I'm not a bot

As a cross-platform solution, Flutter makes our lives more manageable and enhances development. Building applications with a single, shared codebase that can then be delivered to multiple platforms at once, is a comfortable situation for both groups, developers and users. It shouldn't come out as a surprise that Flutter development is the most popular way for one development team to build on all platforms.Among the most important aspects of choosing Flutter is efficiency, native-like performance, customizable user interfaces and much more that Flutter offers. Each new version of Flutter provides significant improvements and offers more and more possibilities for Flutter developers. It's important to note that any improvement in Flutter translates into improvement in Dart language–its' part and parcel. In this blog post I will explain step-by-step how Dart new features, especially --dart-define-from-file parameter, facilitate application configuration management. Let's dive in!The Dart language is an inherent part of Flutter development. Starting from version 3.7, Dart introduces several new features that facilitate application configuration management. Among the most significant are: --dart-define-from-file parameter,JSON files for accessing env variables. These functionalities empower developers to manage better environment variables, like API keys, database credentials, sensitive data, and application configuration. These can be particularly useful when building multi-layered applications or tailoring an application to various environments (flavoring) like production, testing, or development.The solution presented below can also be great for white-label apps. To explain what a white label app is, from a perspective of CI/CD and what Codemagic's documentation provides:According to Codemagic: "white labeling" refers to automating the process of rebranding your core app for each customer and then publishing the app to stores or other distribution channels. A white labeling pipeline will run scripts to change colors, logos, images, and fonts and update other app settings such as bundle identifiers, provisioning profiles, certificates, API endpoints, and other configuration settings unique to each customer.From a product (app) perspective, "white labeling" can be treated as configuring the app's branding rather than hardcoding to the app's code. JSON file-based configuration addresses most white labeling issues. This setup is undoubtedly an essential step towards making this process possible.Let's imagine that we have various secrets and configurations necessary for white labeling scattered throughout our codebase (e.g. resource values in android/app/src/prod/res/values/string.xml), and the developers have to manage it and remember the location of every piece of data. That'd be nearly impossible to keep intact and likely lead to errors as the app grows. Flutter projects often depend on several external dependencies, including backend systems and 3rd-party services configured on the Dart and native side of the codebase. First, to avoid publishing each flavor's configuration with each app release, we must keep the configuration out of the codebase. It is also quite dangerous because we can accidentally introduce something undesirable into public view. Adding appropriate declarations in .gitignore may not work in such a case. Introducing something undesirable is especially easy for a new developer to join the team and be unaware of the entire project. Furthermore, she or he may have trouble finding something. Secondly, such a scattering of the important ones makes managing it very difficult. Thanks to --dart-define-from-file and JSON configuration files, we have everything in one place. We can easily and quickly change this data depending on the environment. This is a highly convenient solution for flavoring apps. Moreover, configuring such JSON files in our Flutter project allows us to access these variables on the native side easily. Since Flutter 3.16, this configuration is still relatively simple but requires adding some code on the native side to make dart-define-from-file work correctly. Previously, no additional configuration was necessary. Flutter, as a cross-platform solution, allows us now to touch the native side here easily without extra effort. Not every Flutter dev has experience with native languages (it's worth having such knowledge :D), so any such facility is valuable.This mechanism lets developers define environment compile-time variables and load configuration data from JSON files. As a result, it becomes possible to tailor the application flexibly to various environments and easily alter its behavior without the need to modify the source code.Previously, to pass multiple environment variables, we had to use the --dart-define parameter and pass a sequence of these variables.--dart-define=var1=value1--dart-define=var2=value2--dart-define=var3=value3We currently use one JSON file for a specific environment, and we have access to all variables simultaneously. This extremely convenient solution can save us time when, for example, we implement various scripts in Makefile.--dart-define-from-file=keys_staging.json keys_staging.json{ ... "MERCHANT_ID": "merchant_id", "TERMS_CONDITIONS_PATH": "/terms", "PRIVACY_POLICY_PATH": "/privacy-policy", ... }Thanks to this solution, CD becomes much simpler. If, for example, we use VS Code, we can also trivially add the appropriate configuration in the launch.json file.{ "configurations": [{ "name": "Debug Prod", "type": "dart", "request": "launch", "program": "lib/main.dart", "args": ["--flavor", "prod", "--dart-define-from-file", "keys_production.json"] }, { "name": "Debug Staging", "type": "dart", "request": "launch", "program": "lib/main.dart", "args": ["--flavor", "staging", "--dart-define-from-file", "keys_staging.json"] },] }When seeking to access specified environment declaration values, it's advisable to employ one of the fromEnvironment constructors within a constant context for optimal efficiency. For boolean values, utilize bool.fromEnvironment; for integers, opt for int.fromEnvironment; and for other types of values, such as strings, rely on String.fromEnvironment. This approach ensures streamlined access and clarity in handling various environment declarations. It's worth remembering that environment declaration constructors only work when called as const. Most compilers need to be able to evaluate their value at compile time.String get merchantId => const String.fromEnvironment('MERCHANT_ID');Each fromEnvironment constructor necessitates the inclusion of the name or key corresponding to the environment declaration. Additionally, it accommodates an optional named argument, defaultValue, which enables the override of the default fallback value. This fallback value is employed when the declaration is undefined or the specified value cannot be parsed as the expected type. This comprehensive approach ensures robust handling of environment declarations, providing flexibility and clarity in configuration management.To understand how iOS deals with env variables, we need to cite the definition of xcconfig format files.A Configuration Settings File (a file with a .xcconfig file extension), also known as a build configuration file or xcconfig file, is a plain text file that defines and overrides the build settings for a particular build configuration of a project or target. This type of file can be edited outside of Xcode and integrates well with source control systems. Build configuration files adhere to specific formatting rules, and produce build warnings if they do not.The Generated.xcconfig file in a Flutter iOS project is an automatically generated configuration file created during the project compilation for the iOS platform. It's part of building an iOS project in the Flutter environment. Depending on changes in the project or configuration, this file may be regenerated to reflect those changes. You can read more about this topic here.When you enter the realm of the project editor's "Build Settings" tab, you're met with an extensive array of configuration options scattered across projects, targets, and configurations. Fortunately, there is a more efficient method for handling this labyrinthine configuration that doesn't involve navigating through a maze of tabs and disclosure arrows. Xcode build configuration files, often referred to by their .xcconfig extension, offer a means to declare and manage build settings for your application independently of Xcode.In essence, each configuration file comprises a series of key-value assignments following this syntax:MERCHANT_ID=merchantIdAll environmental variables defined in the JSON files are similarly mirrored within this file.LimitationsXcconfig files interpret the sequence // as a comment delimiter, irrespective of whether it's enclosed in quotation marks. The easiest thing to do when we want to pass a specific path in our env variable is to simply exclude the scheme and prefix URLs with https:// in our code instead.Starting with Flutter 3.16, handling dart-define for iOS requires additional setup. The environment variables passed via dart-define are Base64-encoded, and this necessitates decoding them manually during the build process to make them accessible in the iOS native environment. It's mandatory to edit the scheme of the runner to add the following on the pre-actions build script:function entry_decode() { echo "${*}" | base64 --decode; } IFS=',' read -r -a define_items Scheme > Manage Schemes.Select Runner and click Edit....Add Pre-actions:In the scheme editor, go to Build > Pre-actions.Click the + button to add a new pre-action script.Set the Provide build settings from option to Runner.Paste the script into the text field.Interesting caseLet's assume we need access to our env variable in AppDelegate.swift. We also know that build settings defined by the Xcode project file, xcconfig files, and environment variables are only available at build time. The solution to this problem is to use the Info.plist file. Info.plist file is compiled according to the build settings provided and copied into the resulting app bundle. Therefore, by adding references to specific env variables, e.g. $(MERCHANT_ID), we can access the values for those settings through the infoDictionary property of Foundation's Bundle API.Info.plistMerchantId $(MERCHANT_ID)AppDelegate.swiftguard let brazeApiKey = Bundle.main.object(forInfoDictionaryKey: "MerchantId") as? String else { throw NSError(domain: "Merchant id error", code: 1, userInfo: ["reason": "Problem with getting merchant id"]) }Let's take a closer look at the Android platform. Flutter 3.7 introduced a fix to pass the env variable set in the JSON file to build.gradle. Great, isn't it? This fix is part of the Flutter build_info.test file in the flutter_tools folder.Starting with Flutter 3.16, adding this gradle configuration is required to use dart-define-from-file and access environment variables on the native side. This snippet processes the dart-defines property passed during the build, decoding base64-encoded key-value pairs provided via --dart-define, splitting them into individual entries, and converting them into a map (dartEnvironmentVariables). This map allows these variables to be easily accessed and utilized in the Android native build process, enabling seamless integration with Flutter's dart-define functionality.android/app/build.gradle… def dartEnvironmentVariables = [] if (project.hasProperty('dart-defines')) { dartEnvironmentVariables = project.property('dart-defines') .split(',') .collectEntries { entry -> def pair = new String(entry.decodeBase64(), 'UTF-8').split('=') [(pair.first()): pair.last()] } }Suppose we want to manage the deep link scheme from configuration files. This may be needed when, for example, we create a white app - different brands, so the deep links should be different. The deep link scheme is set in AndroidManifest.xml using the current configuration and proper string resource. To use our env variable, create a new resValue in build.gradle. The resource created this way is not limited to AndroidMainfest.AndroidMainfest.xml android/app/build.gradledefaultConfig { ... resValue "string","deeplinkScheme",dartEnvironmentVariables.DEEPLINK_SCHEME ... }android/app/src/prod/res/values/string.xml ... @string/deeplinkScheme @string/deeplinkHost ... The resValue value from the build.gradle file is passed to the Android application as a resource at compile time. Gradle processes these values during compilation and includes them in the Android app's resource files (such as strings.xml).--dart-define-from-file parameter and enhancements in JSON file handling in Flutter 3.7 bring new possibilities for developers in managing application configuration. These features make it easier to customize an application for different environments while maintaining code transparency and flexibility. Users can leverage them when building Flutter applications to increase scalability and ease of configuration.Please note that there are other ways to deal with the problem of configuring our app. There are other ways to use the environment variables mechanism: flutter_dotenv, envied. However, this article focuses on solving this problem using JSON files, because this method works great when we have several flavors and want to manage our app from one place better. If someone is planning to create what can be called a white app, this method will certainly help you deal with the basic problems. We have greater control over all data, making working in a larger team easier. Automation becomes easier with this approach, and code changes are usually unnecessary. We also keep the configuration, which may be hidden somewhere in folders for Android and iOS. All you need to do is make appropriate changes to JSON files. I didn't touch on the issue of safety in the article. Honestly, none of the available front-end methods are completely secure. Remember to use --obfuscate when building your app. Additional data encryption can also increase security. Unfortunately, to put it mildly, there is no 100% certainty that some data will not be caught during reverse engineering. The worst thing you can do is deliver hard-coded secrets somewhere in the codebase. Michał Kochmański So, you're diving into Flutter, right? You must have noticed how important it is to keep things flexible, especially when it comes to different environments like development, testing, and production. That's where environment variables come in handy, helping us configure the app according to the environment it's running in. And the good news is, in the world of Flutter, we have a cool way to handle this using --dart-define.What is — dart-define?--dart-define is basically a neat trick you can use on the command line to set environment variables while compiling your Flutter app. This is great because it allows you to pass dynamic information to the app at compile time. For example, you can set an environment variable to point to different API URLs based on the environment you're working in.How to Use?It's pretty straightforward! The basic syntax is --dart-define=VARIABLE=VALUE. So, if you want to set multiple variables, just separate them with spaces. Let me show you some practical examples.Practical ExamplesSetting API URLs: Let's say you want to set your API URL.flutter run --dart-define=API_URL= in your Flutter code, you can access it like this:const String apiUrl = String.fromEnvironment('API_URL', defaultValue: ' ');Activating Debug Mode: Sometimes, it's useful to turn on debug mode.flutter run --dart-define=DEBUG_MODE=trueAnd in the code:const bool debugMode = bool.fromEnvironment('DEBUG_MODE', defaultValue: false);Setting API Keys: For those secret keys.flutter build apk --dart-define=API_KEY=YOUR_API_KEYIn the Flutter code:const String apiKey = String.fromEnvironment('API_KEY');Customizing Themes: How about adjusting your app's theme?flutter run --dart-define=THEME=darkIn the code:const String theme = String.fromEnvironment('THEME', defaultValue: 'dark');ConclusionUsing --dart-define in Flutter is a lifesaver for managing environment variables. It makes it easy to customize your app for different environments and scenarios. But remember, these variables are set during compilation, so you can't change them during app execution. Plan carefully how and when to set these variables to ensure your app is flexible and efficient in all situations. dart.dev uses cookies from Google to deliver and enhance the quality of its services and to analyze traffic. Share — copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. In the process of our application development, it is common to encounter multiple API environments, such as Development (Dev), Production (Prod), and Staging (Stage). It is imperative to configure the app's environment variables accordingly, aligning with the priorities of each environment.Uncommented the baseUrl every time according to our requirementSometimes, managing API keys based on different environments is necessary. This ensures the appropriate keys are used across various contexts and stages of development.However, this process can lead to challenges in tracking the current environment. Typically, this involves maintaining separate branches or toggling between commented and uncommented files, introducing potential errors.SolutionWe are going to develop a formula or module to get rid of all of our manual work step by step.Step One: Create three .json type files under the directory named "env" (or any other name you prefer)Step Two: These three .json files will contain the baseUrl (which points to our default baseUrl), API key, and secret key.#dev.json{ "baseUrl": " ", "apiKey": "some_key_for_DEV", "secret": "some_secret_for_DEV"}#prod.json{ "baseUrl": " ", "apiKey": "some_key_for_PROD", "secret": "some_secret_for_PROD"}#stg.json{ "baseUrl": " ", "apiKey": "some_key_for_STG", "secret": "some_secret_for_STG"}Step Three: Now bring all the environments we've created to our dart code.We will have to do the same for other variables.Demo UIclass HomePage extends StatelessWidget { const HomePage({super.key}); getData() { var url = Uri.parse(EndPoints.baseUrl + EndPoints.getInfo); var header = {'api_key': Consts.apiKey}; //Http call here debugPrint('Details: $url Header: $header'); } @override Widget build(BuildContext context) { return Scaffold(body: Center(child: Column(mainAxisSize: MainAxisSize.min, children: [const Text("Flutter ENV", style: TextStyle(fontSize: 32, fontWeight: FontWeight.w500),), const SizedBox(height: 10.0,), SizedBox(height: 50, width: 200, child: ElevatedButton(style: ElevatedButton.styleFrom(backgroundColor: Colors.blue), onPressed: () { getData(); }, child: const Text("Test", style: TextStyle(color: Colors.white),)),)],),),); }}If we execute these lines of code, we will get the following UI.\After clicking the 'Test' button, we will display our URL link and header through the 'getData(). However, there are some tasks we need to complete.We need to execute our application using a command, declaring different arguments for various environments.//different command lines while calling the different environment typesflutter run --dart-define-from-file=env/dev.jsonflutter run --dart-define-from-file=env/prod.jsonflutter run --dart-define-from-file=env/stg.jsonCommand line terminalNow if we click on the "Test" button, we will see our base url based on the arguments we have passed in our command line.To build our app bundle or APK, we can utilize commands similar to the following.flutter build apk --dart-define-from-file=env/prod.json //build apkflutter build appbundle --dart-define-from-file=env/prod.json //build appbundleSummaryThis is how we can set different environments for our application using dart-define. It allows us to avoid manually changing our base URL every time.Thanks for reading with patience. As a flutter developer ,you might have the need to keep things flexible, especially when it comes to different environments like development, testing, and production. That's where "dart define" feature of flutter comes handy.This blog post aims to delve deep into Dart define, exploring how its used in Flutter and how it makes your app's building process flexible. So, let's explore together into the world of dart define. "dart define" allows Flutter developers to set constants at compile-time, enabling them to set environment-specific configurations while compiling your Flutter apps. For e.g, you want to set an environment variable to point to different API URLs based on the environment you're working in.It provides a secure way to declare this API URL as a compile-time constant. Here's the basic syntax of how Dart define is used in Flutter development: flutter run --dart-define=API_URL= Here, "API_URL" is an environment variable representing your real API URL and it only works at compile time, not at runtime. As a result, "API_URL" can be accessed in your Dart code as follows: const String apiUrl = String.fromEnvironment('API_URL', defaultValue: ' '); Where, we have used "String.fromEnvironment" method to read environment variables at compile time and 'defaultValue' will be used if the environment variable we are trying to access is not found. You can use Dart Define to Customize Theme in Flutter. Here's how you can do it by defining the THEME using the Dart define from the command line: flutter run --dart-define=THEME=dark You can access the value of the THEME variable defined during the build process: const String theme = String.fromEnvironment('THEME', defaultValue: 'dark'); Here's how you can use dart define to change them in flutter. As an app developer, keeping your app's secret codes safe is important such as API keys is crucial. This is something Dart Define can solve. By using Dart define in your Flutter app, you can securely handle API keys – preventing the leakage of confidential information, thereby maximizing the security of your application. Here's how you can do it by defining the API key using the Dart define from the command line: flutter run --dart-define=API_KEY=Your_Secret_API_Key You can access the value of the API_KEY variable defined during the build process: const String apiKey = String.fromEnvironment('API_KEY', defaultValue: ''); In this approach, the API Key is securely stored and accessed only as necessary. By avoiding direct inclusion in the code, it eliminates the risk of exposure to public version control systems such as Git. If you aim to treat your API as confidential within your application, this method is dependable. Leveraging Dart's define feature ensures your sensitive data remains truly 'secret,' thereby bolstering the security of your Flutter applications. The dart-define-from-file feature in Flutter is incredibly useful for managing environment-specific configuration variables in Flutter applications. Here's why and how you can use it effectively: Environment-specific Configurations: In software development, applications often require different configurations for different environments (e.g., development, staging, production). These configurations can include API endpoints, keys, feature toggles, and more. Managing these directly in source code can be cumbersome and error-prone. Separation of Concerns: dart-define-from-file allows you to store environment-specific variables in separate files (typically JSON), keeping them organized and making it easier to update configurations without modifying the source code. Security and Flexibility: It improves security by keeping sensitive information (like API keys) out of version control systems. It also provides flexibility to switch configurations easily during development, testing, and deployment phases. Create Configuration Files: Create JSON files for each environment (dev.json, prod.json, etc.) in your Flutter project. These files should contain key-value pairs of configuration variables, for example: // dev.json { "apiBaseUrl": " ", "apiKey": "your_dev_api_key" } Reference Configuration in pubspec.yaml: Modify your pubspec.yaml to include dart-define flags referencing these files: flutter: # Add this section for each environment you want to define build_runner: options: --define=APP_CONFIG_FILE=dev.json Accessing Configuration in Dart Code: In your Dart code, you can access these configuration variables using const String: const String _appConfigFile = String.fromEnvironment('APP_CONFIG_FILE'); // Load configuration from JSON file final Map appConfig = await loadConfig(_appConfigFile); // Access specific variables final String apiBaseUrl = appConfig['apiBaseUrl']; final String apiKey = appConfig['apiKey']; Loading Configuration: Implement a function (loadConfig in the example above) to read and parse the JSON file based on the provided environment file name (dev.json in this case). Building for Different Environments: When building your Flutter application for different environments (e.g., flutter build apk), specify the appropriate --dart-define flag: flutter build apk --dart-define=APP_CONFIG_FILE=prod.json This ensures that the correct environment-specific configuration is used during the build process. By utilizing dart-define-from-file, Flutter developers can streamline the management of environment-specific configurations, leading to more maintainable and secure applications across different deployment stages. This approach enhances flexibility and scalability while reducing the risk of configuration errors during development and deployment. In short, Dart Define is a powerful tool for Flutter developers. It enhances security by keeping sensitive data like API keys out of your code. It also boosts performance and simplifies management of environment variables and app configurations. By using Dart Define, you'll experience a smoother and more enjoyable development process. So keep exploring, coding, and building fantastic Flutter apps. Wanna Level up Your Flutter game? Then check out our ebook The Complete Guide to Flutter Developement where we teach you how to build production grade cross platform apps from scratch.Do check it out to completely Master Flutter framework from basic to advanced level. If your Flutter app talks to a 3rd party API that requires an API key, where should you store it?According to various sources, for production apps that need maximum security:the API key should be stored on your own secure server (and never on the client)it should never be transmitted back to the client (to prevent man-in-the-middle attacks)the client should only communicate with your server, which acts as a proxy for the 3rd party API you intend to useThis is because storing API keys on the client is insecure and can cause various issues if they become compromised.But not all keys are created equal: some keys can be accessed by the client while others must be secret and stored securely on the server (the Stripe documentation does a good job explaining this).In fact, this answer on StackOverflow offers a good summary:In the end, it's an economic trade-off that you have to make: how important are the keys, how much time or software can you afford, how sophisticated are the hackers who are interested in the keys, how much time will they want to spend, how much worth is a delay before the keys are hacked, on what scale will any successful hackers distribute the keys, etc. Small pieces of information like keys are more difficult to protect than entire applications. Intrinsically, nothing on the client-side is unbreakable, but you can certainly raise the bar.Without a doubt, there is much to learn about mobile app security (entire books could be written about it).So let me offer some context about what we will cover here. Some ContextIf you're like me and have many open source demo apps that may never make it to production 😅, you may be tempted to store less sensitive API keys on the client (at least early on in the development cycle).And when it comes to API keys and security, you should avoid two major mistakes:Committing a secret key to version control, making it visible to everyone on the Internet Forgetting to obfuscate your API keys, making it easier for attackers to reverse engineer your app and extract the keys As part of this guide, we'll learn how to avoid these mistakes.What is Covered in This GuideWe'll take a look at three different techniques for storing API keys on the client (your Flutter app), along with their trade-offs:Hard-coding keys inside a .dart filePassing keys as command line arguments with --dart-define or --dart-define-from-fileLoading keys from a .env file with the ENVied packageAlong the way, we'll keep these rules in mind:Never add your API keys to version controlIf you store API keys on the client, make sure to obfuscate themBy the end, you'll better understand how to store API keys safely.And I'll also include a security checklist that you can follow in your Flutter projects.⚠ These techniques are not fail-proof. If you have an API key that you can't afford to lose, you should store it on the server (and if you're using Firebase, check out my guide about securing API Keys with 2nd-Gen Cloud Functions). Secure client-server communication involves many considerations beyond the scope of this article (see the links at the bottom for more details).Ready? Let's get started! 1. Hard-coding the key inside a Dart fileA simple and effective way to store our API key is to save it into a dart file like this:// api_key.dart final tmdbApiKey = 'a1b2c33d4e5f6g7h8i9jakblc'; To ensure the key is not added to git, we can add a .gitignore file in the same folder with these contents:# Hide key from version control api_key.dart If we have done this correctly, the file should appear like this in the explorer:File explorer after adding api_key.dart to .gitignoreAnd if we need to use the key anywhere, we can import api_key.dart and read it.Here's an example that uses the dio package to fetch data from the TMDB API:import 'api_key.dart'; // import it here import 'package:dio/dio.dart'; Future fetchMovies() async { final url = Uri(scheme: 'https', host: 'api.themoviedb.org', path: '3/movie/now_playing', queryParameters: { 'api_key': tmdbApiKey, // read it here 'include_adult': 'false', 'page': '$page', },).toString(); final response = await Dio().get(url); return TMDBMoviesResponse.fromJson(response.data); } This approach is very simple, but it has some drawbacks:It's hard to manage different API keys for different flavors/environmentsThe key is stored in plaintext in the api_key.dart file, making the attacker's job easierWe should never hardcode API keys in our source code. If we add them to version control by mistake, they will remain in the git history, even if we gitignore them later.So let's look at the second option. 2. Passing the key using --dart-defineAn alternative approach is to pass the API key with the --dart-define flag at compile time.This means that we can run the app like this:flutter run --dart-define TMDB_KEY=a1b2c33d4e5f6g7h8i9jakblc Then, inside our Dart code, we can do:const tmdbApiKey = String.fromEnvironment('TMDB_KEY'); if (tmdbApiKey.isEmpty) { throw AssertionError('TMDB_KEY is not set'); } // TODO: use api key The String.fromEnvironment method allows us to specify an optional defaultValue, which acts as a fallback if the key is not set. But as we said, we shouldn't hardcode the API key inside our code (whether it's gitignored or not), so it's not a good idea to use defaultValue here.Compiling and running the app with --dart-defineThe main advantage of using --dart-define is that we're no longer hardcoding sensitive keys in the source code.But when we compile our app, the keys are still going to be baked in the release binary:API keys and source code are combined to produce the release binaryTo mitigate risk, we can obfuscate our Dart code when we make a release build (more on this below).Also, it becomes impractical to run the app if we have many keys:flutter run \ --dart-define TMDB_KEY=a1b2c33d4e5f6g7h8i9jakblc \ --dart-define STRIPE_PUBLISHABLE_KEY=pk_test_aposdjpa309u2n230ibt23908g \ --dart-define SENTRY_KEY= @a093qhq4.ingest.sentry.io/2130923 Is there a better way?New in Flutter 3.7: use --dart-define-from-fileSince Flutter 3.7, we can store all the API keys inside a json file and pass it to a new --dart-define-from-file flag from the command line.This way, we can do:flutter run --dart-define-from-file=api-keys.json Then, we can add all the keys inside api-keys.json (which should be .gitignored):{ "TMDB_KEY": "a1b2c33d4e5f6g7h8i9jakblc", "STRIPE_PUBLISHABLE_KEY": "pk_test_aposdjpa309u2n230ibt23908g", "SENTRY_KEY": " @a093qhq4.ingest.sentry.io/2130923" } This solution is quite nice.And if we want, we can even combine it with launch configurations. Using dart defines inside launch.json in VSCodeIf we use VSCode, we can edit the .vscode/launch.json file and add some args to our launch configuration:{ "version": "0.2.0", "configurations": [{ "name": "Launch", "request": "launch", "type": "dart", "program": "lib/main.dart", "args": ["--dart-define-from-file", "api-keys.json"] }] } Moreover, we can define multiple launch configurations with different sets of API keys if needed (api-keys.dev.json, api-keys.prod.json, etc.).If you use IntelliJ or Android Studio, you can use run/debug configurations to achieve the same result.But as it turns out, this leads to a chicken and egg problem. If we hardcode the API keys inside api-keys.json, we have to add it to .gitignore (because keys should not be added to version control).If api-keys.json is gitignored and we do a new checkout of the project, we won't be able to run it until we create api-keys.json again and set the API key(s).Overall, this is not a big issue.Note that any variables defined with --dart-define and --dart-define-from-file are on the Dart side, but not yet on the native side. To learn more, read: Allow to provide compile-time variables for native platforms from a fileNext up, let's try a different approach that uses .env files. 3. Loading the key from a .env file.env is a popular file format that was introduced to give developers a single secure place to store sensitive application secrets, such as API keys.To use this with Flutter, we can add a .env file at the root of the project:# example .env file TMDB_KEY=a1b2c33d4e5f6g7h8i9jakblc # add more keys here if needed And since this file contains our API key, we should add it to .gitignore:# exclude all .env files from source control *.env Then, we can go to pub.dev and find a package that helps us deal with .env files.Enter ENViedThe ENVied package helps us generate a Dart class that contains the values from our .env file.For example, given this .env file that contains our API key:# example .env file TMDB_KEY=a1b2c33d4e5f6g7h8i9jakblc We can create an env.dart file that looks like this:import 'package:envied/envied.dart'; part 'env.g.dart'; @Envied(path: '.env') final class Env { @EnviedField(varName: 'TMDB_KEY') static const String tmdbApiKey = _Env.tmdbApiKey; } Then, we can run this command:dart run build_runner build -d And this will use build_runner to generate an env.g.dart file that looks like this:// GENERATED CODE - DO NOT MODIFY BY HAND part of 'env.dart'; // ********** // EnviedGenerator // ********** // coverage:ignore-file // ignore_for_file: type=lint final class _Env { static const String tmdbApiKey = 'a1b2c33d4e5f6g7h8i9jakblc'; } As a result, we can import env.dart and access the tmdbApiKey as needed.And since our precious API key is hardcoded inside env.g.dart, we should add it to .gitignore:# exclude the API key from version control env.g.dart As a result, we should see the following files in the project explorer:File explorer after adding api_key.dart to .gitignoreWhat about Obfuscation?So far, we've managed to generate a tmdbApiKey constant from our .env file.But this is still stored in plaintext, and if an attacker tries to reverse engineer our app, they may be able to extract the key.To make our API key more secure, we can use obfuscation.This is done by adding the obfuscate: true flag in the @EnviedField annotation:import 'package:envied/envied.dart'; part 'env.g.dart'; @Envied(path: '.env') final class Env { @EnviedField(varName: 'TMDB_KEY', obfuscate: true) static final String tmdbApiKey = _Env.tmdbApiKey; } Any variables annotated with the obfuscate flag should be declared as final (not const).Then, we can re-run the code generation step:dart run build_runner build -d And if we inspect the generated env.g.dart file, we'll see that the API key has been obfuscated:// GENERATED CODE - DO NOT MODIFY BY HAND part of 'env.dart'; // ********** // EnviedGenerator // ********** // coverage:ignore-file // ignore_for_file: type=lint final class _Env { static const List _enviedkeytmdbApiKey = [3083777460, 1730462941, // many other lines]; static const List _enviedkeytmdbApiKey = [3083777414, 1730462956, // many other lines]; static final String tmdbApiKey = String.fromCharCodes(List.generate(_enviedkeytmdbApiKey.length, (int i) => i, growable: false,).map((int i) => _enviedkeytmdbApiKey[i] ^ _enviedkeytmdbApiKey[i])); } Excellent! The API key is no longer hardcoded, making it much harder to extract if an attacker decompiles our app. See the Usage section of the ENVied package to learn how to use it with multiple environments/flavors.API Keys: Code Generation vs Reading at RuntimeUsing code generation and obfuscation makes our API keys more secure (while not 100% fail-proof) against reverse engineering attempts.In contrast, packages such as flutter_dotenv work by adding the .env file to the assets folder and reading its contents at runtime. This is very insecure because any asset file can easily be extracted by unzipping the release APK, thus exposing the environment variables.So don't make the mistake of using flutter_dotenv for your API keys. Instead, use the ENVied package and enable obfuscation.API Keys Security ChecklistIf you choose to use .env files with the ENVied package, follow these steps to secure your API keys:create a .env file to store your API keys in plaintextadd that .env. file to .gitignoreinstall the ENVied packagecreate an env.dart file and define the Env class with one field for each API key, using obfuscate: truerun the code generatoradd the env.g.dart file to .gitignoreimport env.dart and read the API key as neededFor an example that uses this approach, check my movies app on GitHub:Which option to choose?We have now explored three techniques for storing API keys on the client:Hard-coding keys inside a .dart file (not recommended)Passing keys as command line arguments with --dart-define or --dart-define-from-fileLoading keys from a .env file with the ENVied packageSo which one should we choose?Options 1 and 3 are similar as they both require the API key(s) to be stored in a Dart file. For this reason, the file must not be added to version control at any point (as the entire git history can be searched).One advantage of option 3 is that the keys can be obfuscated during the code generation step, adding one extra layer of protection.Option 2 is the official way of defining custom environment variables, and the introduction of --dart-define-from-file makes it more practical.For most apps, I prefer option 3 because it's convenient to store all the keys inside a single .env file.However, there is one use case where option 2 is preferable. Accessing the keys on the native Android and iOS sideCertain Flutter plugins such as Google Maps require that the key is stored inside AndroidManifest.xml or AppDelegate.swift.Prior to Flutter 3.16, it was possible to define the keys using --dart-define and access them on the native side by following the steps in this article:However, this functionality was removed on Flutter 3.16, and this proposal aims to bring it back in a future release:Allow to provide compile-time variables for native platforms from a fileWhile we wait for official support in the Flutter SDK, we can use the flutterenvnative package to provide access to make any environment variables available to the native platforms. For more info, read this:Note about ObfuscationNo matter which option you choose, it's a good idea to obfuscate your entire code when you build a release version of your app, and the official documentation explains how to do this:And while we can use obfuscation to mitigate risk when storing API keys on the client, highly sensitive keys should be kept on the server.So make sure you read the documentation of your API provider of choice and follow the recommended guidelines.How to securely share API keys with other team members?There is one last question that we need to answer:If we don't store our API keys in version control, how can we share them with other team members or retrieve them when we checkout our project on a different machine?The solution is to store them in a secure vault load them into our Flutter project using a CLI tool.Tools such as 1Password CLI are very well suited for this, and I managed to integrate it into my workflow in just a few minutes by following the docs.Alternatively, services such as Doppler take a completely different approach to app security and eliminate .env files altogether. However, I haven't tried using Doppler with Flutter projects yet.Additional ResourcesFor more details about mobile application security, check out these resources:And if you don't want to miss my next articles, sign up for my newsletter: Motivation dart_define is an all-in-one tool to handle secrets and launch configurations both locally and in CI/CD pipelines super easily. Say goodbye to .env files or any other tedious and error-prone ways. With this package you can access your secrets in dart code as well as platform specific configurations. Flutter supports adding compile-time variables with --dart-define option, and using them in both dart code and native configurations. However, with many variables, the commands become quite long quite fast. A workaround for that would be to create a custom launch shell scripts, or storing variables in a JSON file and using it with --dart-define-from-file option. Hovewer, that raises a ton of other issues within CI/CD pipelines. You'd have to gitignore the launch config files, so you don't leak any secrets unwantedly. Then you'd have to create some custom script to initialize and fill those files with proper values in your CI/CD environments. Even after working your way around those issues, you'd have to be very careful to always remember update those pipelines when new values are added (missing values are not recognized at compile-time, leading to possible run-time errors in production). With dart_define, you'll get easy ways of handling secrets and configurations in both local development environment as well as CI/CD pipelines. You can specify which variables are required, so the CLI tool will throw an exception if those are missing. This way missing values will result in failed pipeline run, not run-time error in production. When the generator is run, dart_define reads the values from CLI option override You can override values from system environment or yaml config defaults with CLI options, like so --MY_VARIABLE=MY_VALUE System environment dart_define tries to read values from system environment for smooth integration with CI/CD pipelines Default values You can specify default values within pubspec.yaml configuration, which will be used as fallback values if none of the above are provided. Note that if required is true (default), the generator will fail if 1 or 2 are not provided and the default value will be ignored. Quick Start Install the CLI tool, OR add it into dependencies # activate the CLI tool dart pub global activate dart_define OR # add the tool to dependencies flutter pub add --dev dart_define # If you chooce this method, you'll have to add `pub run` # in from of all of the `dart_define` commands. # Eg. `pub run dart_define generate` Define your variables in pubspec.yaml dart_define: # OPTIONAL: Whether to generate the dart boilerplate or not. # Defaults to true dart: true # OPTIONAL: Whether to generate the json boilerplate or not. # Defaults to true json: true # OPTIONAL: The path to the json file to generate. # Defaults to dart_define.json json_path: dart_define.json # OPTIONAL: The path to the dart file to generate. # Defaults to lib/dart_define.gen.dart dart_path: lib/dart_define.gen.dart # OPTIONAL: The name of the generated class. # Defaults to dart_define class_name: dart_define # REQUIRED: The variables to add to the generated config files variables: # REQUIRED: The name of the variable - name: STRING_VALUE # REQUIRED: The description of the variable description: An example String value # REQUIRED: If `required` is false, this field is mandatory. # Otherwise, it is ignored. default: hello world! # OPTIONAL: If this is set to true, the CLI tool will # throw an exception if the variable is not provided # as an argument. This is handy when using the tool # in CI/CD pipelines. # Defaults to true. required: false - name: INT_VALUE description: An example int value default: 3 required: false - name: BOOL_VALUE description: An example bool value required: true # OPTIONAL: The flavors to use within the app flavors: # REQUIRED: The name of the flavor - name: production # REQUIRED: The description of the flavor description: The production flavor - name: staging description: The staging flavor - name: development description: The development flavor Generate boilerplate # If you activated the CLI tool dart_define generate NOTE: You can override values and variables from pubspec.yaml config with CLI arguments NOTE: This will generate a json file with the given secretes / configurations. It is recommended to gitignore that file. Using the configuration ✪ Use your variables in dart code This tool generates dart boilerplate, so you can access the values very easily. /// This is the code this tool creates class DartDefine { /// An example String value static const stringValue = String.fromEnvironment('STRING_VALUE'); /// An example int value static const intValue = int.fromEnvironment('INT_VALUE'); /// An example bool value static const boolValue = bool.fromEnvironment('BOOL_VALUE'); } /// With this you can access the variables like so: String testValue = DartDefine.stringValue; Access the values in platforms native configurations (see this article) Within android build.gradle // In app/build.gradle defaultConfig { applicationId STRING_VALUE // This allows us to access the value in eg. AndroidManifest.xml resValue "string", "STRING_VALUE", STRING_VALUE } Within AndroidManifest.xml

- https://daotaolaixesontay.com/uploads/file/domobojoger_roxaz_woruselofi.pdf
- how to cook rice in a crock pot
- http://tomekorea.com/userData/board/file/95577513836.pdf
- http://capitaloffice.pl/fotki/file/javizisarem-xavofejuvefu-fojok.pdf
- how to write invitation letter for visa ireland
- pudiro