvons l'utiliser comme suit:

```
Transform.scale (
    échelle: 2.0,
    enfant: RaisedButton (
        enfant: texte ("agrandi"),
        onPressed: () {},
    ),
);
```

Et pour obtenir le même résultat en utilisant le constructeur Transform par défaut, nous utilisons ce qui suit:

```
Transformer(
    transform: Matrix4.identity () .. échelle (2.0, 2.0),
    alignment: Alignment.center,
    enfant: RaisedButton (
        enfant: texte ("agrandi"),
        onPressed: () {},
    ),
);
```

De manière très similaire à la rotation, il faut préciser à la fois l'origine de la transformation avec la propriété alignment et l'instance Matrix4 décrivant l'échelle transformation.

Traduire des widgets

De manière très similaire, nous utilisons le constructeur Transform.translate () en ajoutant un widget Transformer en tant que parent du widget que nous voulons déplacer:

```
Transform.translate (
    offset: Offset (100, 300),
    enfant: RaisedButton (
        enfant: texte ("traduit en bas"),
        onPressed: () {},
```

[422]

Épisode 441

Manipulations graphiques des widgets

Chapitre 14

```
    ),  
);
```

Le constructeur par défaut peut également être utilisé avec Matrix4 spécifiant la traduction:

```
Transformer(
    transform: Matrix4.translationValues (100, 300, 0),
    enfant: RaisedButton (
        enfant: texte ("traduit en bas"),
        onPressed: () {},  
),  
);
```

Il suffit de spécifier la propriété transform avec l'instance Matrix4 décrivant la Traduction.

Application de plusieurs transformations

Comme indiqué précédemment, nous avons deux façons d'ajouter plusieurs transformations aux widgets. La première consiste à ajouter plusieurs widgets Transform au-dessus du widget souhaité:

```
Transform.translate (
    offset: Offset (70, 200),
    enfant: Transform.rotate (
        angle: -45 * (math.pi / 180.0),
        enfant: Transform.scale (
            échelle: 2.0,
            enfant: RaisedButton (
                enfant: Texte ("transformations multiples"),
                onPressed: () {},  
),  
),  
),  
);
```

Comme vous pouvez le voir, nous ajoutons un widget Transform en tant qu'enfant à un autre widget Transform, composer la transformation. Bien que plus simple à lire, cette méthode présente un inconvénient: on ajoutez plus de widgets que nécessaire à l'arborescence des widgets.

Lorsque nous ajoutons plusieurs transformations à un widget en même temps, nous

doivent faire attention à l'ordre des transformations. Expérience par vous-même: échanger les positions des widgets Transform entraînera résultats différents.

[423]

Épisode 442

Manipulations graphiques des widgets

Chapitre 14

Comme alternative, nous pouvons utiliser le constructeur Transform par défaut avec le composé transformation avec l'objet Matrix4 à la place:

```
Transformer(
  alignment: Alignment.center,
  transformée: Matrix4.translationValues (70, 200, 0)
    ..rotateZ (-45 * (math.pi / 180.0))
    ..échelle (2.0, 2.0),
  enfant: RaisedButton (
    enfant: Texte ("transformations multiples"),
    onPressed: () {},
  ),
);
```

Tout comme avant, nous spécifions l'alignement de la transformation comme le centre de l'enfant widget, puis l'instance Matrix4 pour la décrire. Comme vous pouvez le voir, il est très similaire au plusieurs versions de widgets Transform mais sans widgets imbriqués provoquant un widget plus profond arbre.

Utilisation de peintres et de toiles personnalisés

Flutter vise à fournir les meilleurs outils possibles au développeur pour construire une application interfaces utilisateur sans limites. À présent, vous en êtes probablement déjà convaincu, avec les nombreux widgets qu'il fournit, la possibilité d'étendre ces widgets et le univers de possibilités qu'offre le framework.

La simplicité que Flutter apporte à la composition de l'interface utilisateur ne s'arrête pas aux widgets. Comment sur le changement de l'apparence du widget? Je ne parle pas d'étendre avec un widget Transform en le traduisant ou en le faisant tourner. Nous pouvons créer un widget avec sa propre apparence unique, son propre forme et ses propres comportements. Cela est possible à l'aide de trois classes principales:

CustomPaint, CustomPainter et Canvas.

La classe Canvas

Si vous avez déjà programmé une sorte d'interface utilisateur dans n'importe quelle langue, vous avez peut-être entendu ou travaillé avec une sorte de toile. Comme son nom l'indique, il permet de peindre des choses. La toile peut être considérée comme l' **espace sur lequel nous travaillons**, en dessinant des formes avec notre styles définis tels que les lignes, les cercles et les rectangles.

[424]

Épisode 443

Manipulations graphiques des widgets

Chapitre 14

Flutter Canvas ne fonctionne pas comme une toile littérale. Fondamentalement, c'est juste une interface pour enregistrement des opérations graphiques à dessiner sur l'image de rendu suivante.

Transformations du canevas

Toutes les opérations que nous effectuons sur Canvas, telles que dessiner une ligne ou un rectangle, sont orientées dans un système de coordonnées, comme tout autre système de dessin d'interface utilisateur. Ce système de coordonnées a un origine. Par défaut, cela est défini par le widget CustomPaint qui possède Canvas. une chose importante à noter est qu'en raison de cette caractéristique, toutes les opérations que nous faisons sur Canvas sont affectés par sa **transformation actuelle**. Chaque fois que nous voulons, nous pouvons transformer le canevas pour affecter les opérations suivantes.

Au départ, Canvas n'a pas de transformation, c'est-à-dire sa matrice de transformation sont une instance d'identité Matrix4.

Clip de toileRect

Comme les transformations, Canvas a une **région de clip actuelle**, ce qui signifie que nous pouvons couper partie de la toile à dessiner. Ceci est utile lorsque nous voulons simplement dessiner une partie d'un complexe forme sans trop se soucier des calculs.

Par défaut, la région de clip de Canvas est infinie, donc toutes les régions sont valide.

Méthodes

Comme indiqué précédemment, Canvas fonctionne en enregistrant les opérations de dessin sur la peinture suivante Cadre. Pour ce faire, il expose de nombreuses méthodes pour nous permettre de dessiner différentes formes. Allons examiner les plus courants:

- drawArc (): utilisé pour dessiner des arcs fermés ou des segments de cercle
- drawCircle (): Utilisé pour dessiner des cercles avec un rayon déterminé
- drawImage (): utilisé pour dessiner une image dans le canevas
- drawLine (): utilisé pour dessiner des lignes dans le canevas

[425]

Épisode 444

Manipulations graphiques des widgets

Chapitre 14

- drawRect (): utilisé pour dessiner des rectangles dans le canevas
- rotate (): ajoute une transformation de rotation à la transformation de canevas actuelle
- scale (): ajoute une transformation d'échelle à la transformation de canevas actuelle
- translate (): ajoute une traduction à la transformation de canevas actuelle

Consultez la documentation de la classe Canvas pour plus de méthodes et plus détails: <https://docs.flutter.dev/flutter/dart-ui/Canvas-class.html>

L'objet Paint

L'objet Paint est une description du style à utiliser lors du dessin sur Canvas. Cela nous permet définir des éléments tels que les couleurs et la largeur du trait. Toutes les méthodes de dessin de canevas récupèrent un Objet de peinture comme paramètre. Nous pouvons réutiliser la même instance de Paint sur plusieurs dessins appels.

Le widget CustomPaint

L'objet Canvas n'est disponible nulle part dans Flutter; cela peut prêter à confusion. N'importe quand

nous voulons dessiner les choses à la main, nous devons utiliser le widget CustomPaint. Le principal but de ce widget est de fournir un objet Canvas sur lequel travailler.

Avoir un canevas et un widget CustomPaint ne suffit pas pour dessiner. le but de CustomPaint est de fournir Canvas et de déléguer un objet CustomPainter qui sera responsable de dessiner dessus.

Détails de construction CustomPaint

Le widget CustomPaint fonctionne simplement comme le pont entre l'arborescence des widgets (en étant un widget) et un calque de peinture de niveau inférieur avec accès à Canvas. Pour créer cela, nous devons avoir une instance CustomPainter car cela n'a pas de sens d'avoir CustomPaint sans peintre.

[426]

Épisode 445

Manipulations graphiques des widgets

Chapitre 14

Pour créer un widget CustomPaint, nous l'ajoutons d'abord à notre arborescence de widgets comme nous le faisons pour les autres widgets. Jetons d'abord un coup d'œil à son constructeur pour le comprendre:

```
const CustomPaint ({  
    Clé clé,  
    Peintre sur mesure,  
    CustomPainter avant-planPainter,  
    Taille taille: Size.zero,  
    booléen isComplex: false,  
    bool willChange: false,  
    Enfant du widget  
})
```

Il y a quelques propriétés que nous devons examiner pour comprendre comment cela fonctionne:

- peintre: l'implémentation du peintre qui dessine le contenu sur la toile
- foregroundPainter: l'implémentation de peintre qui dessine le contenu sur le toile après que l'enfant est peint
- size: Si la propriété enfant n'est pas nulle, la taille de l'enfant est utilisée et cette valeur est ignoré; sinon, cela spécifie la taille nécessaire pour le tirage
- isComplex et willChange: conseils sur le cache raster du compositeur, avec l'analyse des coûts d'équarrissage
- enfant: un enfant en dessous dans l'arborescence des widgets, comme tout autre widget

Nous pouvons voir les propriétés liées au peintre dans la capture d'écran suivante:

Ceci illustre l'ordre du dessin: d'abord, les opérations du peintre sont faites, puis enfant, et enfin, foregroundPainter (le cas échéant) dessine devant l'enfant.

Épisode 446

Manipulations graphiques des widgets

Chapitre 14

L'objet CustomPainter

Nous connaissons l'importance de l'objet CustomPainter (ou peintre). Comme indiqué auparavant, le peintre est responsable de dessiner quelque chose sur toile. Chaque fois que nous souhaitons créer notre propre logique de dessin unique, nous devons étendre la classe CustomPainter et remplacer deux méthodes fondamentales: paint () et shouldRepaint () .

La méthode de peinture

La méthode paint () est l'endroit où le CustomPainter fait son travail. Il est appelé chaque fois que le widget doit être redessiné. Voici à quoi cela ressemble:

```
peinture vide (
    Toile de toile,
    Taille taille
)
```

Les deux seuls arguments qu'il reçoit sont les suivants:

```
canvas, où nous dessinons efficacement en utilisant ses méthodes draw * ()
la taille définit les limites du dessin, que nous devrions considérer
```

Les opérations de peinture doivent rester à l'intérieur de la zone donnée. Voici ce que dit la documentation:

"Les opérations graphiques en dehors des limites peuvent être silencieusement ignorées, tronquées ou non tronquées."

La méthode shouldRepaint

C'est une méthode importante, en particulier pour le moteur Flutter. Voici à quoi cela ressemble:

```
bool shouldRepaint (
    covariante CustomPainter oldDelegate
)
```

Il ne reçoit que l'argument oldDelegate, qui correspond au dernier délégué (cette instance de classe CustomPainter) qui était responsable de la peinture sur CustomPaint. Chaque fois qu'il retourne false, l'appel de peinture peut être optimisé loin (cela ne signifie pas que la peinture ne sera pas appellée). Nous devrions comparer l'ancien et délégué actuel pour voir si les données relatives à la peinture sont différentes, puis retourner vrai dans ce cas.

Épisode 447

Manipulations graphiques des widgets

Chapitre 14

Un exemple pratique

Il est temps de voir comment nous pouvons utiliser les widgets Canvas et CustomPaint pour créer un widget qui a sa propre peinture. Dans cet exemple, nous allons créer des widgets de graphique - un secteur et carte radiale, pour être plus précis. Les camemberts sont un type utile de graphique statistique circulaire, qui est divisé en tranches pour illustrer les proportions numériques.

Nous allons commencer par le widget de graphique à secteurs, où nous récupérons les valeurs des tranches et dessinons les proportionnellement dans un cercle. Voici à quoi cela va ressembler:

Maintenant, définissons le nouveau widget PieChart à l'aide du Canvas et Classes CustomPaint.

[429]

Épisode 448

Manipulations graphiques des widgets

Chapitre 14

Définition d'un widget

Pour commencer, nous définissons généralement un widget, pour maintenir un niveau d'organisation minimal. nous définissons le widget PieChart ici; ce sera un descendant StatelessWidget. Ce widget doit faire abstraction de la couche de peinture et exposer exactement ce dont les autres widgets ont besoin. Dans notre cas, voici à quoi ressemblent les propriétés de PieChart:

```
La classe PieChart étend StatelessWidget {
    valeurs finales de List <int>;
    couleurs de la liste finale <Couleur>;
    ...
}
```

Les seules propriétés qui décrivent le widget sont les valeurs et les couleurs:

values: Cette liste représente chacune des valeurs de section. Ici, nous utilisons des valeurs int pour plus de simplicité, mais il peut s'agir de n'importe quel type pour fonctionner avec n'importe quelle logique.
couleurs: cette liste contient les couleurs qui doivent être utilisées pour peindre chacun des sections du graphique.

Maintenant, jetons un œil à la méthode build () de ce widget:

```
@passer autre
Construction du widget (contexte BuildContext) {
    return Row (
        enfants: <Widget> [
            Étendu(
                enfant: CustomPaint (
                    peintre: PieChartPainter (
                        valeurs,
```

```

    ), couleurs
    ),
    ],
);
}

```

[430]

Épisode 449

Manipulations graphiques des widgets

Chapitre 14

Il y a quelques points auxquels nous devons prêter attention ici:

Le widget CustomPaint a besoin d'une taille pour exister, comme toute sa logique de peinture devrait être fait dans une toile finie. Comme nous l'avons vu précédemment, le widget CustomPaint définit sa taille par ses contraintes enfants. Dans notre cas, nous n'avons pas d'enfant, donc il doit être limité d'une manière ou d'une autre. Nous aurions pu limiter sa taille en utilisant SizedBox, par exemple, mais ce ne serait pas idéal. Au lieu de cela, nous l'avons mis à l'intérieur un widget Row, remplissant son espace horizontal disponible en l'entourant d'un

Widget étendu.

Le widget CustomPaint emmène notre peintre personnalisé, PieChartPainter, à travers la propriété du peintre.

C'est tout ce dont le widget a besoin, car le travail acharné sera effectué par le Classe PieChartPainter.

Définition de CustomPainter

La définition de notre classe descendante CustomPainter est l'étape la plus importante ici. Comme dit avant, dans cet exemple, nous avons défini un peintre qui prend une liste de valeurs int et, basé sur cela, dessine une tarte comme un cercle avec des tranches proportionnelles.

Comme indiqué précédemment, nous devons remplacer deux méthodes de CustomPainter pour que cela fonctionne. Voyons comment nous les définissons.

Remplacer la méthode shouldRepaint

Dans notre exemple, les valeurs et les couleurs décrivent le dessin, donc chaque fois que l'une d'elles changer, nous devons repeindre notre widget. Donc, nous devons refléter cela dans le shouldRepaint méthode, comme suit:

```

// fait partie du fichier pie_chart.dart Classe PieChartPainter

@passer outre
boolen shouldRepaint (PieChartPainter oldDelegate) {
    return! ListEquality (). equals (oldDelegate.values, values) ||
        ! ListEquality (). Equals (oldDelegate.colors, couleurs);
}

```

[431]

Épisode 450

Manipulations graphiques des widgets

Chapitre 14

Remplacer la méthode de peinture

La méthode de peinture est responsable du dessin de notre graphique. Voici comment nous le définissons:

```
// fait partie du fichier pie_chart.dart Classe PieChartPainter

@passer autre
peinture vide (toile, taille de la taille) {
    var center = Offset (taille.largeur / 2, taille.hauteur / 2);
    var rayon = (taille.largeur * 0,75) / 2;

    Rect chartRect = Rect.fromCircle (
        centre: centre,
        rayon: rayon,
    );

    int total = valeurs.reduce ((a, b) => a + b);

    _paintCircle (canevas, total, chartRect);
}
```

Décomposons-le:

1. Tout d'abord, nous devons définir nos extensions de secteurs. Avec le paramètre de taille donné, nous peut définir le centre de notre graphique et le rayon, qui est la moitié de 75 pour cent de la espace disponible, (var radius = (size.width * 0,75) / 2;), à préserver un peu d'espace autour du graphique.
2. Ensuite, nous créons une instance Rect à partir des propriétés de centre et de rayon données. Ce rectangle sera utile lorsque nous dessinerons les arcs de chaque tranche (voir le _paintCircle explication de la méthode plus tard).
3. La valeur totale que nous obtenons en additionnant toutes les valeurs de la tranche donnée. Ce sera également utile lorsque nous dessinons chacun des arcs de tranche.
4. Enfin, nous pouvons dessiner le graphique à secteurs sur la toile.

La méthode _paintCircle () est initialement définie comme suit:

```
void _paintCircle (Canvas canvas, int total, Rect chartRect) {
    Paint sectionPaint = Paint () .. style = PaintingStyle.fill;

    double startAngle = -90;
    pour (var i = 0; i < values.length; i++) {
        valeur finale = valeurs [i];
        couleur finale = couleurs [i];

        double sweepAngle = ((valeur * 360,0) / total);
```

[432]

Épisode 451

Manipulations graphiques des widgets

Chapitre 14

```
sectionPaint.color = couleur;
canvas.drawArc (
    chartRect,
    startAngle * _toRadians,
    sweepAngle * _toRadians,
    vrai,
    sectionPeinture,
);

startAngle += sweepAngle;
}
```

Les sections du graphique sont dessinées séquentiellement. Nous devons savoir sous quel angle commencer un tranche et où il va, car un cercle a des angles de balayage de 360 °. La méthode drawArc

from Canvas est basé sur la spécification d'un angle de départ pour l'arc et son balayage correspondant angle. Nous pouvons obtenir chacun des angles de balayage de la tranche en utilisant une **règle simple de trois** calculs basés sur le total précédemment calculé.

La règle de trois est une règle mathématique qui nous permet de résoudre directement et problèmes de proportion inverse.

Dans cet esprit, voyons comment nous dessinons chacun des arcs de tranche et formons le graphique entier:

1. Tout d'abord, nous définissons startAngle. Nous avons choisi -90° pour commencer à dessiner les arcs. Utilisant un Analogie du cadran d'horloge, cela équivaut à 12 heures. Si nous avons choisi 0° pour commencer, ce serait équivalent à 3 heures ou, comme indiqué dans la méthode drawArc Documentation:

"... zéro radian étant le point sur le côté droit de l'ovale qui traverse le ligne horizontale qui coupe le centre du rectangle et avec positif angles allant dans le sens des aiguilles d'une montre autour de l'ovale. "

[433]

Épisode 452

Manipulations graphiques des widgets

Chapitre 14

2. Enfin, nous passons par chacune des valeurs pour dessiner chacun des arcs:
 1. Tout d'abord, nous calculons l'angle de balayage de l'arc, qui n'est rien de plus que le angle d'un arc dans le cercle donné. Comme dit précédemment, ceci est obtenu à partir de une règle simple de trois, où nous nous interrogeons: si la valeur totale est équivalente à un angle de balayage de 360° , quel est l'angle de balayage (combien degrés) de la valeur de tranche actuelle?
 2. Ensuite, nous définissons la couleur de notre objet Paint sur la couleur de la tranche actuelle. Notre objet Paint défini précédemment a le style PaintingStyle.fill, ce qui signifie que la forme dessinée avec cette peinture sera remplie avec la couleur donnée. Dans notre cas, c'est exactement ce que nous voulons.
 3. Enfin, nous dessinons un arc qui commence à la propriété startAngle donnée et a sweepAngle (consultez l'explication suivante).

Voyons comment nous pouvons utiliser la méthode drawArc de la classe Canvas pour dessiner nos tranches.

Voici à quoi ressemble la méthode drawArc:

```
void drawArc (
    Rect rect,
    double startAngle,
    double balayageAngle,
    bool useCenter,
    Peinture peinture
)
```

Jetons un coup d'œil aux valeurs que nous avons passées à cette fonction:

rect est utilisé pour guider le tirage au sort. L'arc sera dessiné à l'intérieur du rectangle donné, avec son angle de balayage par rapport au centre du rectangle.

startAngle définit où commencer à dessiner l'arc. Rappelez-vous, 0° est équivalent à 3 heures.

sweepAngle définit combien l'arc prend de l'ovale. Nous calculons cela avec le total et chacune des valeurs de tranche.

useCenter nous aide à manipuler la façon dont l'arc est dessiné, comme mentionné dans le

Documentation:

"Si c'est vrai, l'arc est refermé vers le centre, formant un secteur de cercle.
Sinon, l'arc n'est pas fermé, formant un segment de cercle. "

Paint définit comment notre arc est dessiné. Ici, nous faisons l'arc rempli de le style PaintingStyle.fill et définissez sa couleur avec la couleur de trame donnée.

[434]

Épisode 453

Manipulations graphiques des widgets

Chapitre 14

Comme vous pouvez le voir, nous devons convertir les valeurs d'angle en radians avant les envoyer à la fonction drawArc.

La variante du diagramme radial

Pour vous aider à comprendre le potentiel de l'utilisation des widgets CustomPaint, créons un autre widget, cette fois pour dessiner un graphique radial, comme ceci:

La représentation radiale est très similaire au graphique à secteurs; la seule différence est qu'il a un étiquette en son centre indiquant le total des valeurs.

[435]

Épisode 454

Manipulations graphiques des widgets

Chapitre 14

Définition d'un widget

Le widget RadialChart est très similaire au widget PieChart défini précédemment, avec le mêmes paramètres et même objectif fondamental. La seule chose dont nous avons besoin pour prendre un regarder ici est sa méthode build ():

```
// fait partie du widget radial_chart.dart RadialChart

@passer autre
Construction du widget (contexte BuildContext) {
    return Row (
        enfants: <Widget> [
            Étendu(
                enfant: CustomPaint (
                    peintre: RadialChartPainter (
                        valeurs,
                        couleurs,
                        Theme.of(context).textTheme.display1,
                        Directionnalité. De (contexte),
                    ),
                    ),
                    ),
                ],
            );
    }
}
```

Comme vous pouvez le voir, la différence réside dans la valeur transmise à la propriété painter de le widget CustomPaint. Ici, nous utilisons une nouvelle classe RadialChartPainter qui a sa propre implémentation paint (). Outre les valeurs et les couleurs, nous passons deux autres ses paramètres:

TextStyle, qui sera utilisé pour dessiner l'étiquette de valeur totale
Une instance TextDirection nécessaire pour dessiner des textes dans la bonne orientation

Définition de CustomPainter

La classe RadialChartPainter, comme le widget RadialChart, diffère par des parts de PieChartPainter, qui a été défini auparavant. À première vue, sa peinture ()
La méthode est presque la même que dans le graphique à secteurs:

```
// fait partie de la classe radial_chart.dart RadialChartPainter

@passer autre
peinture vide (toile, taille de la taille) {
    var center = Offset (taille.largeur / 2, taille.hauteur / 2);
```

[436]

Épisode 455

Manipulations graphiques des widgets

Chapitre 14

```
var rayon = taille.largeur * 0,75 / 2;

Rect chartRect = Rect.fromCircle (
    centre: centre,
    rayon: rayon,
);

int total = valeurs.reduce ((a, b) => a + b);

_paintTotal (canevas, total, chartRect);
_paintCircle (canevas, total, chartRect);
}
```

Comme vous pouvez le voir, la seule différence est l'appel supplémentaire à _paintTotal (canvas,
total, chartRect);

Avant de vérifier cette nouvelle méthode, voyons d'abord quels changements dans le _paintCircle ()
méthode:

```
// fait partie de la classe radial_chart.dart RadialChartPainter

void _paintCircle (Canvas canvas, int total, Rect chartRect) {
```

```

Paint sectionPaint = Paint()
    .style = PaintingStyle.stroke
    .strokeWidth = 30.0;

double startAngle = -90;
pour (var i = 0; i < values.length; i++) {
    valeur finale = valeurs [i];
    couleur finale = couleurs [i];

    double sweepAngle = ((valeur * 360,0) / total);

    sectionPaint.color = couleur;
    canvas.drawArc (
        chartRect,
        (startAngle + 2) * _toRadians,
        (sweepAngle - 2) * _toRadians,
        faux,
        sectionPeinture,
    );
    startAngle += sweepAngle;
}
}

```

[437]

Épisode 456

Manipulations graphiques des widgets

Chapitre 14

Comme vous pouvez le voir, presque tout est pareil, avec juste quelques points à noter:

Nous avons changé notre sectionPaint style en PaintingStyle.stroke; ce façon, la forme dessinée avec cette peinture ne sera pas remplie - à la place, elle n'aura que son contour dessiné. C'est pourquoi nous définissons également la propriété strokeWidth.
Comme vous l'avez peut-être noté, avant d'envoyer les valeurs d'angle à drawArc fonction, nous ajoutons d'abord 2 ° au startAngle et soustrayons 2 ° du sweepAngle valeur, laissant un peu d'espace entre les tranches pour avoir un meilleur résultat visuel.
Enfin, nous passons false au paramètre useCenter pour ne pas former un cercle rempli, mais un segment d'arc.

C'est tout ce que nous avons changé pour obtenir un graphique radial comme celui-ci:

[438]

Épisode 457

Manipulations graphiques des widgets

Chapitre 14

Enfin, en regardant la peinture du texte, nous avons la méthode `_paintTotal()`:

```
void _paintTotal(Canvas canvas, int total, Rect chartRect) {
    totalPainter final = TextPainter (
        maxLines: 1,
        text: TextSpan (
            style: textStyle,
            text: "$ total",
        ),
        textDirection: textDirection,
    );

    totalPainter.layout (maxWidth: chartRect.width);
    totalPainter.paint (
        Toile,
        chartRect.center.translate (
            -totalPainter.width / 2.0,
            -totalPainter.height / 2.0,
        ),
    );
}
```

Pour dessiner un texte dans le canevas, nous allons suivre ces étapes:

1. Tout d'abord, nous instancions un objet `TextPainter`, qui définit à quoi ressemblera un texte une fois dessiné, tout comme la classe `Paint` le fait pour les formes. Dans notre cas, nous le définissons comme être une **seule ligne** et avoir son style et `textDirection` récupérés à partir du `Widget RadialChart`.
2. Ensuite, nous nous assurons d'appeler la fonction `layout()` à partir du `TextPainter` exemple. Cet appel calculera la position visuelle des glyphes pour peindre le texte.
3. Avec la taille de texte connue, nous pouvons le positionner correctement dans la dernière étape. À positionner le texte exactement au centre du graphique, nous traduisons simplement le centre du rectangle du graphique par la moitié de la taille du texte.

C'est tout pour notre widget `CustomPaint`. Comme vous l'avez peut-être remarqué, nos graphiques semblent très similaires les uns aux autres. La plus grande différence réside dans le peintre défini. Nous pouvons résumer ces en un seul widget, où nous pouvons récupérer le type de graphique souhaité et changer simplement le peintre que nous envoyons au widget `CustomPaint`.

[439]

Épisode 458

Manipulations graphiques des widgets

Chapitre 14

Sommaire

Dans ce chapitre, nous avons appris à changer l'apparence de nos widgets en utilisant le Classe de transformation et ses transformations disponibles, telles que la **mise à l'échelle**, la **traduction** et

tournant. Nous avons également vu comment nous pouvons composer la transformation en utilisant la classe Matrix4 directement.

Nous avons appris comment la classe Canvas peut être utilisée pour prendre le contrôle des widgets dessinés et comment nous pouvons l'utiliser pour créer nos propres peintures.

Enfin, nous avons vu comment le widget CustomPaint peut être utile pour créer nos propres widgets qui ont non seulement des fonctionnalités uniques, mais aussi des apparences uniques définies par un descendant de CustomPainter.

Dans le dernier chapitre, nous verrons comment animer des widgets, en utilisant les transformations apprises ici.

[440]

Épisode 459

15 Animations

Les animations Flutter intégrées peuvent être combinées et étendues pour satisfaire les besoins des développeurs en l'UX. Dans ce chapitre, vous en apprendrez beaucoup plus sur les animations, en utilisant Tween animations pour gérer une chronologie et une courbe d'animation, et en utilisant AnimatedBuilder et AnimatedWidget pour ajouter et combiner de belles animations.

Les sujets suivants seront traités dans ce chapitre:

- Apprendre à connaître les bases des animations
- Utiliser des animations
- Utilisation d'AnimatedBuilder
- Utilisation d'AnimatedWidget

Présentation des animations

Dans Flutter, les animations sont largement prises en charge et le framework offre plusieurs façons de-animer des widgets. En outre, il existe des animations intégrées prêtes à l'emploi dont nous n'avons besoin que-branchez des widgets pour les animer. Bien que Flutter résume bon nombre des-complexités qu'impliquent les animations, il y a quelques concepts importants que nous devons-comprendre avant de plonger dans le sujet des animations.

La classe Animation <T>

Dans Flutter, les animations se composent d'un statut et d'une valeur avec le type T. Le statut de l'animation-correspond à son état (c'est-à-dire s'il est en cours d'exécution ou terminé); sa valeur correspond à son-valeur actuelle, et elle est destinée à changer pendant l'exécution de l'animation.

Épisode 460

Animations

Chapitre 15

En plus de contenir ces informations sur l'animation, cette classe expose des rappels afin que d'autres-les classes peuvent savoir comment les animations s'exécutent, l'état actuel de l'animation et-valeur aussi.

Une instance de classe Animation <T> est uniquement responsable de la conservation et de l'exposition de ces valeurs. Il ne sait rien sur le retour visuel, ce qui est dessiné à l'écran, ou comment dessiner-it (c'est-à-dire les fonctions build ()).

L'un des types d'animation les plus courants que vous verrez est le type Animation <double>-représentation, car la valeur double peut facilement être utilisée pour manipuler tout type de valeurs dans un-sens de l'espace proportionnel.

La classe Animation génère une séquence (pas nécessairement linéaire) de valeurs entre-valeurs minimales et maximales déterminées. Ce processus est également connu sous le nom de-interpolation et, comme dit précédemment, cette interpolation n'est pas seulement linéaire - elle peut être définie comme un-fonction d'étape ou une courbe. Flutter fournit de multiples fonctions et installations pour le fonctionnement-animations. Ils sont les suivants:

AnimationController: Malgré ce que son nom suggère, il n'est pas utilisé pour contrôler-objets d'animation, mais aide dans la tâche de contrôle de lui-même, car il s'étend-la classe Animation et est toujours une animation.

CurvedAnimation: il s'agit d'une animation qui applique Curve à un autre-animation.

Tween: Cela permet de créer une interpolation linéaire entre un début et-valeur finale.

La classe Animation expose des moyens d'accéder à son état et à sa valeur pendant un cycle en cours d'exécution. Grâce aux **écouteurs de statut**, nous pouvons savoir quand une animation commence, se termine ou se termine-direction inverse. En utilisant sa méthode addStatusListener (), on peut, par exemple, manipulez nos widgets en réponse aux événements de début ou de fin d'animation. De la même manière, nous-peut ajouter des écouteurs de valeur avec la méthode addListener () afin que nous soyons avertis à chaque fois que le-la valeur de l'animation change et nous pouvons reconstruire nos widgets en utilisant setState () {}.

Épisode 461

*Animations**Chapitre 15*

AnimationContrôleur

`AnimationController` est l'une des classes d'animation Flutter les plus utilisées. Il est dérivé de la classe `Animation<double>` et ajoute quelques méthodes fondamentales pour manipuler les animations. La classe `Animation` est la base de l'animation dans Flutter; comme dit précédemment, il fait n'ont pas de méthodes liées au contrôle d'animation. `AnimationController` ajoute ces contrôles au concept d'animation, tels que les suivants:

- Commandes de lecture et d'arrêt :** `AnimationController` ajoute la possibilité de lire l'animation avant, arrière ou arrêtez-la
- Durée :** les animations réelles ont un temps limité pour jouer, c'est-à-dire qu'elles jouent pendant un certain temps et terminer, ou répéter
- Permet de définir la valeur actuelle de l'animation :** cela provoque un arrêt de l'animation et notifie le statut et la valeur aux auditeurs
- Permet de définir la limite supérieure et inférieure de l'animation :** c'est pour que nous pouvons connaître les valeurs estimées avant et après la lecture de l'animation

Vérifions le constructeur `AnimationController` et analysons ses principales propriétés:

```
AnimationController ({
    double valeur,
    Durée durée,
    Chaîne debugLabel,
    double limite inférieure: 0,0,
    double limite supérieure: 1,0,
    AnimationBehavior animationBehavior: AnimationBehavior.normal,
    @required TickerProvider vsync
})
```

Comme vous pouvez le voir, certaines propriétés sont explicites, mais passons en revue:

- value:** il s'agit de la valeur initiale de l'animation et la valeur par défaut est `lowerBound` si non précisé.
- duration:** Il s'agit de la durée de l'animation.
- debugLabel:** il s'agit d'une chaîne pour vous aider lors du débogage. Il identifie le contrôleur dans la sortie de débogage.
- lowerBound:** cela ne peut pas être nul; c'est la plus petite valeur de l'animation dans qu'il est considéré comme rejeté, généralement la valeur de départ lors de l'exécution.
- upperBound:** De plus, cela ne peut pas être nul; c'est la plus grande valeur de l'animation à qu'il est considéré comme complet, généralement la valeur finale lors de l'exécution.

[443]

Épisode 462

*Animations**Chapitre 15*

`animationBehavior:` Ceci configure le comportement d'`AnimationController` lorsque les animations sont désactivées. S'il s'agit de `AnimationBehavior.normal`, la durée de l'animation sera réduite et si c'est `AnimationBehavior.preserve`, `AnimationController` conservera son comportement.

`vsync:` Il s'agit d'une instance `TickerProvider` que le contrôleur utilisera pour obtenir un signal chaque fois qu'une trame se déclenche.

Vérifiez toutes les méthodes disponibles pour exécuter des animations avec le

TickerProvider et Ticker

L'interface TickerProvider décrit des objets capables de fournir des objets Ticker.

Les tickers sont utilisés par toute classe qui a besoin de savoir quand la prochaine image va être construite. Ils sont couramment utilisés indirectement via AnimationControllers. Lors de l'utilisation de l'État classe, nous pouvons prolonger avec

`TickerProviderStateMixin` ou `SingleTickerProviderStateMixin` à avoir `TickerProvider` et l'utiliser avec les objets `AnimationController`.

CourbéAnimation

La classe `CurvedAnimation` est utilisée pour définir la progression d'une classe `Animation` en tant que courbe non linéaire. Nous pouvons l'utiliser pour modifier une animation existante en changeant son méthode d'interpolation. Il est également utile lorsque nous voulons utiliser une courbe différente lors de la lecture une animation en avant puis en marche arrière, en utilisant sa courbe et sa courbe inverse propriétés respectivement.

La classe `Curves` définit de nombreuses courbes prêtes à être utilisées dans notre animation plutôt que `Curves.linear`.

[444]

Épisode 463

Animations

Chapitre 15

Consultez la page de documentation `Curves` pour voir, en détail, comment chacun des les courbes se comporte: <https://api.flutter.dev/flutter/dart-ui/Curves-class.html>

Tween

Outre toutes ces classes, nous en avons une qui peut vous aider dans des tâches spécifiques concernant la gamme de l'animation. Comme nous l'avons vu, par défaut, les valeurs simples de début et de fin de l'animation sont respectivement de 0,0 et 1,0. Nous pouvons, en utilisant Tweens, modifier la plage ou le type de `AnimationController` sans le modifier. Les préadolescents peuvent être de n'importe quel type, et nous pouvons également créer notre classe Tween personnalisée si vous le souhaitez. Le fait est que Tweens renvoie des valeurs à des périodes entre le début et la fin, que vous pouvez passer comme accessoires à tout ce que vous animerez, donc il est toujours mis à jour; par exemple, nous pouvons changer la taille d'un widget, position, opacité, couleur, etc. en utilisant des Tweens spécifiques pour chacun.

Nous avons également d'autres classes descendantes de Tween telles que la classe `CurveTween` disponibles afin que nous pouvons modifier une courbe d'animation, ou `ColorTween`, qui crée une interpolation entre les couleurs.

Utiliser des animations

Lorsque vous travaillez avec des animations, nous n'allons pas toujours créer exactement la même chose objets d'animation, mais nous pouvons trouver des similitudes dans les cas d'utilisation. Les objets Tween sont utiles pour changer le type et la plage d'une animation. Nous serons, la plupart du temps, en train de composer animations avec les instances `AnimationController`, `CurvedAnimation` et `Tween`.

Avant d'utiliser une implémentation Tween personnalisée, revisitons nos transformations de widgets du dernier chapitre en appliquant la transformation de manière animée. Nous obtiendrons le même effet final mais d'une manière douce et meilleure.

[445]

Épisode 464

Animations

Chapitre 15

Faire pivoter l'animation

Au lieu de changer directement la rotation des boutons, nous pouvons plutôt la rendre progressive en en utilisant la classe `AnimationController`:

Consultez l'exemple `hands_on_animations` sur GitHub pour le exemples complets.

Dans cet exemple, nous créons notre widget d'une manière très similaire à celle d'avant (au [chapitre 14](#), *Manipulations graphiques du widget*):

```
_rotationAnimationButton () {
    retourne Transform.rotate (
        angle: _angle,
        enfant: RaisedButton (
            enfant: Texte ("Bouton pivoté"),
            onPressed: () {
                if (_animation.status == AnimationStatus.completed) {
                    _animation.reset ();
                    _animation.forward ();
                }
            },
        ),
    );
}
```

[446]

Épisode 465

*Animations**Chapitre 15*

Comme vous pouvez le voir, il y a deux choses importantes à noter:

- La valeur de l'angle est maintenant définie avec une propriété `_angle` au lieu de directement affectation à un littéral
- Dans la propriété `onPressed`, nous vérifions si `_animation` est terminée, et si ça l'est, on le répète depuis le début

Voyons maintenant comment se déroule la partie animation. Donc, nous devons savoir comment créer notre objet `AnimationController` et faites-le fonctionner. Jetons un coup d'œil à notre exemple de classe première:

```
La classe _RotationAnimationsState étend State <RotationAnimations> avec
SingleTickerProviderStateMixin {
  double _angle = 0.0;
  AnimationController _animation;
  ...
}
```

Quelques points sont importants à noter dans cette classe:

- Nous avons un objet `StatefulWidget` appelé `RotationAnimations`, à utiliser la classe `SingleTickerProviderStateMixin` que nous avons vu précédemment et fournissez l'objet `Ticker` requis pour que notre contrôleur s'exécute.
- En plus de cela, nous avons la propriété `_angle`, utilisée pour définir le courant de notre bouton angle. Nous pouvons utiliser la méthode `setState()` pour la faire construire avec un nouveau angle.
- Et enfin, nous avons notre objet `_animation`, pour tenir une animation et nous permettre pour le gérer.

La fonction `initState()` de notre classe `State` est l'endroit idéal pour configurer le animation et démarrez-le:

```
@passer autre
void initState () {
  super.initState ();

  _animation = createRotationAnimation ();
  _animation.forward ();
}
```

[447]

Épisode 466

*Animations**Chapitre 15*

Comme vous pouvez le voir, nous définissons notre animation via `createRotationAnimation()` et faites-la fonctionner en appelant sa fonction `forward()`. Voyons maintenant comment le l'animation est définie:

```
createRotationAnimation () {
  var animation = AnimationController (
    vsync: ceci,
    debugLabel: "démonstration d'animations",
    durée: Durée (secondes: 3),
  );

  animation.addListener (() {
    setState (() {
      _angle = (animation.value * 360.0) * _toRadians;
    });
  });
}
```

```
    retourne l'animation;
}
```

Nous pouvons diviser la création de l'animation en deux parties importantes:

Il y a la définition de l'animation elle-même, où nous définissons l'animation propriété debugLabel à des fins de débogage; le vsync, de sorte qu'il puisse avoir un Ticker et savoir quand produire une nouvelle valeur d'animation; et enfin, la durée de l'animation.

La deuxième étape importante consiste à écouter les changements de valeur d'animation. Ici, chaque fois que l'animation a une nouvelle valeur, on l'obtient et on la multiplie par 360 degrés, de sorte que nous obtenons une valeur de rotation proportionnelle.

Comme vous pouvez le voir, nous pouvons générer les valeurs souhaitées en fonction de valeurs d'animation doubles, donc, la plupart du temps, l'animation <double> suffira à jouer avec les animations.

[448]

Épisode 467

Animations

Chapitre 15

Si nous le voulions, nous pourrions ajouter une courbe différente à l'animation en utilisant CurveTween, pour exemple, comme vous pouvez le voir dans la méthode createBounceInRotationAnimation ():

```
createBounceInRotationAnimation () {
    var controller = AnimationController (
        vsync: ceci,
        debugLabel: "démonstration d'animations",
        durée: Durée (secondes: 3),
    );

    var animation = controller.drive (CurveTween (
        courbe: Curves.bounceIn,
    ));

    animation.addListener (() {
        setState (() {
            _angle = (animation.value * 360.0) * _toRadians;
        });
    });

    contrôle de retour;
}
```

Ici, nous créons une autre instance d'Animation en utilisant la méthode drive () du contrôleur et en passant la courbe souhaitée avec un objet CurveTween. Notez que nous avons ajouté des auditeurs à le nouvel objet d'animation au lieu du contrôleur, car nous voulons des valeurs relatives à la courbe.

Un point important à noter est que nous devons disposer de notre AnimationController classe à la fin de la vie de notre classe State pour éviter les fuites:

```
@passer outre
void dispose () {
    _animation.dispose ();
    super.dispose ();
}
```

Cela doit être fait pour chaque type d'animation que nous faisons, car nous travaillerons toujours avec AnimationController.

Voyons maintenant comment créer des animations à l'échelle.

[449]

Épisode 468

Animations

Chapitre 15

Animation d'échelle

Pour créer une animation d'échelle et avoir un meilleur effet que de changer l'attribut d'échelle directement, nous pouvons, encore une fois, utiliser la classe AnimationController:

Cette fois, pour construire notre widget RaisedButton avec une échelle, nous définissons un widget Transform avec le constructeur bien connu Transform.scale:

```
_scaleAnimationButton () {
    retourne Transform.scale (
        échelle: _scale,
        enfant: RaisedButton (
            enfant: Texte ("Bouton mis à l'échelle"),
            onPressed: () {
                if (_animation.status == AnimationStatus.completed) {
                    _animation.reverse ();
                } else if (_animation.status == AnimationStatus.dismissed) {
                    _animation.forward ();
                }
            },
        ),
    );
}
```

[450]

Épisode 469

Animations

Chapitre 15

Notez que, maintenant, nous utilisons une propriété `_scale` en place et examinons le changement de https://translate.googleusercontent.com/translate_f

la méthode onPressed. Ici, nous jouons l'animation en mode inverse en utilisant la fonction reverse () d'AnimationController si elle est terminée, et lire en avant si elle est à son état initial (c'est-à-dire après l'avoir inversé).

La création d'un objet d'animation se produit d'une manière très similaire à une animation de rotation, mais there are slight modifications to the controller construction:

```
createScaleAnimation() {
    var animation = AnimationController (
        vsync: ceci,
        lowerBound: 1.0,
        upperBound: 2.0,
        debugLabel: "démonstration d'animations",
        duration: Duration(seconds: 2),
    );

    animation.addListener () {
        setState () {
            _scale = animation.value;
        });
    });

    retourne l'animation;
}
```

As you can see, now we change the controller's lowerBound and upperBound values to make more sense in our case, as we want the button to grow until its size is twice as big, and we do not want it to be smaller than its initial size (scale = 1.0). Besides that, we change our animation value listener just to get the value from the animation without any calculations.

[451]

Page 470

Animations

Chapitre 15

Traduire l'animation

Just like before, we can accomplish a better look in our translation transformation and make it smoother by using AnimationController:

The construction of our widget similar to before; the only exception is the usage of the `Transform.translate()` construction. Now, we have a different value type than `double`. Let's see what we need to change to make an `Offset` animation:

```
createTranslateAnimation() {
  var controller = AnimationController(
    vsync: ceci,
    debugLabel: "démonstration d'animations",
    duration: Duration(seconds: 2),
  );
}
```

[452]

Page 471

Animations

Chapitre 15

```
var animation = controller.drive(Tween<Offset>(
  begin: Offset.zero,
  end: Offset(70, 200),
));
animation.addListener(() {
  setState(() {
    _offset = animation.value;
  });
});
contrôleur de retour;
```

As you can see, here, we used a different approach to modify our widget offset. We used a `Tween<Offset>` instance, passed down to the `AnimationController` object through the `drive()` method, just like we did with `CurveTween` before. This works because the `Offset` class overrides mathematical operators such as subtraction and addition:

```
// part of geometry.dart file from dart:ui package
class Offset extends OffsetBase {
  ...
  Offset operator -(Offset other) => new Offset(dx - other.dx, dy -
other.dy);
  Offset operator +(Offset other) => new Offset(dx + other.dx, dy +
other.dy);
  ...
}
```

This makes the calculation of intermediate offsets (animation values) possible and then the interpolation between two `Offset` values can be achieved.

Check the source code of the `Offset` class for details: <https://github.com/flutter/flutter/blob/main/packages/flutter/lib/src/geometry/offset.dart>.

[com/flu](https://github.com/flutter/flutter/blob/main/packages/flutter/lib/src/geometry/offset.dart)

that to

see the next example for more details.

Transformations multiples et Tween personnalisé

If you remember, we can compose multiple transformations by using the `Matrix4` class. For animations, things are similar; we can combine animations, run one after another, and play them—it's all in our hands. To create a composed animation, we can simply create

multiple transformation values based on a single Animation object.

[453]

Page 472

Animations

Chapitre 15

By doing that, we can achieve something like this:

Thinking in a simple way, we can follow these steps:

1. We can simply have multiple values defined in our class, like this:

```
class _ComposedAnimationsState extends State<ComposedAnimations>
    with SingleTickerProviderStateMixin {
    Offset _offset = Offset.zero;
    double _scale = 1.0;
    double _angle = 0.0;
    ...
}
```

2. And whenever the animation value changes, we can calculate our values based on it:

```
animation.addListener () {
    setState () {
        _offset = Offset(animation.value * 70, animation.value *
            200);
        _scale = 1.0 + animation.value;
        _angle = 360 * animation.value;
    });
}
```

[454]

Page 473

Animations

Chapitre 15

3. And then, we apply the values we have calculated at each step of animation execution in our build() method:

```
_composedAnimationButton() {
    return Transform.translate(
        offset: _offset,
```

```

        child: Transform.rotate(
            angle: _angle - _piRadians,
            child: Transform.scale(
                échelle: _scale,
                enfant: RaisedButton (
                    child: Text("multiple animation"),
                    onPressed: () {
                        if (_animation.status == AnimationStatus.completed) {
                            _animation.reverse ();
                        } else if (_animation.status ==
                            AnimationStatus.dismissed) {
                                _animation.forward ();
                            }
                        },
                    ),
                ),
            );
        }
    }
}

```

This works, and for simple cases it's best to keep like this, as we have fewer objects to take care of and a single animation to play.

To make it more maintainable, however, it's better to separate the value calculation from the animation itself. That's how we can use Tweens; remember the Offset example, where it is calculated and we simply get the value ready for use.

Tween personnalisé

To create a custom Tween class, first, we need to define our value object. Here, we have opted for grouping the transformation values:

```

class ButtonTransformation {
    final double scale;
    final double angle;
    final Offset offset;

    // this none getter returns a initial state of transformation
    // with default scale, no rotation or translation
    static ButtonTransformation get none => ButtonTransformation(

```

[455]

Page 474

Animations

Chapitre 15

```

        scale: 1.0,
        angle: 0.0,
        offset: Offset.zero,
    );
}

```

And then, we extend the Tween class with our defined type:

```

class CustomTween extends Tween<ButtonTransformation> {

    CustomTween({ButtonTransformation begin, ButtonTransformation end} ):
        super(begin: begin, end: end);

    @override
    lerp(double t) {
        return super.lerp(t);
    }
}

```

We need to define our custom Tween lerp() method (lerp stands for linear interpolation), which is responsible for returning the intermediate ButtonTransformation value between begin and end, based on the t value.

By taking a look into the default Tween class's lerp() implementation, we can see it is very simple:

```

// part of tween.dart Tween class

@protected

```

```
T lerp(double t) {
    assert(begin != null);
    assert(end != null);
    return begin + (end - begin) * t;
}
```

It calculates the lerp() value by using the +, -, and * operators on the type T objects. This means we can simply implement those operators in our ButtonTransformation and

Tween will work as it does with any other type:

```
class ButtonTransformation {
    ...
    ButtonTransformation operator -(ButtonTransformation other) =>
        ButtonTransformation(
            scale: scale - other.scale,
            angle: angle - other.angle,
            offset: offset - other.offset,
        );
}
```

[456]

Page 475

Animations

Chapitre 15

```
ButtonTransformation operator +(ButtonTransformation other) =>
    ButtonTransformation(
        scale: scale + other.scale,
        angle: angle + other.angle,
        offset: offset + other.offset,
    );
    ...
    ButtonTransformation operator *(double t) => ButtonTransformation(
        scale: scale * t,
        angle: angle * t,
        offset: offset * t,
    );
}
```

Now, the Tween class is able to generate intermediate ButtonTransformation values as well. We can then use the generated animation values just like before:

```
createCustomTweenAnimation() {
    var controller = AnimationController (
        vsync: ceci,
        debugLabel: "démonstration d'animations",
        durée: Durée (secondes: 3),
    );

    var animation = controller.drive(CustomTween(
        begin: ButtonTransformation.none, // initial state of the animation
        end: ButtonTransformation(
            angle: 360.0,
            offset: Offset (70, 200),
            échelle: 2.0,
        )));
    animation.addListener (() {
        setState () {
            _buttonTransformation = animation.value;
        });
    });

    contrôleur de retour;
}
```

As you can see, the big difference is in the usage of our CustomTween property. Note that we always need to define begin and end values, as Tweens are based on a range defined by the corresponding interpolation.

[457]

Page 476*Animations**Chapitre 15*

With those examples, we have seen how to use and apply the most important animations in Flutter. In the next sections, we will see alternative ways of applying animations to our widgets.

We can build multiple simultaneous animations using separate Animation objects, typically, by setting the same AnimationController as their parent. They are guaranteed to be in sync as we will be using the same Ticker object.

Utilisation d'AnimatedBuilder

Looking at the code that we wrote in the last section, there is nothing wrong with it: it's not too complex or big. However, looking closely, we can see a small problem with it, our button animation is mixed up with other widgets. As long as our code does not scale and get more complex, this is fine, but we know this is not the case most of the time, so we might have a real problem.

The AnimatedBuilder class can help us with the task of separating responsibilities; our widget, whether it is RaisedButton or anything else, does not need to know it is rendered in animation, and breaking down the build method to widgets that each have a single responsibility can be seen as one of the fundamental lemmas in the Flutter framework.

La classe AnimatedBuilder

The AnimatedBuilder widget exists so that we can build *complex widgets that wish to include animation as part of a larger build function*. Just like any other widget, it is included in the widgets tree and has a child property. Let's check its constructor:

```
const AnimatedBuilder({  
    Clé clé,  
    @required Listenable animation,  
    @required TransitionBuilder builder,  
    Enfant du widget  
)
```

[458]

Page 477*Animations**Chapitre 15*

As you can see, we have a few important properties here, besides the well-known key property:

animation: This is the proper animation as a Listenable object.
Listenable is a type that holds a list of listeners and notify them whenever the object changes. As you may already be thinking, AnimatedBuilder will listen for animation updates, so we do not need to do it manually with the addListener() method anymore.

builder: This is where we modify the child widget based on the animation values.

This will be the widget that drives the animation. So, we construct

Revisiter notre animation

To break down our code, modify our animation and make it more maintainable, we start separating what we need for each responsibility. Typically, three things are needed:

The animation itself: Here, we do not need to change anything. Our AnimationController will still be the same.
 Add the AnimatedBuilder widget to our build() method: We will be extracting much of the code related to the animation of the button to make it clear.

The child widget: In our case, it is just RaisedButton that changes according to the progress of the animation:

```
class _AnimationBuilderAnimationsState extends
State<AnimationBuilderAnimations>
  with SingleTickerProviderStateMixin {
  AnimationController _controller;
  Animation<ButtonTransformation> _animation;

  @passer autre
  void initState () {
    super.initState ();
    _animation = createAnimation();
    _controller.forward();
  }
  ...
}
```

[459]

Page 478

Animations

Chapitre 15

As you can see, we have a few changes here:

We do not have a ButtonTransformation field anymore, as it will be managed in our new widget.

We separate the AnimationController from our Animation object. This is very common and better than making type casting everywhere.

And finally, there's just a small detail in the createAnimation() method:

```
createAnimation() {
  _controller = AnimationController(
    vsync: ceci,
    debugLabel: "démonstration d'animations",
    durée: Durée (secondes: 3),
  );
  return _controller.drive(CustomTween(
    begin: ButtonTransformation.none,
    end: ButtonTransformation(
      angle: 360.0,
      offset: Offset (70, 200),
      échelle: 2.0,
    )));
}
```

We do not need to listen for animation updates anymore (we do not have an addListener() call), as this is done directly by the AnimatedBuilder widget.

Then, we modify the build() method to use a new widget:

```
@passer autre
Construction du widget (contexte BuildContext) {
  Conteneur de retour (
    couleur: Colors.grey,
    enfant: Centre (
```

```
child: ButtonTransition(
    animation: _animation,
    enfant: RaisedButton (
        child: Text("AnimatedBuilder animation"),
        onPressed: () {
            if (_controller.status == AnimationStatus.completed) {
                _controller.reverse();
            } else if (_controller.status == AnimationStatus.dismissed) {
                _controller.forward();
            }
        },
    ),
),
```

[460]

Page 479

Animations

Chapitre 15

);
}

As you can see, the animation is clearly separated from the creation of RaisedButton. We instantiate and pass it to a new widget called ButtonTransition, together with our

_animation object. Let's see this brand new widget:

```

class ButtonTransition extends StatelessWidget {
  final Animation<ButtonTransformation> _animation;
  final RaisedButton child;

  const ButtonTransition({
    Clé clé,
    @required Animation<ButtonTransformation> animation,
    this.child,
  }) : _animation = animation,
        super(key: key);

  @passer outre
  Construction du widget (contexte BuildContext) {
    return AnimatedBuilder(
      animation: _animation,
      child: child,
      builder: (context, child) => Transform(
        transform: Matrix4.translationValues(
          _animation.value.offset.dx,
          _animation.value.offset.dy,
          0,
        )
        ..rotateZ(_animation.value.angle * _toRadians)
        ..scale(_animation.value.scale, _animation.value.scale),
      child: child,
    ),
  );
}

```

Basically, `ButtonTransition` handles the modification of its child (`RaisedButton`) without touching it. The important steps of this `build()` method are as follows:

1. First, we add an `AnimatedBuilder` widget to the widget tree.
 2. The child class passed to it will be passed back to us in the `builder` method with optimizations in mind. The whole child subtree does not need to be rebuilt every time the animation gets updated. Holding it and just placing again helps the framework to rebuild only the needed widgets in the `builder` method.

[461]

Page 480

The documentation says:

"Using this pre-built child is entirely optional, but can improve performance significantly in some cases and is, therefore, a good practice."

3. The builder method constructs the tree below it with the required animation changes. Note that we do not have to worry about listening to the animation changes; this builder method will be called whenever the animation is updated.

Although the final visual result is the same, breaking things down into small parts with single responsibilities is an important concept that improves the maintainability of the code and can lead to better performance.

Utilisation d'AnimatedWidget

Separating our animation from widgets with the help of the AnimatedBuilder widget is incredibly easy and can bring up many benefits, as we have seen. Flutter offers another interesting alternative that does the same thing as the AnimatedBuilder widget with a simpler syntax.

This is common when dealing with a well-structured framework such as Flutter; there is typically more than one way of doing something, and it does not mean that there are significant differences between one way or another. AnimatedWidget and AnimatedBuilder are great examples of this. Both aim to separate the animation part from the widget building part.

While the AnimatedBuilder widget delegates the creation of the widget to a builder method, AnimatedWidget defines everything needed with relation to the animation and we simply need to override its build() method to reflect animation updates. At the end, AnimatedBuilder is itself an AnimatedWidget class.

[462]

Page 481

La classe AnimatedWidget

AnimatedWidget is an abstract class and, as we said before, we need to override its build() method directly to reflect animation changes. Its constructor is defined as such:

```
const AnimatedWidget({
  Clé clé,
  @required Listenable listenable
})
```

As you can see, the only required property is the Listenable object so that it can listen to animation updates. The whole widget build logic is the responsibility of its descending class.

Réécrire l'animation avec AnimatedWidget

Using AnimatedWidget in our case would require us to simply modify our ButtonTransition widget. However, as you remember, there is a concept behind this. To follow this, we need to extend the AnimatedWidget class and transform our widget into an animated button in its build() method.

We start by defining our new AnimatedWidget based widget:

```
class AnimatedButton extends AnimatedWidget {
    final RaisedButton button;

    const AnimatedButton({
        Clé clé,
        @required Listenable animation,
        this.button,
    }): super (
        clé: clé,
        listenable: animation,
    );

    @passer outre
    Construction du widget (contexte BuildContext) {
        Animation<ButtonTransformation> animation = listenable;
        return Transform(
            transform: Matrix4.translationValues(
                animation.value.offset.dx,
                animation.value.offset.dy,
                0,
            )
    }
}
```

[463]

Page 482

Animations

Chapitre 15

```
..rotateZ(animation.value.angle * _toRadians)
..scale(animation.value.scale, animation.value.scale),
child: button,
);
}
}
```

Now, we have defined our AnimatedButton widget derived from the AnimatedWidget class. We can highlight two fundamental points here:

The only thing we need to pass to the super AnimatedWidget class is the animation object, so it can listen to animation updates and rebuild itself at the right time.

In the build() method, we access the animation from the listenable property of the superclass and use the animation value just like before.

Choosing when to use AnimatedBuilder and AnimatedWidget may appear confusing at first, but keeping in mind that both can bring up the same benefits helps on this decision. Start by breaking down your widgets with a single responsibility in mind, and taking such decisions will become natural.

Sommaire

In this final chapter, we dived into Flutter animations. We learned the fundamental concepts of animation, which are concept mainly defined by the Animation class.

We saw important classes the framework provides that AnimationController, CurvedAnimation, and Tween. We also revisited our Transformation examples and added animations to them by using the concepts learned in this chapter. Finally, we saw how to create our own custom Tween objects.

Lastly, we saw how to use AnimatedBuilder and AnimatedWidget to make our animation code cleaner and simpler to understand.

That's all folks. In this book, I have tried to go over some basic but fundamental concepts of this incredible platform. I hope you enjoyed and learned something: that's what motivates

us to continue.

[464]

Page 483

Si que vous pourriez apprécier

: other books by Packt:

React Native Cookbook - Second Edition

Dan Ward

ISBN: 978-1-78899-192-6

Build UI features and components using React Native
Create advanced animations for UI components
Develop universal apps that run on phones and tablets
Leverage Redux to manage application flow and data
Expose both custom native UI components and application logic to React Native
Employ open-source third-party plugins to create React Native apps more efficiently

Page 484

Xamarin.Forms Projects

Johan Karlsson, Daniel Hindrikes

ISBN: 978-1-78953-750-5

Set up a machine for Xamarin development

Get to know about MVVM and data bindings in Xamarin.Forms

Understand how to use custom renderers to gain platform-specific access

Discover Geolocation services through Xamarin Essentials

Create an abstraction of ARKit and ARCore to expose as a single API for the game

Learn how to train a model for image classification with Azure Cognitive Services

[466]

Page 485

Autres livres que vous pourriez apprécier

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

[467]

Page 486

Indice

A

- abstract classes [51](#), [52](#), [53](#)
- abstraction [38](#)
- Ahead-Of-Time (AOT) [2](#)
- Android app
 - configuring [252](#), [253](#), [2](#):
 - Android plugin
 - implementing [301](#), [302](#)
 - Android Services
 - reference link [400](#)
 - Android view
 - creating [331](#), [332](#), [3](#)
 - Android virtual device
 - Android-specific code
 - adding, to execute Dart code in background [400](#)
 - BackgroundProcessService class [402](#), [404](#)
 - HandsOn[~] · dProcessPlugin class [400](#), [401](#), [40](#)
 - Android
 - app, releasing for [371](#)
 - camera permission, declaring [324](#)
 - contact permission, adding [320](#)
 - AndroidManifest
 - about [371](#), [372](#)
 - application icon
 - application name [373](#), [374](#)
 - meta tags [372](#)
 - permissions [372](#)
 - AnimatedBuilder c'
 - about [458](#), [459](#)
 - animation, revisi [51](#), [462](#)
 - using [458](#)
 - AnimatedWidget class
 - about [463](#)
 - used, for rewriting animation [463](#)
 - using [462](#)
 - AnimationController class
 - reference link [444](#)
 - AnimationContrôleur
 - about [443](#)
 - concept [443](#)
 - properties [443](#)
 - animations
 - about [441](#)
 - animation class [441](#), [442](#)
 - custom Tween [453](#), [454](#)
 - multiple transformations [453](#)
 - rotate animation [446](#), [447](#), [448](#)
 - scale animation [450](#), [451](#)
 - translate animation [452](#)
 - using [445](#)
 - AOT compilation [10](#)
 - app code [142](#)
 - app permissions
 - managing [315](#)
 - managing, on Flutter [316](#)
 - managing, on Flutter with permission_handler plugin [317](#)
 - app screens
 - app code [141](#), [142](#)
 - favors app, home scr
 - request favor screen [153](#), [154](#)
 - App Store Connect [377](#)
 - app
 - preparing, for deployment [370](#)
 - release mode [371](#)
 - releasing, for Android [371](#)
 - releasing, for iOS [377](#)
 - application packages
 - about [72](#)
 - versus library packages [73](#)
 - arithmetic operators [20](#)
 - async programming

Page 487

with Dart futures [84](#)
with Dart isolates [84](#)

B

background execution, from iOS
reference link [407](#)
background isolate [398](#)
background process
about [395](#), 396, 397
calculation, initial
compute() example [394](#), 395
creating [392](#)
Flutter compute() function [392](#), 393
BackgroundProcs class
about [402](#), 404
PluginRegistrant property [405](#), 406
banners [287](#)
BitmapDescriptor class
reference link [344](#)
bitwise operators
manipulating [22](#), 23
BoxFit enum
reference link [218](#)
brightness [194](#)
brightness property [194](#)
build.gradle
about [371](#), 372
app, signing [375](#)
application ID and version
built-in data types
about [24](#)
BigInt type [25](#)
Booleans [25](#)
collections [25](#)
literals [27](#)
numbers [24](#)
string interpolation [26](#)
strings [26](#)
built-in layout widgets
about [139](#)
container widgets [139](#)
built-in widget
Material Design widgets [135](#)
built-in widgets
about [132](#)

Image widget [133](#), 134, 13
iOS Cupertino widgets [135](#)
Text widget [133](#)

C

callable classes [56](#), 57
camera permission
declaring, on Android [324](#)
declaring, on iOS [324](#)
requesting, in Flutter [325](#)
requesting, with permission_handler plugin [323](#)
canvas class, methods
reference link [426](#)
canvas class
about [424](#), 425
canvas ClipRect
canvas transformation [425](#)
methods [425](#)
Paint object [426](#)
canvas
using [424](#)
caret character [82](#)
cascade notation [44](#)
class inheritance
about [50](#), 51
toString() method
classes [37](#)
Cloud Firestore
about [268](#)
enabling, on Firebase [266](#), 267
favors, loading [269](#), 271
favors, saving on Firebase [271](#)
favors, updating on Firebase [272](#)
used, with NoSQL database [265](#), 266
Cloud Storage
with Firebase Storage [275](#)
collection if operator [18](#)
collections [269](#)
CommonFinders class
reference link [359](#)
composed transformation [420](#), 421
composition [38](#)
compute() example [394](#), 395
constructors
about [42](#), 44, 4

[469]

Page 488

factory constructors [46](#), 47
named constructors [45](#), 46
contact permission
adding, on Android [320](#)
adding, on iOS [321](#)
adding, with permission_handler [320](#)
checking, in Flutter [321](#), 322
requesting, in Flutter [321](#)

custom theme [203](#), 204, 205
custom Tween [455](#), 456, 457
custom widgets
creating [157](#), 158
CustomPaint object
about [428](#)
paint method [428](#)
shouldRepaint method [428](#)

contact
 importing, from phone [317](#)
 importing, with contact_picker plugin [318](#), 320

contact_picker plugin
 used, for importing contact [318](#), 320

container widgets
 about [139](#)
 child widget, positioning [140](#)
 child widget, styling [140](#)

contrast specifications
 reference link [381](#)

control flows [29](#)
 control flows, language tour
 reference link [29](#)

Cupertino widgets
 reference link [205](#)

Cupertino
 about [98](#)
 practice [206](#), 207, 2

CupertinoApp [206](#)

CupertinoPageRoute [222](#)

CupertinoTextField class
 reference link [137](#)

current clip region [425](#)

current transformation [425](#)

CurvedAnimation [444](#)

Curves class
 reference link [444](#)

custom fonts
 default font, overriding in app [210](#), 211
 importing, to Flutter project [208](#), 209, 210
 using [208](#)

custom inputs widgets
 example [175](#)

custom inputs
 about [174](#)
 creating [174](#)

CustomPaint widget
 about [426](#)
 construction details [426](#)
 defining [430](#), 431
 descendant class, [426](#)
 example [429](#)
 paint method, overriding [432](#), 433, 434
 properties [427](#)
 radial chart variant [435](#)
 shouldRepaint method, overriding [431](#)

D

Dart API [303](#)

Dart classes
 about [42](#), 43
 accessors [42](#)
 cascade notation [44](#)
 class inheritance [50](#), 51
 constructor [42](#)
 enum type [43](#), 44
 fields [42](#)
 getters and setters, filed accessors [47](#), 48
 methods [42](#)
 static fields [48](#), 49, 50
 static methods [48](#), 49,

Dart code
 AOT compilation [10](#)
 JIT compilation [9](#)

Dart development tools [11](#)

Dart futures
 about [84](#), 85, 8
 used, in sync

Dart isolates
 about [88](#), 89
 used, in sync naming [84](#)

Dart language, structure
 about [19](#)

[470]

Page 489

collections [33](#)
 control flows [29](#)
 Dart operators [19](#), 20
 Dart types [23](#)
 Dart variables [23](#)
 data structures [33](#)
 function [29](#), 30
 generics [33](#)
 looping [29](#)

Langue de fléchettes
 reference link [19](#)

Dart libraries
 about [58](#)
 creating [63](#), 64
 importing [59](#), 60
 library member priv [65](#)
 library, definition [65](#)
 path variants, importing [62](#), 63
 prefixes, importing [61](#), 62
 show and hide [60](#)
 using [59](#), 60

Dart Maps [26](#)

Opérateurs de fléchette
 about [19](#), 20
 arithmetic oper [0](#)
 bitwise operators, manipulating [22](#), 23
 decrement operators [21](#)
 equality operators [21](#)
 increment operators [21](#)

project structure [73](#), 75
 project, initiating [296](#), [297](#), [298](#)
 pubspec file [77](#), 78
 stagehand [75](#), 77
 versus Flutter packag

Dart script
 code [12](#), 13

Dart SDK
 tools [11](#)

Dart test package [90](#)

Dart types
 about [23](#)
 built-in data types [24](#)
 type inference [27](#), 28

Dart variables
 about [23](#)
 final and const [23](#)

Dart Virtual Machines (VMs) [9](#)

Dart, benefits
 about [8](#)
 garbage collection [8](#)
 portability [8](#)
 productive tooling [8](#)
 statically typed [8](#)
 type annotations [8](#)

Dart, guidance
 reference link [17](#)

Dart, web development
 tools [12](#)

logical operators [22](#), [23](#)
 null-aware operators [23](#)
 null-safe operators [23](#)
 relational operators [21](#)
 type casting operators [22](#)
 type casting, operators [21](#)
 type checking operators [21](#), [22](#)
 Dart package, project structure
 file and directory [74](#)
 Dart package
 about [59](#), [72](#)
 application pac [72](#)
 application packages, versus library packages
[72](#)
 library packages [72](#)
 package dependencies [79](#), [80](#)

DartPad tool
 about [10](#)
 reference link [10](#)

data [193](#)
 decrement operators [21](#)

[471]

Page 490

DevTools
 about [363](#), [364](#), [:](#)
 Flutter inspector
 reference link [365](#)
 document [269](#)
 Document [269](#)
 documentation files [306](#)
 documentation
 adding, to package [305](#)
 generating [307](#)
 drag gestures
 horizontal version drag [164](#)
 pan version drag [166](#)
 scale version drag [166](#), [167](#)
 vertical version drag [165](#)
 Dynamic styling
 used, with MediaQuery [211](#)
 with LayoutBuilder [211](#)
 with MediaQuery [211](#)
 dynamic type [31](#)

side note, on iOS [287](#)

Firebase console tool
 reference link [248](#)

Kit Firebase ML
 adding, to Flutter [290](#)
 barcode scanning [290](#)
 custom model inference [290](#)
 face detection [290](#)
 image labeling [290](#)
 label detector, used in Flutter [291](#), [292](#), [293](#),
[294](#)
 landmark recognition [290](#)
 language identification [290](#)
 Text recognition (OCR) [290](#)
 used, in machine learning (ML) [290](#)

Storage Firebase
 about [275](#), [276](#)
 dependencies, ac
 files, uploading [276](#), [277](#), [278](#)
 used, in Cloud Storage [275](#)

Firebase, technologies
 AdMob [247](#)
 authentication [247](#)
 cloud firestore [247](#)
 cloud function [247](#)
 firebase cloud messaging [247](#)
 hosting [247](#)
 machine learning kit [247](#)
 performance monitoring [247](#)
 realtime database [247](#)

E

EdgeInsets class
 reference link [152](#)
 encapsulation [38](#)
 enum type [43](#), [44](#)
 equality operators [21](#)

F

factory constructors [46](#), [47](#)
 favor tab
 tap gestures [181](#)
 FavorCards
 tap gestures [182](#)
 favors app, home screen
 about [143](#)
 layout code [145](#), [146](#), [14](#)
 favors screen [179](#), [181](#)
 Firebase AdMob
 account [281](#), [282](#)
 account, creating [28](#)
 ads [280](#)
 ads, displaying in Flutter [287](#), [288](#), [289](#)
 in Flutter [285](#)
 side note, on Android [287](#)

authentication [256](#), [257](#)
 authentication screen [259](#)
 authentication services, enabling [257](#), [258](#)
 Flutter app, connecting to [252](#)
 login status, updating [~`](#), [~`](#)
 login with [260](#), [261](#)
 overview [246](#)
 profile, updating [264](#), [~~](#)
 setting up [247](#), [248](#), [24](#)
 SMS code, verifying [2](#)
 verification code, sending [261](#), [262](#)

firebase_auth plugin
 reference link [262](#)

first-class functions [36](#)

[472]

Page 491

Flutter and native
 communication between, with platform channel [389](#)

Flutter app
 .arb files, ~~~~~; with intl_translation [385](#), [386](#), [38](#)
 about [38](#)
 Android app, configuring [252](#), [253](#), [25](#)
 AppLocalization class [303](#), [384](#)
 compiling [14](#), [15](#)
 connecting, to Fireb.
 debugging [360](#)
 debugging features [363](#)
 dependencies [383](#)
 DevTools [363](#), [364](#), [36](#)
 FlutterFire plugin [255](#)
 iOSapp, configuring [255](#)
 localization, adding [382](#)
 Observatory [360](#), [361](#), [36](#).
 Observatory profiler [365](#)
 profile mode [366](#), [367](#)
 profiling [365](#)
 translated resources, using [387](#), [388](#)

Compilation de flutter
 about [103](#), [104](#)
 development cor [14](#)
 release compilation [104](#)
 supported platforms [104](#)

Flutter compute() function
 about [392](#), [393](#)

ReceivePort [393](#)
 SendPort [393](#)

Flutter development environment
 reference link [11](#)
 flutter doctor tool [103](#)

Flutter framework
 Dart, using [13](#)
 Flutter apps, compiling [14](#), [15](#)
 Flutter hot reload, compiling [14](#), [15](#)
 productivity, adding [14](#)

Flutter Gallery
 installation link [135](#)

Flutter hot reload
 compiling [14](#), [15](#)

Flutter inspector

in DevTools [369](#), [370](#)

Internationalisation de Flutt
 about [381](#)
 flutter_localizations package [382](#)
 intl package [382](#)
 intl_translation package [382](#)

Flutter issues
 reference link [396](#)

Flutter logs
 reference link [363](#)

Flutter package
 Dart packages [296](#)
 plugin package, creating [298](#)
 plugin packages [296](#)
 reference link [307](#)
 versus Dart package [296](#)

Rendu flottant
 about [105](#), [108](#)
 framework [107](#)
 OEM widgets [107](#)
 web-based technologies [106](#)

Flutter repository-related issues
 reference link [329](#)

Flutter widget tree
 inspecting [368](#)
 widget inspector [369](#)

Flutter's gesture system:
 gestures [161](#), [162](#)
 pointer layers [160](#), [1](#)

Flutter, accessibility compone
 contrast [381](#)
 large fonts [381](#)
 screen readers [381](#)

Flutter/Dart ecosystems [295](#)

Battement
 about [100](#), [101](#), [102](#)
 accessibility [380](#)
 app permissions, managi
 contact permission, checking [321](#)
 contact permission, re
 executing [117](#), [118](#), [11](#)
 generated project, exec
 lib/main.dart file [117](#)
 pubspec file [114](#), [116](#)
 translations, adding to ap

[473]

Page 492

flutter_localizations package [382](#)
 flutter_test library
 reference link [360](#)

flutter_test package
 about [357](#)
 testWidgets function [357](#)

FlutterFire dependency
 adding, to Flutter project [255](#), [256](#)

FlutterFire plugin
 reference link [255](#)

fonts
 reference link [208](#)

FontStyle enum
 reference link [210](#)

drag [164](#)
 in material widgets [167](#)

pan [164](#)
 press and hold [164](#)

scale [164](#)
 tap [162](#)

getters [47](#), [48](#)

Google Develop sole
 reference link [350](#)

Google Maps API integration
 on Android [341](#)
 on iOS [341](#)

Google Places API
 enabling [350](#)

Form widget
about [170](#)
InheritedWidget, using [171](#), 172
key, using [171](#)
state, accessing [171](#)
used, for validating input [188](#)

FormField [174](#)

FormField state
accessing [169](#), 170

FormField widget
Form state, accessing [171](#)
widget, turning into [177](#), [178](#), [179](#)

Fuchsia OS [100](#), [101](#)

function [29](#), 30
function parameters
about [30](#)
anonymous functions [32](#)
lexical scope [33](#)
optional named parameters [32](#)
optional positional parameters [30](#), [31](#)
required parameters [30](#), [31](#)

G

generics concept
reference link [35](#)

generics
about [34](#), 35
Dart literals, using [34](#)
need for [34](#)

gestures
double tap [163](#)

H

HandsOnBoardingProcessPlugin class [400](#), [401](#), 40
headless iso

Hero animation [235](#)

Hero transition
implementing [236](#), 238, 24

Hero widget
about [235](#), 236
reference link [24](#)

[474]

Page 493

horizontal version drag [164](#)
hot reload [14](#)
Hummingbird [104](#)

I

Image class, constructors
references [134](#)

Image widget [133](#), 134, 13:
image_picker
used, for taking pictures [323](#)

implicit interface [38](#)

implicit interfaces [53](#)

increment operators [21](#)

inheritance [38](#)

inherited widgets [122](#), 130, 131

Input (Forms)
user input, validating [173](#)
validating [173](#)

Input widget
creating [176](#), 177

input widgets
about [168](#)
FormField [168](#)
TextField [168](#)

input
validating, Form widget used [188](#)

integrate
place address obtaining,
google_maps_webservice plugin used [353](#)

integrated development environment (IDE) [8](#)

interfaces [51](#), 53, 54

internationalization
interpolation [442](#)

intl package [382](#)

intl_translation package [382](#)

iOS, threading
reference link [411](#)

iOS-specific code
adding, to execute Dart code in background [406](#)
SwiftHandleBackgroundPlugin class [407](#), 40

iOS
app, releasing for [377](#)
camera permission, declaring [324](#)
contact permission, adding [321](#)

iOSApp
configuring [255](#)

IsolateNameServer class [394](#)

J

JavaScript compilation [9](#)
JIT compilation [9](#)
Just-In-Time (JIT) [9](#)

L

last in first out (LIFO) [220](#)

LayoutBuilder
about [211](#), 213, 214
dynamic styling,

library documentation [306](#), 307

library member privacy [64](#), 65

library packages [72](#)

bibliothèque
defining [65](#)
multiple-file library, export statement [69](#), 70, 71
single-file library [65](#), 66
splitting, into multiple files [6](#), 8, 69

logical operators [22](#)

looping [29](#)

iOS Cupertino [205](#), [206](#)
 iOS Cupertino widgets
 about [135](#)
 reference link [135](#)
 iOS plugin
 implementing [302](#), [303](#)
 iOS view
 creating [333](#)
 iOS views, issues
 reference link [330](#)

M

machine learning (ML)
 about [247](#)
 with Firebase ML Kit [290](#)
 map camera
 animating, to location [349](#)
 map interactions
 adding [347](#)
 GoogleMapController [348](#)
 GoogleMapController, obtaining [348](#)

[475]

Page 494

map camera, animating to location [349](#)
 markers, adding [347](#), [348](#)
 map
 displaying [328](#), [329](#)
 displaying, on Flutter [2](#)
 displaying, with google_maps_flutter [336](#), [337](#)
 google_maps_flutter plugin [336](#), [337](#)
 markers, adding [344](#)
 platform view widget, creating [330](#), [331](#)
 platform views [329](#)
 Maps API
 enabling, on Google Cloud Console [339](#), [340](#)
 Maps Platform
 reference link [339](#)
 MapType
 reference link [337](#)
 Marker class [344](#)
 markers
 adding, to GoogleMap widget [345](#), [346](#)
 adding, to map [344](#)
 Marker class [344](#)
 Material Design icons
 reference link [116](#)
 Material Design widgets, components
 about [138](#)
 buttons [136](#)
 date and time pickers [138](#)
 dialogs [137](#)
 scaffold [136](#), [137](#)
 selection widgets [1](#)
 text fields [137](#)
 Material Design widgets
 about [135](#)
 reference link [135](#)
 Material Design, color tool
 reference link [204](#)
 Material Design, for Flutter platform
 reference link [198](#)
 Conception matérielle
 about [198](#)
 custom theme [203](#), [204](#), [205](#)
 MaterialApp widget [199](#), [20](#)
 Scaffold widget [201](#), [203](#)
 MaterialApp widget [199](#), [201](#)
 MaterialPageRoute [222](#)

Matrix4 class
 about [415](#)
 reference link [415](#)
 MediaQuery
 about [214](#)
 dynamic styling, using [211](#)
 example [214](#), [217](#)
 responsive classes [2](#)
 message codec
 reference link [334](#)
 message codecs [391](#), [392](#)
 method channels [389](#)
 method invocations [300](#)
 MethodChannel [300](#)
 mixins
 about [51](#)
 behavior, adding to class [54](#), [55](#), [56](#)
 mobile app development frameworks
 by Google [100](#)
 comparison [94](#)
 Dart [99](#), [100](#)
 developer resources [102](#), [103](#)
 existing frameworks, difference between [94](#)
 Flutter, problems solving [95](#), [96](#)
 high performance [97](#)
 open source frameworks [101](#), [101](#)
 tooling [101](#), [102](#),
 UI, control [97](#)

N

named routes
 about [228](#)
 arguments [230](#)
 overview [229](#), [230](#)
 results, retrieving from [229](#), [232](#)
 native and Flutter
 communication between, with platform channel [389](#)
 navigation history [221](#)
 navigation stack [221](#)
 Navigator widget [219](#), [220](#)
 implementing [222](#), [223](#), [22](#)
 NoSQL database
 about [266](#)
 with Cloud Firestore [265](#)

[476]

Page 495

null-aware operators [23](#)
null-safe operators [23](#)

O

objects [37](#)
Observatory profiler [365](#)
Observatory UI address [361](#)
Observatoire
 about [360](#), 361, 362
 reference link [36](#)
Offset class
 reference link [453](#)
OOP, artifacts
 class [36](#)
 enumerated class [36](#)
 interface [36](#)
 mixins [36](#)
OOP, features
 abstraction [38](#)
 composition [38](#)
 encapsulation [38](#)
 in Dart [36](#)
 inheritance [38](#)
 objects and classes [37](#)
 polymorphism [39](#)
OOP
 in Dart [35](#)
Original Equipment Manufacturer (OEM) [27](#)
Overlay widget [220](#)
overloading [39](#)

P

package dependency
 about [79](#), 80
 source constraint
 specifying [80](#), 81
 version constraint [81](#)
package/plugin project
 creating [295](#), 296
package
 documentation files [306](#)
 documentation, adding [305](#)
 library documentation [306](#), 307
 publishing [308](#)
PageRouteBuilder class

reference link [233](#)
PageRouteBuilder
 about [233](#)
 custom transition, in practice [233](#), 234
painters
 using [424](#)
pan version [166](#)
path variants
 absolute file path [63](#)
 package [63](#)
 relative file path [62](#)
 URL, over web [63](#)
performance overlay
 about [367](#), 368
 reference link [36](#)
permission_handler
 used, for adding contact permission [320](#)
 used, for requesting camera permission [323](#)
phone's camera
 integrating [322](#)
pictures
 taking, with image_picker [323](#)
platform channel
 about [390](#)
 message codecs [391](#), 392
 reference link [391](#)
 used, in communication between Flutter and native [389](#)
platform channels [389](#)
platform class
 about [197](#), 198
 reference link [19](#)
Platform View [140](#)
platform view widget
 Android view, creating [332](#), 333
 creating [330](#), 331
 iOS view, creating [334](#)
 usage [334](#), 336
platform views
 about [329](#)
 enabling, on iOS [330](#)
plugin package
 example [304](#)
 using [304](#), 305
plugin project dev

[477]

Page 496

recommendations [308](#), 309
plugin project structure
 about [298](#), 299
 Android plugin, defining [301](#), 302
 Dart API [303](#)
 iOS plugin, implementing [302](#), 303
 MethodChannel [300](#)
PluginRegistrantCallback property [405](#), 406
pointer layers [161](#)
polymorphism [39](#)
profile mode, Flutter app
 performance overlay [270](#), 270
pubspec file [77](#), 78, 114

R

radial chart variant
 about [435](#)
RadialChart widget, defining [436](#)
RadialChartPainter class, defining [436](#), 438,

scale transformation [417](#)
scale version drag [166](#), 167
screen transitions
 about [232](#)
 PageRouteBuilder [233](#)
SendPort [393](#)
Services APIs [100](#)
setters [47](#), 48
single line [439](#)
single-file library [65](#), 66
Skia graphics engine [97](#)
software development kit (SDK) [10](#)
source constraint
 about [82](#), 83
 Git source [83](#)
 hosted source [82](#)
 path source [83](#)
 SDK source [83](#)
stagehand [76](#), 77
stateful widgets

ReceivePort [393](#)
 relational operators [21](#)
 release mode [371](#)
 Request a favor button
 user, tapping on [186](#)
 request favor screen
 about [153](#), 154
 layout code [154](#),
 Requesting a favor
 about [187](#), 188
 close button [188](#)
 SAVE button [188](#)
 rotate animation [446](#), 447, 448
 rotate transformation [416](#)
 rotate() factory constructor
 properties [418](#)

Route widget [221](#)
 RouteSettings class
 reference link [221](#)
 RouteSettings instance [221](#)
 runes concept [27](#)

S

Scaffold widget [201](#), 203
 scale animation [450](#), 451

about [124](#)
 in code [124](#), 125, 1
 versus stateless widgets
 stateless widgets
 about [123](#)
 in code [124](#), 125, 1
 versus stateful widgets
 static code analysis, built-in analyzer
 reference link [14](#)
 static fields [48](#), 49, 50
 static methods [48](#), 49,
 status listeners [442](#)
 Stepper widget
 reference link [260](#)
 SwiftHand^s [407](#), 40
 syntactic sugar

T

TalkBack [381](#)
 tap gestures, on FavorCards
 about [182](#)
 do action, handling [185](#), 186
 FavorsPage [183](#), 184
 going to Stateful' [183](#), 184
 [^] [183](#), 18

[478]

Page 497

refuse action, handling [184](#), 185
 tap gestures

 on favor tab [181](#)

TensorFlow Lite
 reference link [290](#)

Text widget [133](#)

Text widget, properties
 reference link [133](#)

TextField class
 reference link [137](#)

TextField widget
 controller, using [169](#)

theme widget

 about [191](#), 192,

 brightness property

 platform class [197](#), 198

ThemeData property [193](#)

theming, in practice [195](#), 196, 197

ThemeData class

 reference link [193](#)

ThemeData property [193](#)

threads, in Android

 reference link [404](#)

Ticker interface [444](#)

TickerProvider interface [444](#)

top-level functions [56](#), 58

toString() method [51](#)

Transform class

 used, for transforming widget [413](#)

Transform constructor

 properties [417](#)

Transform widget [414](#)

Transform.translate() factory constructor

 properties [420](#)

transformation

 applying, to widget [421](#)

 composed transformation [420](#), 421

 rotate transformation [416](#)

 scale transformation [417](#)

 translate transformation [418](#)

types, exploring [415](#)

Tween [445](#)
 type casting operators [21](#), 22
 type checking operators [21](#), 22
 type inference [27](#), 28

U

UI, with widgets
 app screens [141](#)
 creating [141](#)
 unit testing, in Flutter app [356](#)
 unit testing
 with Dart [89](#)
 unit tests
 writing [90](#), 91, 92
 URL, from app
 flutter_linkify plugin [212](#), 213
 launching [310](#), 311
 link, displaying [311](#)
 URL, from Flutter app
 launching [314](#)
 url_launcher plugin [314](#), 315
 user gestures
 handling [160](#)
 user interface (UI) [97](#)

V

variables [56](#), 58
 version constraint
 about [81](#), 82
 any/empty [81](#)
 concrete version [81](#)
 minimal bound [81](#)
 range [82](#)
 semantic range [82](#)
 vertical version drag [165](#)
 Virtual Machines (VMs) [9](#)
 VoiceOver [381](#)

W

widget inspector [369](#)

transitive dependencies [72](#)
 translate animation [452](#)
 translate transformation [418](#)
 true object-oriented language [36](#)

widget key_property [132](#)
 widget testing, in F
 about [356](#), 357
 example [358](#), 35
 flutter_test package [.](#)

[479]

Page 498

widget
 about [108](#), 109,
 composability [11](#)
 for animations [140](#)
 for gestures [140](#)
 for transformations [140](#)
 immutability [109](#)
 multiple transforms, applying [423](#), 424
 rotating [421](#), 422
 scaling [422](#)
 transformation, applying [421](#)
 transforming, with Transform class [413](#)
 translating [422](#)

tree [110](#), 111
 turning, into F [177](#), 178, 179
 used, for creating UI [177](#)
 WidgetsApp [226](#), 227, 22

X

Xcode
 about [378](#)
 AdMob [378](#)
 application details [378](#)
 apps, signing [378](#), 379
 Bundle ID [378](#)