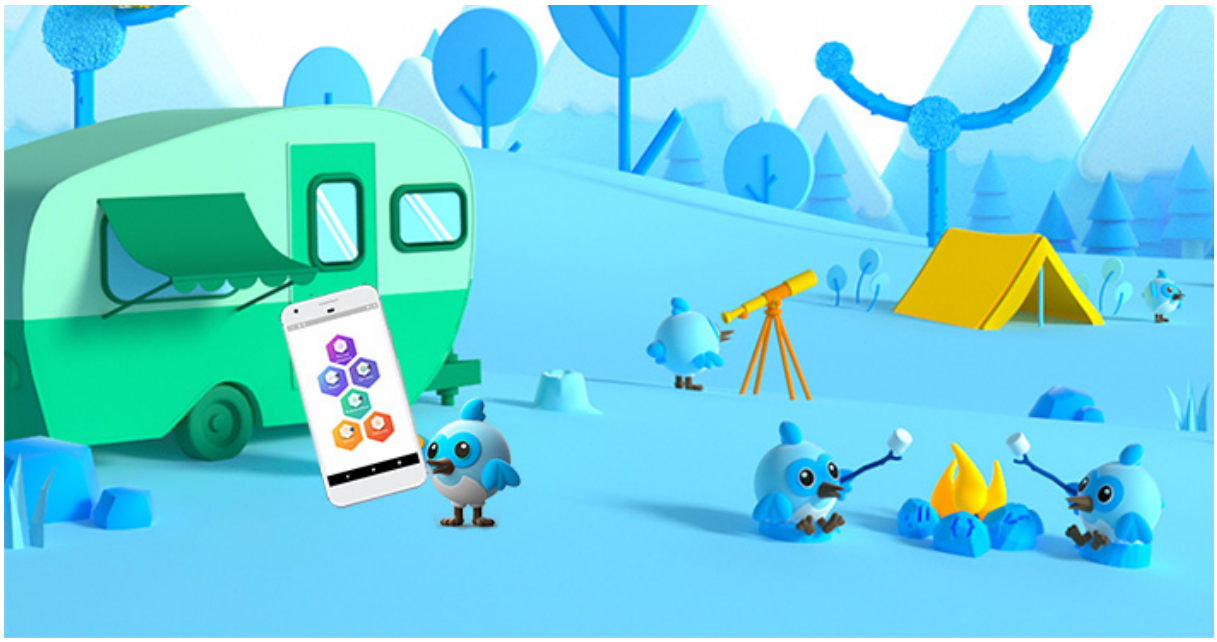
Миграция мобильного приложения на Dart 2.12 (Flutter 2)

- Блог компании Миландр,
- Программирование,
- Разработка мобильных приложений,
- Dart,
- Flutter



3 марта 2021 года разработчики Google представили Flutter 2. Что появилось в новой версии языка Dart? Как теперь быть с разработкой и поддержкой приложений, созданных с использованием Flutter предыдущих версий? И, самое главное, насколько сложно будет мигрировать на версию 2? В этой статье подробно опишем опыт миграции приложения на новую версию Flutter и проблемы, которые могут возникнуть в процессе миграции.

Кто такой и зачем нужен Flutter?

Для тех, кто набрел на статью случайно и понятия не имеет, что такое Flutter — это технология от Google для разработки кроссплатформенных мобильных приложений - да, да приложения будут работать и на Android, и на iOS устройствах. Flutter активно завоевывает сердца разработчиков и очень быстро идет от только мобильной разработки к Web и Desktop. Он достаточно прост в освоении, позволяет отказаться от одновременной разработки двух приложений для Android и iOS, он показывает высокую производительность и значительно ускоряет разработку приложений. Ну не прелесть ли? Первая версия была представлена в

начале 2018, а спустя два года мы уже видим Flutter 2 с весьма серьезными доработками и нововведениями.

Что появилось во Flutter 2?

Наиболее громкие изменения:

- Новая версия языка Dart 2.12 с «Sound null safety»;
- Выход «Flutter for web»;
- Большой шаг к мультиплатформенности с «Flutter for desktop».

Разработчики обещают, что миграция существующих решений на Flutter 2 должна пройти просто и быстро, есть гайды по миграции с примерами простых приложений, но можем ли мы им доверять, имея на руках приложение с множеством экранов, общающееся с сервером по api и использующее внешние библиотеки?

Если изменения по части Web и Desktop не затрагивают существующие приложения, то «sound null safety» может потребовать доработок. Почему и зачем «sound null safety» вообще нужна? «Sound null safety» - это фича, которая появилась в новой версии языка Dart 2.12, вышедшей вместе с Flutter 2.0. На просторах сети уже довольно много сказано о нововведениях и о «null safety», поэтому очень подробно на этом останавливаться не будем. Но для целостности картины происходящего немного все же нужно сказать.

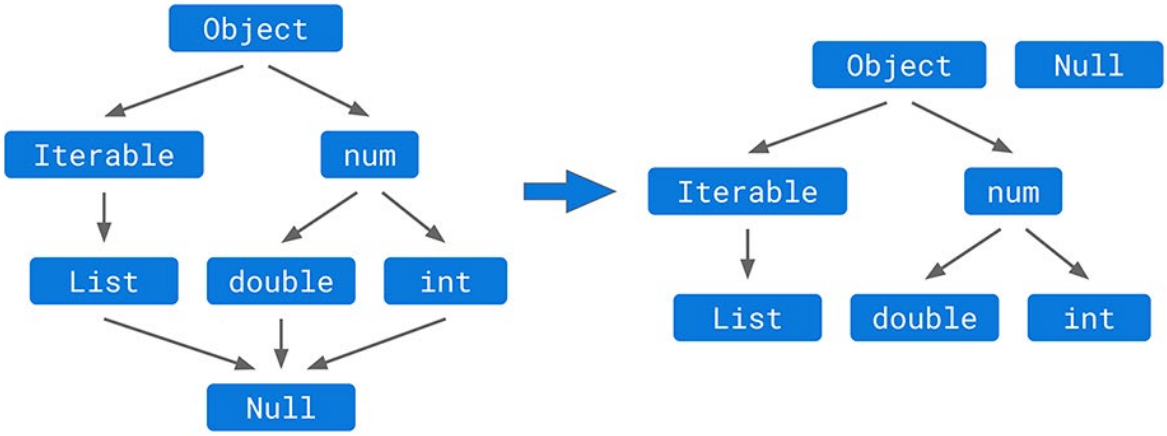
До появления «Sound null safety» все переменные в языке Dart могли принимать значение null. Если разработчик забывал добавить проверку на null перед использованием переменной, то во время работы приложения внезапно можно было получить экран со следующим содержанием:

Не самые приятные события. Разработчики понимают, что такую проверку легко забыть сделать, и, если вместо ссылки на экземпляр определенного типа мы получаем null, который «ничего не знает» о методах и состояниях экземпляра, мы имеем красный экран смерти с «NoSuchMethodError».

Решить эту проблему призван «Sound null safety», основные принципы которого:

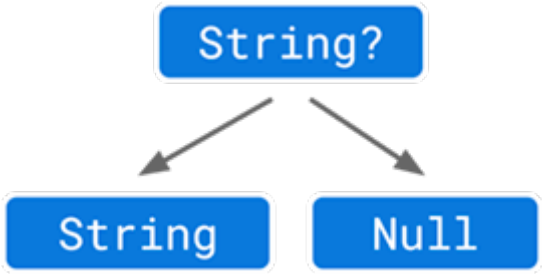
- **Безопасность кода по умолчанию** - все переменные, которые мы создаем, по умолчанию будут non-nullable, пока мы не разрешим им другого поведения.
- **Простота написания кода** - с этим все понятно: не хотелось бы в обмен на безопасность получать сложности в написании и понимании кода.
- **Непротиворечивость кода** - если мы определяем какую-то переменную как переменную non-nullable типа, то она абсолютно точно никогда не будет равна null. Как сказал Евгений Сатуров в своем подкасте, это самый главный принцип «Sound null safety», и загадочное «sound» переводится именно как «непротиворечивость».

Итак, в языке Dart иерархия типов претерпевает некоторые изменения:



Null больше не является подтипом для всех типов, а существует по соседству с ними. Если у нас есть переменная типа String, то она всегда будет содержать строку. Но на практике, конечно, случаются ситуации, когда нужно использовать null в качестве значения для переменной.

В таком случае можно воспользоваться соседом-компаньоном типа, допускающим значение null. В нашем случае этот тип - String?. Мы как будто заранее заставляем себя и других разработчиков сомневаться в том что в этой переменной будет строка: может, будет, а может, и нет.



Ниже пример использования этого типа в деле приготовления бургеров:

```
makeBurger(String burger, [String? meat]) {
  if (meat != null) {
    print('$burger with $meat');
  } else {
    print('Vegan $burger');   }
}
```

Пользоваться переменными с «?» небезопасно, поэтому нужно прибегать к дополнительным проверкам на null или специальным операторам .
Более подробно о новой иерархии типов можно почитать в официальной документации здесь.
В изменениях типов по умолчанию кроются сложности и страхи миграции проекта на Flutter 2. Второй момент, вызывающий опасения, — это сторонние пакеты из pub.dev. На момент написания статьи более 85% пакетов из топ-250 на pub.dev уже поддерживают «null safety». Никто, конечно, не застрахован от того, что нужный пакет и вовсе больше не поддерживается, поэтому перед использованием стоит проверить нужный пакет на pub.dev.

Подготовка к миграции кода

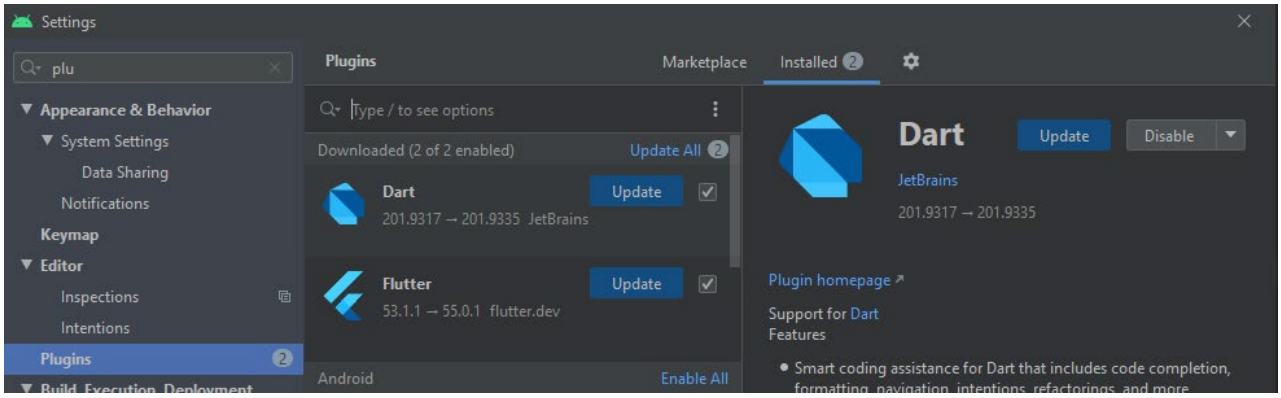
Начнем переезжать на Flutter 2, подсматривая в официальный гайд с использованием мигратора.

Первым делом, конечно же, обновляем Dart и Flutter до последних версий с помощью команды:

```
flutter upgrade
```

Вместе с Flutter'ом обновляется и Dart до версии 2.12.

Помимо SDK, нужно обновить плагины Flutter и Dart, сделать это можно в среде разработки. В AndroidStudio откроем Settings->Plugins. Там нас уже ждут кнопочки "Update". Для применения апдейта обязательно потребуется перезапуск среды.



Далее проверяем, что все пакеты, которые мы используем в своем проекте, обновлены до версии с поддержкой «null safety». Сделать это можно с помощью команды:

```
dart pub outdated --mode=null-safety
```

Выполнять ее нужно из директории проекта, оттуда, где лежит файл pubspec.yaml, который как раз анализируется на присутствие устаревших пакетов. Если такие пакеты в проекте есть, в консоли будет подсказка с информацией, кому из них требуется обновление, с какой версии пакета начата поддержка «null safety», и какая версия наиболее свежая. У нас целых 5 таких пакетов.

```
Computing null safety support...
Package Name        Current    Upgradable  Resolvable  Latest

direct dependencies:
cupertino_icons     X0.1.3     X0.1.3      √1.0.2      √1.0.2
device_id           X0.1.3     X0.1.3      X0.2.0      X0.2.0
http                X0.12.2    X0.12.2     √0.13.1     √0.13.1
json_serializable   X3.5.1     X3.5.1      √4.1.0      √4.1.0
shared_preferences  X0.5.12+4  X0.5.12+4   √2.0.5      √2.0.5

5  dependencies are constrained to versions that are older than a resolvable version.
To update these dependencies, edit pubspec.yaml, or run `dart pub upgrade --null-safety`.
```

Здесь Dart подсказывает нам, что для автоматической смены версий пакетов можно воспользоваться командой:

```
dart pub upgrade --null-safety
```

Пробуем, не получается...

```
null-safety compatible versions do not exist for:
 - device_id

You can choose to upgrade only some dependencies to null-safety using:
  dart pub upgrade --nullsafety cupertino_icons http intl shared_preferences json_serializable

Warning: Using null-safety features before upgrading all dependencies is
discouraged. For more details see: https://dart.dev/null-safety/migration-guide
```

Кажется, пакет device_id все еще не может в «null safety». На pub.dev выясняется, что все еще хуже: обновляли его последний раз в апреле 2019. Но есть и хорошие новости, этот пакет в проекте используется только в одном месте при формировании http запросов к серверу для определения ID устройства. Уходит некоторое время на поиски и тестирование альтернативы, удается найти «null safety» пакет, в котором есть методы для определения ID устройства, - platform_device_id. Если бы такой альтернативы не нашлось, пришлось бы делать форк и допиливать пакет самостоятельно. Добавляем platform_device_id актуальной версии в pubspec.yaml вместо device_id. Пробуем выполнить апгрейд пакетов еще раз.

Теперь все сработало отлично, пакеты обновлены!

Другой путь обновления пакетов: поправить версии руками в pubspec.yaml, а потом выполнить команды:

```
dart pub get
```

```
dart pub upgrade
```

Результат будет таким же.
Сразу переходить к миграции кода в нашем случае не получается: в проекте появились ошибки. Методы *post()* и *get()* пакета http в новой версии поменяли тип аргумента uri, вместо String теперь нужен Uri. Эта проблема тоже довольно быстро решается с помощью метода *Uri.parse()*.

После обновления SDK, плагинов и пакетов проект все еще собирается и работает, но остается самый главный шаг - миграция кода.

Миграция кода

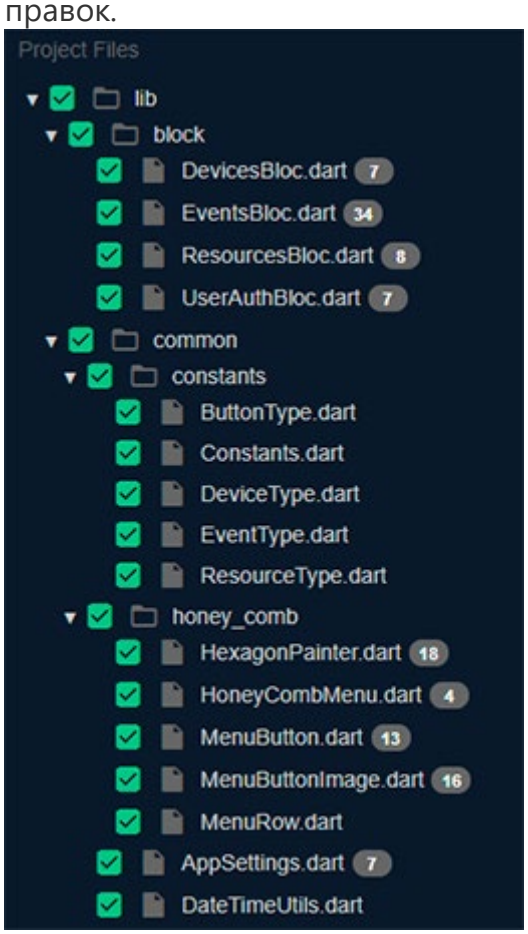
Ее можно осуществить вручную или воспользоваться мигратором, вызвав команду:

```
dart migrate
```

Мигратор проводит анализ нашего проекта и, если все хорошо, отдает ссылку, перейдя по которой можно найти “предложения по миграции”, в которых мы

видим файлы исходного кода нашего проекта с автоматически внесенными изменениями, поддерживающими «null safety».

Напротив каждого файла - цифра с количеством изменений, которые мигратор внес в него. Есть нетронутые файлы, но есть и те, в которых достаточно много правок.



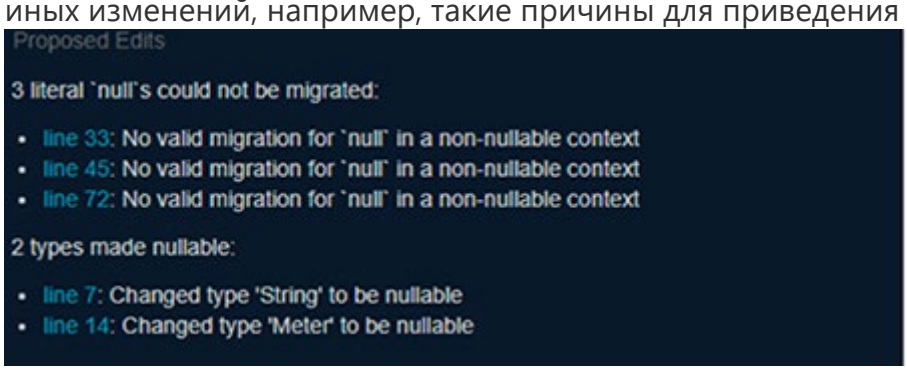
Рассмотрим более подробно предлагаемые изменения. Где-то мигратор добавил «?» к типу поля или переменной, сделав его nullable.

```
class ButtonItem {
  String? title;
  Color  gradientStartColor;
  Color  gradientEndColor;
  String  imagePath;
  String  route;
  bool  locked;
  ButtonType  type;
  Meter? meter;
```

Появились комментарии `/* no valid migration */` как в примере к строчке, где метод возвращает null, а принимающая сторона этого не ждет.

```
factory ButtonItem.fromType(ButtonType buttonType) {
  switch (buttonType) {
    case ButtonType.RESOURCE_CONSUMPTION: {
      return new ButtonItem('Расход ресурсов', Color(0xFFD954D3), Color(0xFF7D42D7), 'image/menu_item_counters.png',
    } default: {
      print('Error! Unknown ButtonType: ' + buttonType.toString());
      return null /* no valid migration */;
    }
  }
}
```

В окошке справа от кода можно найти подробности о причинах внесения тех или иных изменений, например, такие причины для приведения к nullable поля title:



- поле не является final, значит его значение может измениться после определения в конструкторе;
- его значение задается другой переменной, которая может быть не инициализирована при создании.

Здесь же можно воспользоваться кнопочками «Add…», чтобы добавить к типу String /*!*/, сообщив мигратору, что мы уверены, что это поле должно быть non-nullable, перезапустить миграцию и понаблюдать, как меняется код, использующий это поле. Вот, например, при передаче «meter.customName» в конструктор ButtonItem появился «!».

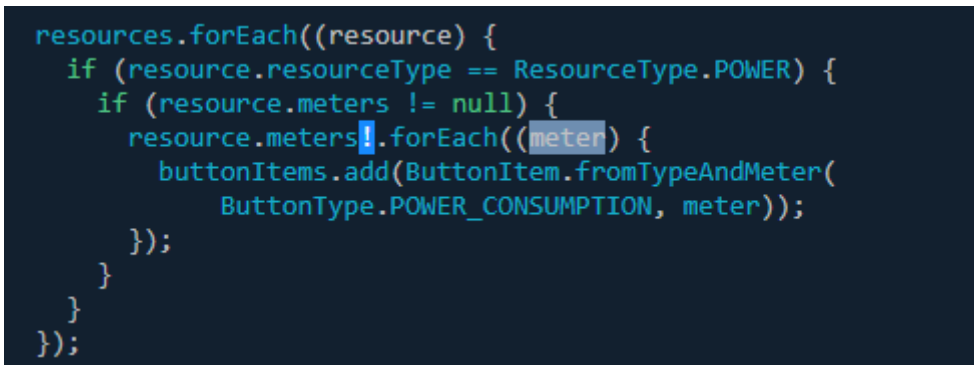
```
factory ButtonItem.fromTypeAndMeter(ButtonType buttonType, Meter meter) {
  switch (buttonType) {
    case ButtonType.POWER_CONSUMPTION: {
      return new ButtonItem(meter.customName!, Color(0xFFFBC63D), Color(0xFFFB9535), 'image/menu_item_control.png', 'device_consump
    }
    default: {
      print('Error! Unknown ButtonType: ' + buttonType.toString());
      return null /* no valid migration */;
    }
  }
}
```

Пробежавшись по коду, можно заметить, что везде, где есть использование nullable, но требуется non-nullable переменная, мигратор добавил оператор «!». Он «уговаривает» метод или выражение, которое ждет только non-nullable, взять из наших рук nullable. Оператор «!» относится к null-aware операторам, таким как «?.», «??», «!.» (подробно про их использование можно почитать здесь).

```
ComboMeal(Drink? drink) {
  drink!.addIce(); //приложение упадет
}

ComboMeal(null);
```

Мигратор часто использует оператор «!» в коде, но, так как это может быть небезопасно, нужно обязательно проверить каждое появление «!» и проанализировать, можно ли как-то это избежать.

Вместо «!» подойдет простая проверка на null, но не всегда. Можно встретить, например, вот такую ситуацию с полем meters:



Проверка есть, но мигратор все равно добавляет «!».

Оказывается, проверка на null перед использованием «перетаскивает» на светлую non –nullable сторону только локальные переменные и не работает с полями класса. На простых примерах это выглядит так:

```
ComboMeal(Drink drink) {
  if (drink.bestTemperature != null) {
    keepTemperature(drink.bestTemperature); // ошибка компиляции
  }
}
```

```
ComboMeal(Drink drink) {
  int? bestTemperature = drink.bestTemperature;
  if (bestTemperature!= null) {
    keepTemperature(bestTemperature); // null safety
  }
}
```

Такое поведение объясняется тем, что обращение к полю класса может скрывать за собой обращение к геттеру, а внутри геттера может быть реализована какая-то функция, меняющая свое поведение в зависимости от ситуации. При сохранении значения поля в локальную переменную и работе с ней мы уже не беспокоимся о том, что значение самого поля могло поменяться.

Оператор «?.» – равнозначен проверке на null, но с ним тоже лучше быть осторожнее. В примере ниже метод addIce() просто не вызовется.Приложение не упадет, но что, если для дальнейшей работы приложения выполнение этого метода было критично?

```
ComboMeal(Drink? drink) {
  drink?.addIce(); // addIce не вызывается
}
...
ComboMeal(null);
```

Еще обойти опасный «!» там, где это возможно, хорошо помогает оператор «??», позволяющий заменить возможный null на значение по умолчанию.

```
ComboMeal(Drink? drink) {
  keepTemperature(drink.bestTemperature ?? 70);
}
```

Перед некоторыми полями классов мигратор добавил слово late.

Ключевое слово late позволяет обойти использование nullable типов для полей класса, когда они не инициализируется сразу, но и значение null не является для них необходимым. В примере ниже получим ошибку компиляции, так как поле «burgerName» не может быть nullable и не инициализируется сразу либо в конструкторе.

```
class ComboMeal {
  String burgerName; // ошибка компиляции

  void comboWithCheeseburger() {
    burgerName = 'Cheeseburger';
  }

  void comboWithChickenBurger() {
    burgerName = 'Chicken burger';
  }

  getComboMealName() {
    return 'ComboMeal with ' + burgerName;
  }
}
```

Следующий фрагмент кода будет работать, но для поля «burgerName» значение null бесполезно и скрывает за собой возможность получить ошибку при неаккуратном обращении.

```
class ComboMeal {
  String? burgerName;
```

```
  void comboWithCheeseburger() {
    burgerName = 'Cheeseburger';
  }

  void comboWithChickenBurger() {
    burgerName = 'Chicken burger';
  }

  getComboMealName() {
    return 'Combo meal with ' + burgerName!;
  }
}
```

Ключевое слово late позволяет избежать необходимости делать это поле nullable.

```
class ComboMeal {
  late String burgerName; //null safety

  void comboWithCheeseburger() {
    burgerName = 'Cheeseburger';
  }

  void comboWithChickenBurger() {
    burgerName = 'Chicken burger';
  }

  getComboMealName() {
```

```
    return 'Combo Meal with ' + burgerName;
  }
}
```

```
ComboMeal comboMeal = ComboMeal();
comboMeal.comboWithCheeseburger();
print(comboMeal.getComboMealName());
```

Здесь стоит обратить внимание на то, что такая инициализация все равно не является абсолютно безопасной. Если поменять местами последние две строки кода из примера выше, то ошибки компиляции не будет, но произойдет ошибка «LateInitializationError» во время работы приложения, так как поле «burgerName» не было проинициализировано.

Еще одним применением ключевого слова late является «ленивая» инициализация полей класса.

```
class ComboMeal {
  late String burgerName = _getSurpriseName();
}
```

В нашем случае поле «burgerName» будет инициализировано не при создании экземпляра класса «ComboMeal», а при первом обращении к нему. Такой прием может быть полезен, когда мы уверены в том, что инициализация потребует много ресурсов, но не обязательно будет вызвана.

Автоматическая миграция

Насколько долго придется дорабатывать код после миграции? В нашем проекте около 1200 строк кода. Я потратила на внесение изменений немного времени - около одного рабочего дня. Предлагаю оставить этап несогласия с мигратором и

правок за кадром и нажать кнопку “Apply Migration”, чтобы понять, насколько работоспособен код после автоматической миграции.

Предложения по миграции приняты, в исходном коде проекта появились все изменения. Пробуем собрать проект, получаем ошибки компиляции.

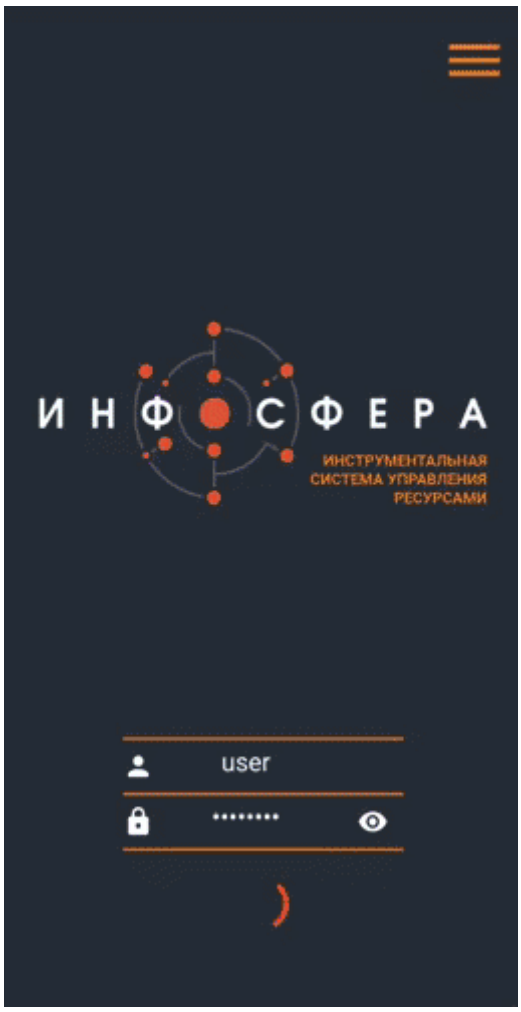
В первую очередь, видим ошибки там, где были комментарии /* no valid migration */. Это те места, где в методах возвращается или передается null. Правим.

Второй тип ошибок компиляции после миграции связан с удалением конструктора по умолчанию для списка в языке Dart. То есть нельзя больше объявить список таким образом:

```
List<String> words = List<String>();
```

Это решение было принято разработчиками из-за того, что конструктор по умолчанию создает список определенного размера, но не инициализирует его элементы, то есть все они имеют значение null. Такой конструктор теперь нельзя применить даже для списка, допускающего содержание nullable элементов. Для разрешения этой проблемы в зависимости от ситуации можно воспользоваться инструментами создания фиксированного или нефиксированного списка – List.empty(), List.generate(), List.fill(), []. Правим места с использованием конструктора по умолчанию.

Ошибки компиляции закончились, приложение собирается, запускается. Согласитесь, довольно быстро.



На этом миграцию приложения можно считать успешной, но с небольшими оговорками.

Впечатления от миграции

Автоматическая миграция не отдает 100% рабочего кода. Но, справедливости ради, стоит признать - так получается из-за того, что исходный код не обеспечивает «null safety». Возможно, если провести какую-то предварительную подготовку, то таких проблем не будет. С другой стороны, правки после миграции были не сложными, заняли не так уж много времени, и вдобавок список ошибок акцентирует внимание на проблемах, которые можно было и не заметить при подготовке.

Выбор мигратора делать переменную/поле nullable или non-nullable, судя по всему, полностью зависит от их использования в коде. Это хорошо прослеживается в окошке с причинами добавления nullable к типу в «предложениях по миграции». Понятно, что нельзя полностью уйти от использования null. Например, при получении данных от сервера и декодировании json-ответов в классы невозможно гарантировать, что все поля будут заполнены, как мы ожидаем. Мигратор, конечно, сам не догадается о контексте и не сделает все поля response-класса nullable. Разработчику придется уделить время на доработку, чтобы получить хороший код.

Встречались еще странные моменты с использованием ключевого слова «late» для полей класса, которые инициализируются в конструкторе. Часть из этих полей мигратор отмечал как «late», а часть делал nullable, как, например, ниже:

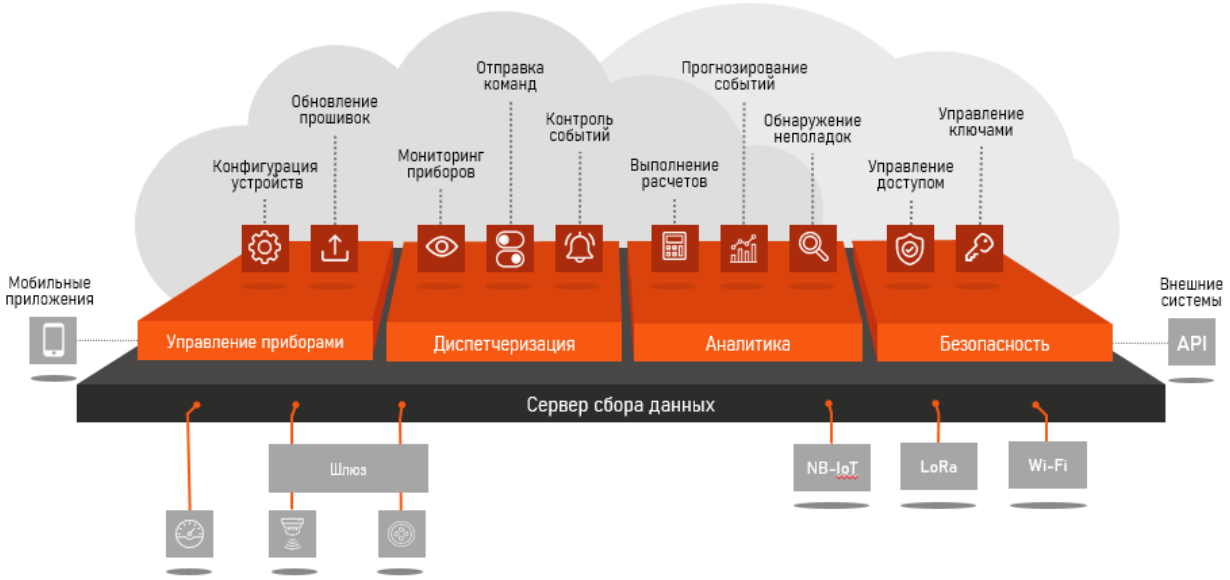
```
class DevicesBloc {
  late DevicesRepository _devicesRepository;

  StreamController? _mainMenuController;

 DevicesBloc() {
   _mainMenuController = StreamController<ApiResponse<List<ButtonItem>>>();
   _devicesRepository = DevicesRepository();
 }
```

Но все это опять же легко поправилось.
Документация по «null-safety» языка Dart и вообще вся документация Dart и Flutter отлично написана, в ней можно найти ответы на большинство возникших вопросов, связанных с работой nullable или non-nullable. Ну и конечно, когда при написании нового кода на Dart 2.12 встает вопрос - делать переменную nullable или non-nullable, лучше выбирать non-nullable и работать с этой концепцией пока не станет очевидно, что без nullable не обойтись.

В целом, процесс миграции оказался довольно простым. Думаю, потраченное время стоит того, чтобы в будущем не ловить красные экраны во время работы приложения. Всем успехов в миграции и разработке!

P.S. Об «Инфосфере»

В завершение стоит немного рассказать о мобильном приложение, которое было героем этой статьи и переживало миграцию на Flutter 2. Оно является частью платформы «Инфосфера», которая разработана центром проектирования программного обеспечения компании «Миландр».



«Инфосфера» представляет собой целый комплекс приложений для мониторинга и управлениями различными умными устройствами. Сейчас наша платформа используется в разных отраслях промышленности и городской среды: ЖКХ, энергетике, спортивно-развлекательных комплексах. Само мобильное приложение предоставляет пользователю информацию о его устройствах и позволяет управлять ими удаленно.

Теги:

- миграция на flutter 2
- мобильные приложения

Хабы:

- Блог компании Миландр
- Программирование
- Разработка мобильных приложений
- Dart
- Flutter