Shared externally

New Flutter Template: Final Decisions

Is a template the right tool for the job?

The requirements[3][6] document states:

> *We don't, however, provide a template that would be closer to a complete app, with several widgets in separate files, app-wide state, navigation or state restoration. This creates a large gap for novice users to leap over once they outgrow the one-file, one-widget, setState-only nature of Counter. And that leap lands them among a myriad of community solutions and opinions, most of which a novice user is not yet ready to review critically.*

This decision asks: Is a template the right tool to solve this particular problem?

Yes

- Our interviews revealed that Intermediate Flutter devs seek out and rigorously follow templates [2]
    - Indicated they want to see recommendations from the Flutter team [2]
    - Currently rely on community templates [4]
- Provides a concrete starting point they can follow
- Opportunity to define architecture & folder structure based on community best practices
- Optionally demonstrate Flutter best practices: a11y, l10n, analysis_options, assets, etc
- Make baseline decisions for configurable or community templates. e.g. Feature vs Function-first folder structure, include analyis_options or not, etc.[1]

No

- Advanced Users provided feedback saying a community-maintained wizard-like CLI would be preferable to a template[1]
- Advanced users provided feedback stating they believe a Style Guide and Code Samples would more helpful to intermediate developers[1]

Alternatives considered

- Status quo — keep a single template, the Counter app
    - PROs
        - Zero effort
    - CONs
        - Does not address the problem
- Let the community take the lead
    - PROs

        - More open
    - CONs
        - It's hard for the community to sync with all the stakeholders
        - At the end of the day, there would need to be a single point of contact and an arbiter anyway
        - Likely much longer as a project
- Only one template — replacing Counter with the new template
    - PROs
        - Decreases cognitive load when starting development — only one path
        - Decreases maintenance cost (1 template vs 2)
    - CONs
        - By cutting the Counter app, we are losing the "educational template" (widgets, setState, etc.).
        - By cutting the Counter app, we lose the I-know-what-I'm-doing, almost-blank-slate sample which developers use to start their projects.
        - Android Studio stats indicate that no single template is a clear winner. There are two winners and then the rest.
- A plethora of templates — instead of just two, make many templates that address different needs
    - PROs
        - Easier start for specific types of apps (e.g. a template for a "master-detail" apps makes it easy to crank out simple apps of that genre)
    - CONs
        - High maintenance cost.
        - High cognitive load for new developers.
        - Quickly diminishing results, as evidenced by stats from Android Studio.
    - Notes
        - While it seems that two templates is a small number compared to 13 in Android Studio and 8 in XCode, we have to remember that those variants often exist only because the corresponding frameworks (Android SDK and UIKit, respectively) require a lot of non-trivial boilerplate to achieve things that are relatively easy in Flutter.
- Creating a starter app templating tool instead
    - PROs
        - Extensibility
        - Developers can more easily "get coding". For example, a developer who decides to use approach X will do something like `flutter create -t x` or `flutter yeoman --load x` and everything is prepared for them.
    - CONs
        - Much more expensive to implement. Templating systems are famously easy to start but hard to make into something that satisfies (relevant text).
        - Such a tool will need some basic shared structure anyway.
        - A completely open-ended templating system with no "starter" template would do nothing to improve ecosystem cohesion.

Decision

Yes, and may need to be extended with additional documentation and samples

How many templates?

How many templates should Flutter have for apps? As of February 2021, there is only a single one, the Counter app. Should we add one more, or five, or a thousand?

Decision

This is discussed above. Create one additional template.

Who is the target audience?

The following levels of experience are defined by the Flutter learning journeys[5] document.

Beginner

Cons

- Beginner Flutter Developers told us:
  - they are focusing Layouts and basic Navigation [6]
  - learning how to separate Widgets from Data is next goal (or if you need to?) [6]
  - Are still learning the Counter template [6]

Intermediate

Pros

- Intermediate Flutter Interviewees told us they follow templates rigorously and want to see the Flutter team's approach [2]
- Are ready for "Advanced State Management" according to the Learning Journeys Document [5]

Advanced

Cons

- Told us this template does not suit their needs [1]
- Asked for customizable CLI tool, not a new template [1]

Decision

Intermediate Flutter Developers.

What does the template do?

The current template provides users with a Counter App. What should the new template provide users with?

CRUD App

Evaluations[7] [9] Demonstrate:

- Feature-first architecture
- Forms and Form Validation
- Architecture for CRUD setup
- Navigation
- shared_preferences Package
- Serialization

Pros

- Interviewees asked how to separate Widgets from Data [6]
- Interviewees asked for a List/Detail template [2] [6]

Cons

- Evaluations[7] [9] Reveal:
    - Working implementations require quite a bit of code [8]
    - Settings Code may be recycled by end user, but Journal portion would certainly be deleted. Too much to delete?
    - SharedPreferences is great for simple data, but not more. Without nuanced explanation, might guide users down wrong path.
- Seems to make more sense as a Sample Application than template

Networking App (no Authentication)

Evaluation[10] Demonstrates

- http package
- Navigation
- Serialization

Pros

- Interviewees asked how to separate Widgets from Data [6]
- Interviewees asked for a List/Detail template [2] [6]

Cons

- Evaluation[10] reveals:
  - Working implementation requires too much code for a template [8]
  - Almost all of the code would be deleted by end user
  - HackerNews evaluation requires additional flutter_html community packages to render html comments
  - Evaluation Relies on 3rd party service
- Seems to make more sense as a Sample Application than template

Authentication App

Evaluation[11] Demonstrates

- HTTP Package
- in-memory token storage
- Forms and Validation
- Navigation based on ChangeNotifier

Pros

- Interviewees asked how to separate Widgets from Data [6]
- Login & Register is a common requirement
- If the Developer needs authentication, much of the code is reusable

Cons

- Evaluation[11] Reveals:
  - Working implementation requires too much code for a template [8]
  - ReqRes implementation requires 3rd party service
  - Secure Token storage is tricky topic, outside the scope of a template.
  - If the Developer does not need Authentication, much of the code is thrown away
- Seems to make more sense as a Sample Application than template

List/Detail App with Dummy Data

- Evaluations[13] - [30] Demonstrate a gradient of options
  - No file structure[31]
  - File Structure (Feature first)[13]
  - Navigator.push[13]
  - Navigator.onGenerateRoute with arguments[17]
  - Navigator.onGenerateRoute /entity/{id} routing[22]
  - Analysis options
  - MVC Architecture with ChangeNotifier[13]
  - MVC Architecture with ValueNotifier[20], [21], [22], [23]
  - MVC Architecture with Community packages[14], [16], [17], [18], [19], [25], [27], [30]
  - Custom Responsive Design [22]

Pros

- Interviewees asked how to separate Widgets from Data [6]
- Interviewees asked for a List/Detail template [2] [6]
- Architecture similar to Simple State App Management tutorial[32]
- Demonstrates structure, not implementation
- Least amount to rename or remove[8]
- Little code changes required to “upgrade” from Flutter basics to community packages (4-14 lines of code depending on the package[8])

Cons

- Could include or exclude many different features, such as l10n or routing.
- Despite similarity of approaches[3, 13, 14, 16, 17, 18, 19, 25, 27, 30], community disagreement about naming [33, 34]

Decision

List/Detail App with Dummy Data

- with Settings
- Or Logging

Should the template set up localizations?

Should the template include the necessary work to set up localizations? This includes adding an l10n.yaml, en.arb, modify the pubspec.yaml to generate localization, and use the AppLocalizations for Strings throughout the app.

No localizations

Pros

- “…[localizations] add more things a novice needs to think about right off the bat.” [35]
- Beginner Developers did not mention localizations during interview [3]

Cons

- User must search for l10n solution, may end up with non-standard solution
- May cut against requirements to provide concrete guidance[36]

Placeholder that links to more information

Pros

- Intermediate Flutter Interviewees told us they want to see how to approach l10n[2]
- Provide directions to devs who seek it
- No extraneous code to remove for folks who do not want localizations

Cons

- Developer must set up localizations themselves
- Adding localizations to a medium-sized Flutter app is not a trivial task
- Brian's Experience: I've added localizations to two apps, each with around 100-150 Strings each. Each time it has taken me 2-3 working days.
    - I was able to determine this time-frame by reviewing the project boards and commit history of two commercial projects. Unfortunately, I do not have permission to share closed-source code as a reference for these claims.
    - Localizing Strings with access to the BuildContext is quick and easy.
    - Localizing strings without access to the BuildContext is tricky and time-consuming.

Localize only the title of the app

Pros

- Intermediate Flutter Interviewees told us they want to see how to approach I10n[2]
- Demonstrate how to properly localize title of app using onGenerateTitle[29]
- Developer does not have to follow guide to localize app
- Only need to update appTitle in arb, no other code changes required
- "Yes, I think localization by default is a great feature!" [35]
- "Would appreciate it. Both 3 and 4 sound great to me." [35]

Cons

- Does not demonstrate additional features of I10n: variables, pluralization, etc.[29]
- "Setting up localization requires a lot of boilerplate code and very hard to understand at first glance. IMO for beginners, it will be too much to learn." [35]
- "Sure it's nice to have it on a project but this also adds more things a novice need to think about right of the bat." [35]

All strings localized

Pros

- Intermediate Flutter Interviewees told us they want to see how to approach I10n[2]
- Demonstrate how to properly localize title of app using onGenerateTitle[29]
- Demonstrate additional features of I10n: variables, pluralization, etc.[29]
- "Yes, I'm in favor of new localization support by using basic localization delegate and translating all strings to English as default. It might be necessary to explain flutter.generate property in pubspec.yaml in more detail." [35]
- "Yes, I think localization by default is a great feature!" [35]
- "Would appreciate it. Both 3 and 4 sound great to me." [35]
- Developer does not have to follow guide to localize app

Cons

- Devs need to delete or modify the most references as they change the localizations[29]
- Every String in the template requires at least 2 lines instead of 1[29]
- “Setting up localization requires a lot of boilerplate code and very hard to understand at first glance. IMO for beginners, it will be too much to learn.” [35]
- “Sure it’s nice to have it on a project but this also adds more things a novice need to think about right of the bat.” [35]

Decision

Only Localize Title of Application

Should the new template include an analysis_options.yaml file?

The current template does not include an analysis_options.yaml file, but most projects include one. Should the template include an analysis_options.yaml file, and if so, what analysis options should be enabled?

Options

No analyis_options.yaml

Pros

- Was not asked for by beginner or Intermediate interviewees [2, 3]
- No “surprising” or confusing additional analysis[37]
- No additional concepts to learn
- Introduce new lints to people who have not made an explicit choice to have an analysis options file.
- Flutter team will update the defaults soon

Cons

- Unable to disable Implicit Dynamic and Downcasts[40, 41]
- May cut against requirements to provide concrete guidance[36]?
- Community supports including analysis_options.yaml [37]
- Mature projects generally include an analysis_options.yaml

Yes, placeholder with comment on how to customize lints

Pros

- Was not asked for by beginner or Intermediate interviewees [2, 3]
- Teach folks about additional static analyses without imposing any burden

Cons

- “Until then, the problem with giving a default is that anyone who doesn't want to manage their own lint settings will be lost to us: we have no way to update them.”
- Unable to disable Implicit Dynamic and Downcasts[40, 41]
- May cut against requirements to provide concrete guidance[36]
- Was not asked for by beginner or Intermediate interviewees [2, 3]
- Dart Team members wrote: “It may seem that… someone with a close knowledge of Dart should be able to sit down with the list of 146 available lints and produce a list of recommended lints without too much trouble. But that isn’t what we’ve found; it’s simply too big a task.” [39]

Yes, using pedantic package

Pros

- Most popular choice according to user feedback[37]
- Linting rules derived from real applications[39]
- 100% popularity on pub and actively maintained by Google[39]
- May disable Implicit Dynamic and Downcasts as well[40, 41]

Cons

- “Until then, the problem with giving a default is that anyone who doesn't want to manage their own lint settings will be lost to us: we have no way to update them.”
- Was not asked for by beginner or Intermediate interviewees [2, 3]
- Favors a particular package over others[2, 3]
- Does not teach folks how to curate their own analysis options for their project
- User mention too strict

Yes, Community Package

Pros

- May disable Implicit Dynamic and Downcasts[40, 41]
- Promote a community package!
- May have rules that are appropriate specifically for Flutter apps
- A more opinionated package may be a “healthy burden” on devs

Cons

- “Until then, the problem with giving a default is that anyone who doesn't want to manage their own lint settings will be lost to us: we have no way to update them.”
- "You probably also don’t want to ‘pick a winning team’ by showcasing a particular library"[42]
- User Feedback indicates many folks do not want to use community packages in the template[37]

- Was not asked for by beginner or Intermediate interviewees [2, 3]
- More pedantic than pedantic

Yes, custom lints

Pros

- May disable Implicit Dynamic and Downcasts[40, 41]
- Could enable only lints that are helpful for Flutter apps
- Ability to teach folks how to enable or customize specific rules

Cons

- “Until then, the problem with giving a default is that anyone who doesn't want to manage their own lint settings will be lost to us: we have no way to update them.”
- Was not asked for by beginner or Intermediate interviewees [2, 3]
- Dart Team members wrote: “It may seem that… someone with a close knowledge of Dart should be able to sit down with the list of 146 available lints and produce a list of recommended lints without too much trouble. But that isn’t what we’ve found; it’s simply too big a task.” [39]
- Agreeing on linting rules may be time consuming & distract from main goal of the template
- Perhaps Too much information for a developer at this stage in the learning journey?
- Prefer_const_constructors nice-to-have, but produces potentially confusing error messages

Decision

- No Analysis Options. The Flutter Team wants to provide the defaults and is working to improve them in the Future.

What architecture should the app demonstrate?

Most popular Flutter libraries separate Business and Data Logic from Widgets, with some kind of object that connects them together.[3] Beginner/Intermediate Flutter developers have asked for concrete guidance on how to structure Flutter apps. What kind of structure should the new template demonstrate?

No Explicit Architecture

Pros

- No new concepts to learn
- Flutter.dev and community resources exist to learn app architecture[1]
- Advanced Users have asked for a configurable “baseline” rather than a one-sized-fits-all approach[1]

Cons

- Beginner/Intermediate Interviewees explicitly asked for guidance on how to structure apps[2, 6]
- May cut against requirements to provide concrete guidance[36]

Model,View,Controller/ViewModel/Presenter (MVC, MVVM, MVP) with Change/ValueNotifier

Pros

- Beginner/Intermediate developers familiar with ChangeNotifier from “Simple State Management” tutorial, which is part of the Beginner Learning Journey [5]
- Beginner & Intermediate interviewees asked for guidance on how to structure Flutter apps [2, 6]
- Fulfills requirement to provide concrete guidance[36]
- Controller is a common name for a class that makes use of a Change or ValueNotifier to drive one or several Widgets (TextEditingController, ScrollController, etc)
- Controller Naming used by Community [3]
- ViewModel naming is also commonly used by community [45]
- Follows established community best practices: most popular Flutter State Management and architecture libraries separate Business and Data Logic from Widgets, with some kind of object that connects them together.[3, 46]
- Community generally agrees on approach, but appears to disagree on naming[3,44]
- Beginner/Intermediate developers familiar with ChangeNotifier from “Simple State Management” tutorial, which is part of the Beginner Learning Journey [5]
- Few code changes required to “upgrade” from Change/ValueNotifier to community package if desired[8]
- Feedback indicates most folks want the template to use Dart/Flutter libraries[43]
- “I’d lean towards MVC with either ChangeNotifier or ValueNotifier. This seems as a nice balance between complexity and possibilities.” [46]

Cons

- "None of the above because Flutter is MVU because of its reactive nature."[45]
- "My vote would also be for none of the above. I’d either make it specific to the state management chosen (notifiers for Value/ChangeNotifier, cubits for Bloc, etc.) or more generic (blocs, managers, etc.)"[45]
- Some users do not think of Controllers in reactive terms[45]
- “Model” layer of “MVC” is not well defined. Model layers are often broken down into further layers, such as PODOs, “Services,” and "Repositories"[44]
- “Presenter” naming not widely used by Community nor asked for

MVC, MVVM, MVP with Community Package

Pros

- Beginner & Intermediate interviewees asked for guidance on how to structure Flutter apps [2, 6]
- Fulfills requirement to provide concrete guidance[36]
- Follows established community best practices: most popular Flutter State Management and architecture libraries separate Business and Data Logic from Widgets, with some kind of object that connects them together.[3, 46]
- Beginner/Intermediate developers familiar with ChangeNotifier from “Simple State Management” tutorial, which is part of the Beginner Learning Journey [5]
- Feedback indicates most folks want the template to use Dart/Flutter libraries[43]

Cons

- "You probably also don’t want to ‘pick a winning team’ by showcasing a particular library"[42]
- User Feedback indicates many folks do not want to use community packages in the template[37]
- Some users do not think of Controllers in reactive terms[45]
- Some advanced users reject “MVC” structure and terminology[45]
    - "None of the above because Flutter is MVU because of its reactive nature."[45]
    - "My vote would also be for none of the above. I’d either make it specific to the state management chosen (notifiers for Value/ChangeNotifier, cubits for Bloc, etc.) or more generic (blocs, managers, etc.)"[45]
- “Model” layer of “MVC” is not well defined. Model layers are often broken down into further layers, such as PODOs, “Services,” and "Repositories"[44]
- “I’d lean towards MVC with either ChangeNotifier or ValueNotifier. This seems as a nice balance between complexity and possibilities.” [46]
- Community generally agrees on approach, but appears to disagree on naming. Therefore, package may use terms like “Bloc” or “ViewModel” or “Manager” instead of Controller[3,44]

Clean Architecture

Pros

- Beginner & Intermediate interviewees asked for guidance on how to structure Flutter apps [2, 6]
- Fulfills requirement to provide concrete guidance[36]
- Splits the “M” in “MVC” down into distinct layers:
    - UseCase for business logic[49]
    - Repository for data storage[49]
    - Aligns with how advanced community members structure their apps[44]
- Popular architecture for Android applications[51]

- Some advanced users reject “MVC” structure and mentioned [45]

Cons

- No requests for Clean Architecture from community[46]
- Resulted in largest codebase with most classes[8, 52]
- Many concepts to learn, perhaps too much for an intermediate developer[49]
- In smaller apps, the “business logic” layer is often a pass-through to a data layer. For example, often a UseCase class will not do anything other than return the result from a repository class.
    - This is useful for large apps where you may need to add business logic to the use case or combine repositories[47]
    - May be overkill for newer folks, who can combine repositories inside of “Controllers”.[47]
- “Invented” / Promoted by Uncle Bob, who has a history of sexism. May be discouraging for diversity to explicitly support or advocate his architecture.[50]

Decision

- MVC with ChangeNotifier (e.g. Simple App State Management)
- (optional) unify baseline terminology: Controller, MVC, etc.

What libraries or packages should be used?

If we need to include external libraries in the template, what kind of libraries should be considered? Should the template try to stick to only what the Flutter and Dart provide, with extension points to community packages? Should the template make use of community packages to demonstrate best practices?

Only Flutter/Dart team packages

Pros

- Evaluations demonstrate layered architecture without requiring 3rd party packages[13, 20]
- Intermediate Interviewee used unnecessary 3rd party localizations packages because he was unaware of Flutter I10n capabilities[2]
- Feedback indicates this is the preferred approach
    - "In my opinion, it would be easier to maintain the template if it will use only Flutter/Dart team libraries. Otherwise, it may be quickly outdated due if used community package will change the API and docs."[43]
    - "In my opinion, this template should have as less as dependencies as possible in order simplify the maintenance."[43]
    - "As less dependencies as you can. "[43]

Cons

- Popular 3rd party packages reduce Lines of Code and eliminate some types of boilerplate[8]
- Simple App State Management demonstrates Provider package[32]

Flutter Favorites

Pros

- Flutter favorites vetted for quality[53]
- Would allow us to use provider package, used by Simple App State Management tutorial [32]

Cons

- Maintenance burden for community developer
- Package authors may change API
- Most feedback expressed hesitation about using 3rd party packages[43]
    - "Only Flutter/Dart team libraries please"[43]
    - "You probably also don't want to 'pick a winning team' by showcasing a particular library"[42]

Any Community Package

Pros

- Widest range of choices
- Includes some popular libraries, such as get_it or dio that are not from the Flutter team nor Flutter Favorites

Cons

- Maintenance burden for community developer
- Some packages changed rapidly, maintenance burden for Flutter team
- Little code required to "upgrade" to popular community packages[8]
- Most feedback expressed hesitation about using 3rd party packages[43]
    - "Only Flutter/Dart team libraries please"[43]
    - "You probably also don't want to 'pick a winning team' by showcasing a particular library"[42]

Decision

Only Flutter packages (with possible exception of a top 3 flutter favorite)

What kind of Routing should be supported?

Assuming we work with a list view / detail view application, what kind of routing should the template employ? Should it worry about Flutter Web urls? Should it have some url parsing to demonstrate the concept? Should it use simple Navigator.push? What is important for routing in a template?

Navigator 2.0

Pros

- Supports Flutter web urls
- Some navigation flows only possible with Navigator 2.0[54]
- Provides greatest control over the navigation stack[54]
- Support for deep linking
- Supports different navigation stacks depending on device size

Cons

- Learning Journey teaches Beginners Navigator 1.0 with push and pushNamed[5]
- Evaluations reveal supporting navigator 2.0 requires ~180 LOC[52]
    - Most evaluations with Navigator 1.0 contain fewer than 180 LOC for the entire app[8]
- Asked to hold off for now by Flutter team

Navigator.push

Pros

- Learning Journey teaches Beginners Navigator 1.0 with push[5]
- Pass data directly to Widget. No need to extract from URL or Route Arguments.[10]

Cons

- Not requested by any community member in feedback thread [55]
- Partial Support for Flutter web urls
    - May pass name to RouteSettings to update Flutter web urls[23]
    - But user cannot navigate directly to such a url[23]
- No support for deep linking

Named Routes Table

Pros

- Learning Journey teaches Beginners Navigator 1.0 with push and pushNamed[5]
- Supports Flutter web urls[55]
- Demonstrate how to pass information to a named route via arguments[17]

Cons

- Unable to parse URL for critical information, e.g. extract id from /entity/{id} url
- Extracting data from arguments may be unintuitive
- Was not requested by community [55]
- No support for deep linking
- Not a path to a larger app (if the user needs any of the features below, they need to reimplement — the named routes table has a feature wall)

onGenerateRoute with static / and /entity endpoints

Pros

- Most popular option according to community feedback[55]
- Supports Flutter web urls[23]
- Demonstrate how to pass information to a named route via arguments[17]
- Teach intermediate developers about onGenerateRoute
- Requires only 2 lines of code

Cons

- No way to navigate to /entity/{id} on Flutter web
- Does not provide any capabilities beyond Named Routes Table
- Does not parse URL for critical information, e.g. extract id from /entity/{id} url
- Extracting data from arguments may be unintuitive
- No support for deep linking
- Should we encourage Navigator 1.0 usage?

onGenerateRoute that supports /entity/{id}

Pros

- Second most popular option according to community feedback[55]
- Fully supports Flutter web urls and normal web navigation[23]
- Demonstrate how to pass information to a named route via url parameters[17]
- Teach intermediate developers about onGenerateRoute
- Support for deep linking
- Requires 7 lines of code

Cons

- Should we encourage Navigator 1.0 usage?
- Extracting data from url may not be necessary for medium-sized apps.
- (minor) The specific extraction code will need to be deleted by the developer 99% of the time.

Decision

onGenerateRoute with static / and /entity endpoints

How should objects be constructed and passed down the tree?

There are many ways to pass values from parent to descendant Widgets. Passing via constructors can be educational for newer Flutter developers, but many projects appear to include some kind of library like Provider, Riverpod or get_it to make such passing easier. Should the template provide concrete guidance on how to pass dependencies down the Widget tree, or should the template pass dependencies via normal Class constructors?

Plain Dart instances and constructors

Pros

- No 3rd party libraries, which is community preference[43]
- Most popular option according to feedback thread[42]
    - "I think it makes more sense to use constructors, to keep it simple and easier to understand for people new to flutter."[42]
    - "I totally feel like that a template should have an agnostic approach on dependencies. Since this is targeting beginner-intermediate folks I feel like it would be counter productive to introduce an opinionated tool that hide all the complexity."[42]
    - "I think that we should avoid community solutions here, and should use plain Dart constructors for an example. This will allow to don't add an extra layer of complexity for newcomers."[42]
    - "You probably also don't want to "pick a winning team" by showcasing a particular library."[42]
- Raw InheritedWidgets may be confusing, having been described by beginner / intermediate Flutter developers as "Flutter's Monad" [56]

Cons

- Requires refactoring deeply nested Widget trees if a dependency is added or removed
- Intermediate Developers should be familiar with Provider from Beginner learning journey[5]
- May not provide enough guidance to intermediate developers
    - "I'm not sure what the usefulness of this template would be if it doesn't produce something that is readily usable and practical." [42]
- Might introduce barrier to refactoring (every new widget needs constructor boilerplate)

Community Solution, such as Provider, Get_it, or Riverpod

Pros

- Provider taught as part of beginner learning journey [5]
- Provide concrete guidance
- Common requirement for applications
    - “I’m not sure what the usefulness of this template would be if it doesn’t produce something that is readily usable and practical.” [42]

Cons

- Community disagreement about best solution for dependency injection
    - "You’re right that in many cases these implementations would have to be removed/replaced, which is why Riverpod may be a nice middle-ground due to it being more easily replaceable than other solutions"[42]
- Feedback indicates folks prefer if we do not include any dependencies [43]
    - "You probably also don’t want to ‘pick a winning team’ by showcasing a particular library"[42]

Decision

- let's start with: Plain Dart instances and constructors
- let's be open to Provider

Feature-first or Function-first folder structure?

There are generally two major approaches to structuring code bases: Feature-First or Function-First. From your experience working on Flutter projects, what has been the most effective or useful?

Feature-first

e.g. /search/ folder contains all code related to the “Search Feature” – models, views, controllers, etc

Pros

- "Discoverability: When looking for a class, it is usually easier to build a mental model around features versus layer (naming, layer separation, and other architectural decisions are always very opinionated). It’s usually easier to categorize it by a feature than the specific layer."[57]
- "Folder Size: Sorting by function usually results in big folders containing a lot of files, and therefore making it harder to navigate"[57]
- "What gets modified together is together"[57]
- Favored by everyone who responded to feedback thread[57]
- Most common approach taken by other frameworks, such as Ember and Angular[58]

Cons

- Some types of code do not fit neatly into a feature-first approach (utility classes, etc)

Function-First

e.g. /controllers/ folder contains controllers for all features, /models folder contains models for all features.

Pros

- Some types of code do not fit neatly into a feature-first approach (utility classes, etc)

Cons

- Need to navigate several folders to work on one feature[57]
- More compact folders[57]
- Noone who responded preferred this approach[57]

Decision

- feature-first, very lax
    - this is a big reason to have 2 features and not just one

Should the template include a folder for assets?

Yes, with sample image

Pros

- Conform to requirements to provide concrete advice to intermediate devs[36]
- Community preferred solution according to feedback thread[59]
- Demonstrate how to reference assets in code[59]
- Removes need to read Flutter documentation to create correct folder structure manually[59]
- Other templates, such as Create React App and Android Templates, include basic assets[3, 59]

Cons

- Everyone who uses the template will need to remove the assets and references

Yes, empty folder(s)

Pros

- Conform to requirements to provide concrete advice to intermediate devs[36]

- Demonstrate how to reference assets in code[59]
- Removes need to read Flutter documentation to create correct folder structure manually[59]
- No need to remove any assets or references

Cons

- Does not demonstrate how to use assets in Dart code
- Community preferred solution with image asset[59]

Instructions Only

Pros

- Developer may arrange assets in folders that make sense to them

Cons

- May not provide enough concrete advice to intermediate devs[36]
- Least favorite solution according to community feedback[59]
- Folks must read documentation to use assets correctly

Decision

Yes, and leaning to having a (tiny!) asset there. Like a png of dash.

- have 3x 2x variants

What kind of tests should the template feature?

What kind of tests are useful in a template? Does it make sense to include tests at all? If so, what kind are helpful in a template?

No Tests

Pros

- Nothing for the developer to change or remove

Cons

- Least favorite option according to community[59]
- Does not provide any guidance to intermediate developers
- May cut against requirements to provide concrete guidance[36]

Placeholder tests

Pros

- Show differences between types of Tests: Unit, Widget, Integration[60]
- Most popular option according to community[60]
- Demonstrate file structure for testing
- Does not break when user modifies template

Cons

- Developer needs to delete or rename placeholder files & tests

Implementation Tests

Pros

- Demonstrate how to test implementations of a Controller, Widget & Service
- Show differences between types of Tests: Unit, Widget, Integration[60]

Cons

- Tests invalid after modifying app code
- Developer likely needs to rename or delete tests and files

Decision

Placeholder tests

Should the template set up logging?

Logging is one of those things every app will need sooner or later. Should it be set up in the template? To what extent?

Yes

- Community in favor of logging solution[61]
  - "Since most projects will need logging, I think it should be included with a simple example."[61]
  - "This is totally needed, we are struggling on implement this on a mature app."[61]
- Potential demonstration of Services or Repositories[61]
  - "Perhaps a logging abstraction over print that can be easily overridden?"[61]

No

- Community has said “yes” to everything: Logging, assets, tests, etc. Need to cut some things?[59, 60, 61]

- Many feedback posts reference “example” rather than template
    - "Yes to logging, possibly a complete but simple example on how to set it up correctly"[61]
- Different users had various use-cases in mind for logging. Different use-cases require different implementations. Difficult to abstract into a templatable solution. Example uses cases:
    - Replacement for print statements
    - Crash reporting
    - Analytics
    - Log aggregation
    - Capture logs from libraries
- Different Crash Reporting Libraries follow different instantiation paths that cannot be captured by a generic template
    - https://pub.dev/packages/sentry
    - https://firebase.flutter.dev/docs/crashlytics/usage

Decision

No Logging.

Should the template set up responsive breakpoints?

From requirements:

> The new template MUST support the following targets: mobile, desktop and web. The template app SHOULD look and behave naturally on all these platforms. This includes things like layout, touch/mouse/keyboard input, and platform-specific idioms (e.g. navigation). It MAY have an explicit responsive breakpoint (as opposed to "just" using things like Flex), to teach people how to do such a thing.

Yes

- Demonstrate usage of LayoutBuilder to handle breakpoints
- Basic ListView + ListTile looks a bit “empty” on large screens
- Common requirement for many apps

No

- Depending on design
    - LayoutBuilder needs to be repositioned in the Widget hierarchy
    - Descendants need to be refactored
    - Common changes, such as repositioning the AppBar, require a refactor
    - In summary, LayoutBuilder code may be more hindrance than help.
- Using Navigator 1.0, the template cannot handle Navigation Stack changes on resize
- Increases code size by ~150% (157 vs 238 LOC)

Decision

No Breakpoints.

Additional questions

- State restoration - leaning NO
- Background processing - NO
    - would be thrown away immediately
- Serialization and deserialization - NO
    - would be thrown away immediately
- Sending events to an external service such as Crashlytics or Sentry.io. - NO
    - would be thrown away
- Stored user preferences.
    - TBD
- Light vs Dark mode - using system default - YES
- Light vs Dark mode - allowing the user to set - NO
    - has a lot of dependencies/assumption
- LTR / RTL - leaning NO
- Notifications - NO
- Licenses page - NO

References

1. Is it really a new template we need?
2. New Flutter Template: Interview 2 (restricted access for privacy purposes)
3. New Flutter Template: Research and Competitive Analysis
4. Flutter Kostlivec
5. Flutter Learning Journeys
6. New Flutter Template: Interview 1 (restricted access for privacy purposes)
7. Journal MVC Evaluation
8. Line Counts for Evaluations
9. Tiny Journal MVC Evaluation
10. Hacker News Vanilla Evaluation
11. Login MVC Evaluation
12. Feature-first or Function-first folder structure?
13. list_detail_mvc evaluation
14. list_detail_mvc_cubit evaluation
15. list_detail_mvc_dummy evaluation
16. list_detail_mvc_mobx evaluation
17. list_detail_mvc_provider evaluation
18. list_detail_mvc_riverpod evaluation
19. list_detail_mvc_state_notifier evaluation
20. list_detail_mvc_value_notifier evaluation

21. list_detail_mvc_vn_prettier evaluation
22. list_detail_mvc_vn_responsive_dummies evaluation
23. list_detail_mvc_vn_two_features evaluation
24. list_detail_mvc_vn_two_features_boilerplate evaluation
25. list_detail_mvc_vn_two_features_boilerplate_getit evaluation
26. list_detail_mvc_vn_two_features_boilerplate_impl evaluation
27. list_detail_mvc_vn_two_features_boilerplate_provider evaluation
28. list_detail_with_data_layer evaluation
29. list_detail_with_some_boilerplate evaluation
30. list_detail_mvc_getx evaluation
31. list_detail_breadcrumbs evaluation
32. Simple app state management
33. MVC? MVP? MVVM? Naming is hard!
34. What Architecture should the app employ?
35. Should the template setup localizations?
36. A more complete app template for Flutter — requirements
37. Should the new template include an analysis_options.yaml?
38. Pedantic | Dart Package
39. Pedantic Dart
40. I wish to remove implicit downcasts by default
41. Getting started: Creating your Flutter project
42. How should objects be constructed and passed down the tree?
43. What libraries or packages should be used?
44. “Service” or “Repository” – Naming is hard!
45. MVC? MVP? MVVM? Naming is hard!
46. What Architecture should the app employ?
47. The Repository-Service Pattern with DI and ASP.NET Core
48. Are you saying that my code is boring? Thank you!
49. The Clean Code Blog
50. What Uncle Bob Gets Wrong
51. Android-CleanArchitecture Project
52. Clean Architecture Evaluation
53. Flutter Favorite Program
54. Learning Flutter’s new navigation and routing system
55. What kind of Routing should be supported?
56. Flutter Tutorial: Pros and Cons of popular State Management Approaches
57. Feature-first or Function-first folder structure?
58. Angular coding style guide
59. Should the template include a folder for assets?
60. What kind of tests should the template feature?
61. Should the template set up logging?