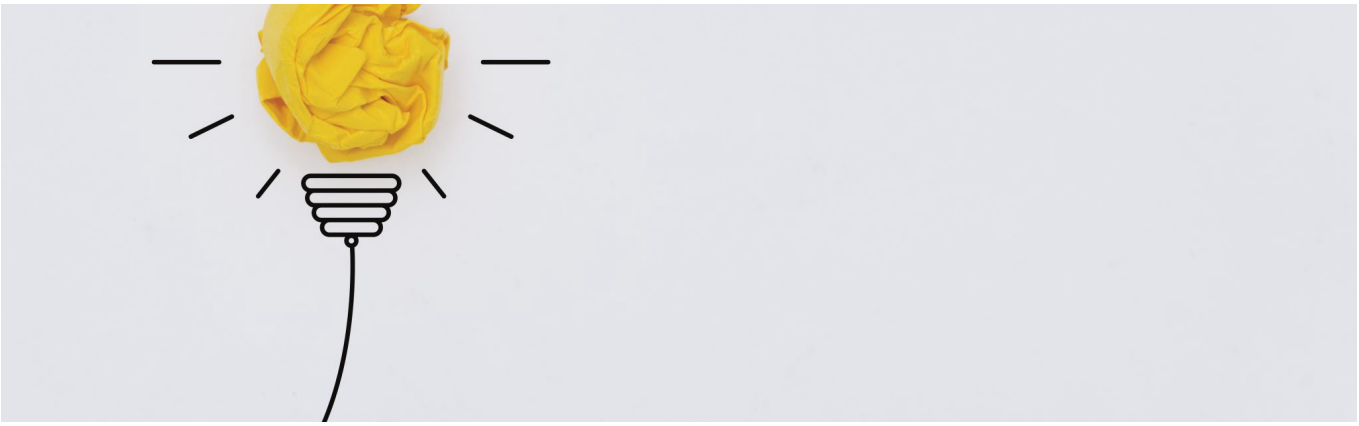

Flutter bootcamp idea bucket

Bootcamp Topics & App Challenges

Beginner Topics

Introduction

Module description: familiarity with basic dart and how to run a Flutter app

Core topics (the current module will get you up to speed with these topics):

- What is Dart?
  - Dart overview - doc
- What is Flutter? Why Flutter?
  - Why Flutter: Pros and cons - article
- Installation & Setup and development environment
  - Get started & install - doc
  - Install Dart SDK - doc
  - Set up an editor (Android Studio, VSCode) - doc
  - IDEs and editors - doc
  - Flutter Version Management – tool
  - Flutter builds release channels - doc

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Anatomy of a Flutter app
    - Anatomy&Architectural overview - doc
    - Flutter app structure: widget and layout design - article
    - E2E Flutter guide - article
- Working with Flutter hot reload (move this to Business card?)

Specific technical topics (must have, you need this to complete this module):

- Null safety
    - Null safety in Flutter - doc
    - Null safety migration guide - doc
    - Understanding null safety - doc
    - Comprehensive guide - article
    - Null safety intro - video
    - Sound null safety migration - video
- Function Name Overloading vs Optional Function Parameters
    - Effective Dart - Members - doc
    - Effective Dart - Parameters - doc

Additional technical topics (you are encouraged to look at these to enhance knowledge):

- Dart official page - homepage
- Dart comprehensive overview - doc
- Effective dart - doc
- Dart Core libraries - doc
- Dart language tour (comprehensive overview) - doc
- Widget Overview Series - video
- Android Studio vs VS Code - article

Cheat sheet/summary (a quick reference to what you have learned here):

- Dart cheat sheets
    - DCS – variant 1
    - DCS – variant 2
    - DCS – variant 3
    - DCS – variant 4

Challenge (do this to get hands-on experience to what you have learned): Running the app on Physical Devices

Create Flutter Apps from Scratch – I Am Rich app

Module description: In this module you will create a clone of the I Am Rich app using some assets provided during the course. (we'll provide a sample screenshot of the target app)

Core topics (the current module will get you up to speed with these topics):

- How to create a new project
    - Create a Flutter project
    - Info:
        - Your first Flutter app (google.com) ← Codelab Page 2
- Material and Cupertino widgets
    - Check out the usable widgets
    - Read about differences of Material and Cupertino widgets
    - Info:
        - Cupertino or Material: Which is the Best Flutter Widget? (dhiwise.com)
- Scaffolding a Flutter App
    - Learn how to customize a Scaffold for the home page of the new project
    - Set a background color

    - Info:
        - Know Your Widgets: Scaffold in Flutter | by Aditya Syal | FlutterDevs
        - Scaffold class - material library - Dart API (flutter.dev)
- Working with Assets
    - Add the assets attached to this page into your project
    - Info:
        - Adding assets and images | Flutter
- Adding Native Icons
    - Change the icons of the app using the ones attached to this page
    - Read:
        - Change Flutter App Launcher Icon. New flutter app comes with a flutter… | by Hussain Habibullah | Flutter Community | Medium

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Dynamic App Icons in Flutter: Ultimate Guide | by Pavel Ryabov | Medium

Specific technical topics (must have, you need this to complete this module):

- Setting up the environment:
    - Your first Flutter app (google.com) ← Codelab page 1
- Material design & widgets
    - Material Components widgets - doc
- Cupertino design & widgets
    - Cupertino widgets - doc

Additional technical topics (you are encouraged to look at these to enhance knowledge):

- Material design - doc
- Material design guidelines - doc

Cheat sheet/summary (a quick reference to what you have learned here):

Challenge (do this to get hands-on experience to what you have learned):

Challenge: Create I Am Poor app by yourself ☺

- Create the same app but with different assets and darker background on your own.
- Ask for review from our mentors

Building Beautiful UI with Flutter – Business Card App

Module description:

Core topics (the current module will get you up to speed with these topics):

- Using Containers
  - Containers in Flutter - article
  - The Container class – doc
- Using Column & Row, Flexible Layout
  - Row, Column and Flex explained - article
  - Layouts in Flutter – doc
- Widget properties
  - Widgets & properties - article
- Flutter Card & ListTile widgets
  - ListTile class – doc
  - Card class - doc

Additional recommended topics (we recommend you read up on this to do a deeper dive):

Specific technical topics (must have, you need this to complete this module):

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

- Widgets & layout cheat sheets - article

WIDGET PROPERTY SUMMARY

CORE WIDGET PROPERTIES

- **key:** A unique identifier for a widget within its parent hierarchy. Used for certain optimizations or maintaining state when reordering widgets.
- **child:** Nested widget within another widget (composing complex UIs).

- **children:** A list of nested widgets for widgets that accept multiple children (e.g., Column, Row).

Layout Properties

- **width/height:** Control the size of a widget.
- **padding:** Adds space around the widget's child.
- **margin:** Adds space outside the widget's boundary.
- **alignment:** Positions a child within its parent.
- **constraints:** Provides size constraints for widgets that need them (e.g., SizedBox, Container).

Appearance Properties

- **color/backgroundColor:** Set the color of a widget or its background.
- **decoration:** Adds borders, gradients, shadows (using the BoxDecoration class).
- **textStyle:** Configures text styles (font, size, weight, color).

Behavior Properties

- **onPressed/onTap:** Callback functions for button presses or taps on interactive widgets.
- **onChanged:** Callback for when a value changes (e.g., in TextField, Slider).
- **controller:** Associates a TextEditingController or ScrollController for specialized controls.

Challenge: Build your own Business Card App with your own picture

Building Apps with State – Dice App

Module description:

Core topics (the current module will get you up to speed with these topics):

- Stateless & Stateful Widgets
  - Building interactivity - doc
  - Stateful vs Stateless widget explained - article
- Buttons and Gesture Detection
  - Flutter Gestures - doc
- Randomizing the dice value and updating the UI
- ChangeNotifier
  - Flutter State management with ChangeNotifier - article
  - Simple app state management - doc

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Flutter gesture detection - video
- Handling gestures - article

Specific technical topics (must have, you need this to complete this module):

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

KEY POINTS:

- **State**: Data that changes during the lifetime of your app.
- **ChangeNotifier**: Mechanism to notify widgets about state changes.
- **Provider**: Package to make state easily accessible across the widget tree.
- **GestureDetector**: Widget for detecting common gestures.

EXAMPLE SCENARIO:

- A button that increments a counter displayed on the screen.
- Use GestureDetector to wrap the button.
- In the onTap handler, update the state in your ChangeNotifier model.
- Widgets using Consumer will automatically rebuild to reflect the change.

Challenge: change the app to store the state in ChangeNotifier instead of the widget

Using Packages – Xylophone App

Module description:

Core topics (the current module will get you up to speed with these topics):

- Flutter & Dart packages
    - Packages and Plugins – doc
    - Using packages - doc
    - Package Manager - video
- Picking an audio player package
    - Audio player packages in Flutter - doc
    - Flutter Audio Service - article
    - Audio Service package – pub.dev
- Playing sound across platforms
- Creating the UI

Additional recommended topics (we recommend you read up on this to do a deeper dive):

Specific technical topics (must have, you need this to complete this module):

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

Challenge: Customize the app with own sounds

Intermediate Topics

Flutter State Management – Funny Cards App

Module description:

Core topics (the current module will get you up to speed with these topics):

- Ephemeral state, local state & application state
  - State management introduction - doc
- Provider
  - Flutter Provider guide - article
  - The Provider package – pub.dev
- BLoC Basics
  - BLoC state management in Flutter - article
- Riverpod Basics
  - Riverpod for beginners - article
  - Riverpod – getting started - doc

Additional recommended topics (we recommend you read up on this to do a deeper dive):

Specific technical topics (must have, you need this to complete this module):

PROVIDER:

- Architecting app with Provider - article

BLOC & CUBIT:

- Flutter BLoC - doc
- Flutter BLoC – github
- Cubit and BLoC essentials - doc
- Flutter bloc for beginners - article
- Architecting app with BLoC - article
- BLoC architecture - doc

RIVERPOD:

- Riverpod – official site
- App architecture with Riverpod - article
- Exploring Riverpod and Building a Todo App - Video
- State Management Like a Pro. Flutter Riverpod - Video

COMPARISON:

- Bloc vs Riverpod - article
- Bloc vs Provider - article

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

Challenge 1: Implement the app using BLoC

Challenge 2: Implement the app using Riverpod

Animations with Flutter – Animated Funny Cards App

Module description:

Core topics (the current module will get you up to speed with these topics):

- Styles & themes
    - Themes and styles intro - doc
    - Theme widget - doc
    - Flutter theming guide - article
- Animations with Flutter
    - Design Language in Programming - video
    - Making Animations in Flutter - video
- Animating the change between cards

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Custom themes and theme extensionsarch - article

Specific technical topics (must have, you need this to complete this module):

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

Challenge: Add the Flip Cards package to reveal the back of the cards

Powering the app with real-time data – Funny Cards with jokes from an API

Module description:

Core topics (the current module will get you up to speed with these topics):

- Accessing Rest and other kinds of API's with Dart
    - Consume REST API in Flutter - article

- Http package
    - Http package reference - pub.dev doc
    - Fetch data from the internet - doc & example
- Dio Package
    - Dio package reference – pub.dev doc
    - Exploring the power of Flutter Dio - article
- Modeling data with Dart classes
    - Basics of creating models in Flutter - article
- JsonSerialization
    - JSON and serialization - doc
    - Encode/decode JSON - article
    - Parse and serialize primer - article
- Displaying the data

Specific technical topics (must have, you need this to complete this module):

- JSON handling with json_serializable **- doc**

Cheat sheet/summary (a quick reference to what you have learned here):

NETWORKING

- **http package**: The foundation for most network requests in Flutter (https://pub.dev/packages/http)
    - **Key functions**:
        - http.get(uri): Perform a GET request.
        - http.post(uri, body): Perform a POST request.
        - http.put(uri, body): Perform a PUT request.
        - http.delete(uri): Perform a DELETE request.

- **dio package:** A more versatile HTTP client (https://pub.dev/packages/dio)
    - **Benefits**: Interceptors (for logging, auth, etc.), easier error handling, download progress.

        - dio.get(): Perform an HTTP GET request
        - dio.post(): Perform an HTTP POST request
        - dio.put(): Perform an HTTP PUT request
        - dio.delete(): Perform an HTTP DELETE request
        - Options: Options to configure headers, content-type, etc.
        - QueryParameters: Map of query parameters to append to the URL
        - Response: Object containing response data, headers, status code
    - **Important Notes**:
        - Error Handling: Use try-catch blocks or the response.statuscode to handle request errors.
        - Interceptors: Explore the power of dio.interceptors.add().
        - Transformers: Use dio.transformer for automatic response data handling.

Data Modeling

- **Built-in Decoding**: jsonDecode(jsonString): Convert raw JSON string into Dart Maps and Lists.
- **Data Classes**: Define custom classes to mirror your data structure:

Important Considerations

- **Choose the right HTTP client**: http for basic needs or dio for more power.
- **Error Handling:** Always implement robust error handling for network requests.
- **State Management:** Integrate with your chosen state management solution (Provider, BLoC, Riverpod, etc.).

Challenge: incorporate repository layer either with Riverpod or with plain Dart implementation

Navigation in Flutter apps – Adding menu to Funny Cards App

Module description: Construct seamless and intuitive user flows within your Flutter applications. This module summarizes the essentials of navigation and routing, enabling you to create well-structured, dynamic user interfaces.

- **Learning Objectives:**
    - Understand the difference between basic navigation and named routes.
    - Use the Navigator to push and pop routes.
    - Pass data between screens during navigation.
    - Implement named routes for organized and scalable navigation.
    - Handle back button behavior to maintain a predictable user experience.

Core topics (the current module will get you up to speed with these topics):

- Routing, Named routes, Static routes
    - Navigation basics - doc
    - Navigation overview - doc
    - Named routes - doc
- Routing packages
    - go_router
    - auto_route
    - beamer
    - fluro
- Adding a menu drawer
- Navigating to routes

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Comprehensive Navigation Guide - article
- Understanding navigation and routing - article
- Flutter routing deep-dive - article

Cheat sheet/summary (a quick reference to what you have learned here):

- Flutter Named Routing Cheat sheet
- Navigate in Flutter - video

Challenge: Add all implementations (Provider, Bloc and Riverpod) together in one app and make them selectable from the menu

Using Firebase Firestore with Flutter – FlashChat app

Module description: In this module, you'll learn how to integrate Firebase Firestore, into your Flutter applications. You'll use the StreamBuilder widget to display and update data in real-time as changes occur in your Firestore database.

Learning Objectives:

- Understanding Firebase Firestore:
- Setting up a Firebase project and integrating it with Flutter
- Basic Firestore Operations and using StreamBuilder for presenting data

Core topics (the current module will get you up to speed with these topics):

- Building the pages and the UI
- Creating a Firebase project
    - Firebase in Flutter - doc
- Android Firebase Project setup
    - Android setup - doc
- iOS Firebase Project setup
    - IOS setup - doc
- Authentication with FirebaseAuth
    - Authentication startup guide - doc
- Listening for data using streams
    - Realtime updates on data - doc

- Using StreamBuilder
    - StreamBuilder in Flutter - article
    - Streambuilder example - article

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Firebase Firestore startup guide - article
- Connecting Cloud Firestore to a Flutter app - article
- Streams and StreamBuilder - article
- Firestore & Streambuilder - video

Specific technical topics (must have, you need this to complete this module):

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

- Firebase CheatSheet in Flutter - article

Challenge: Add either Bloc layer or Riverpod controller layer to handle the stream changes and notify the UI

Advanced Topics

Platform channels in Flutter

Module description: Platform channels act as a bridge between your Flutter code (written in Dart) and the native code of the target platform (Android's Java/Kotlin or iOS's Objective-

C/Swift). They enable you to leverage platform-specific functionalities that Flutter itself might not provide directly

Core topics (the current module will get you up to speed with these topics):

- Core technology and architecture
    - Flutter platform channels - doc
    - Detailed technical explanation - article

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Example projects
    - Example proj 1
    - Example proj 2
    - Example proj 3

Cheat sheet/summary (a quick reference to what you have learned here):

Challenge: Fitness Tracker App with Native Pedometer Integration. This app tracks the user's steps using platform channels to access the native pedometer functionality of Android and iOS devices.

Working with Flutter embedders

Module description: A Flutter embedder is the bridge between the Flutter engine (written in C++) and your target platform. It is the component that allows Flutter's rendering, input, and platform interaction mechanisms to function on non-standard platforms that Flutter does not directly support out-of-the-box.

Reasons to Create an Embedder:

- Embedding Flutter in Existing Apps: You might want to integrate Flutter components into an existing native application on a particular OS.
- Novel Platforms and Devices: Bringing Flutter to embedded systems, new operating systems, game consoles, etc.
- Special Use Cases: Need very fine-grained control over how Flutter interacts with the underlying system (e.g., custom graphics backends).

Core topics (the current module will get you up to speed with these topics):

- Role and Concept of Flutter embedder - article
- Embedder API - API declaration
- Custom Flutter Engine Embedders - wiki
- Flutter on embedded Linux target - article

Cheat sheet/summary (a quick reference to what you have learned here):

Key C++ Classes

- FlutterEngine: The core of the embedder API. You'll instantiate and manage this.
- FlutterRenderer: Manages rendering tasks.
- FlutterPlatformMessage: Represents a message passed between Dart and the platform.
- FlutterPlatformMessageResponseHandle: Used for handling responses to platform messages.
- FlutterProjectArgs: Configuration data passed while initializing the FlutterEngine

Crucial Functions

- FlutterEngineRun: Starts the engine and runs rendering, event loops.
- FlutterEngineShutdown: Cleanly stops the engine.

- FlutterEngineSendWindowMetricsEvent: Notify engine of display size changes.
- FlutterEngineSendPointerEvent: Forward touch/mouse events to the engine.
- FlutterRendererRender: Trigger a rendering frame.

Essential Setup Steps

- Include Flutter headers (e.g., #include "flutter/embedder.h")
- Create a FlutterProjectArgs struct, and configure:
    - assets_path
    - icu_data_path
    - vm_snapshot_data
    - isolate_snapshot_data
- Instantiate a FlutterEngine.
- Attach a FlutterRenderer.
- Run the engine (FlutterEngineRun).

Platform Integration Hints

- Rendering: You'll likely interface with low-level graphics APIs (OpenGL, Skia, Metal, Vulkan, etc.)
- Input: Map native input events from the platform to Flutter events.
- Message Passing: Set up channels to handle messages between your embedder and the Dart code

Challenge: N/A

Architectural and Design patterns in Flutter and Dart

Module description: Architectural and design patterns in Dart offer blueprints for organizing your Flutter application's code, promoting maintainability, testability, and scalability as it grows.

Core topics (the current module will get you up to speed with these topics):

- Design patterns in dart:
  - Collection of applicable design patterns - git repo
- Flutter architecture samples:
  - Collection of Flutter architecture samples - git repo
- Flutter's layered architecture
  - Architectural layers - doc
- Other
  - SOLID principles in dart - article

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Clean architecture in Flutter
  - Course about applying clean architecture to Flutter app development - video
  - Clean architecture, a practical example - article
- Architecting Flutter app with Riverpod
  - Production ready architecture including Firebase and Riverpod - article
- Architecting Flutter app with BLoC
  - Applied example for Flutter app with BLoC - article

- Architecting Flutter app with Provider and Stream
    - Insight in using Provider and Stream - article
- Large scale applications
    - Architecting large scale Flutter apps - article
    - Large scale Flutter app best practices - article
- General architecture
    - Flutter architectural components - video

Cheat sheet/summary (a quick reference to what you have learned here):

Key Architecture Layers

FRAMEWORK: (Highest level) Flutter's core rendering, widgets, gestures, and foundational elements.

ENGINE: (C++ level) Skia graphics engine, text rendering, platform channels, etc.

EMBEDDER: (Lowest level) Adapts the Flutter engine to your specific hardware/OS platform.

Commonly Used Patterns

BLoC (Business Logic Component)

Purpose: Separates business logic UI. Enforces state management and testability.

When to Use: Medium to complex apps where you need a clear division between data, logic, and presentation.

Provider

Purpose: Simple state management via dependency injection, making data accessible throughout different parts of the widget tree.

When to Use: Smaller apps or managing state that doesn't require complex transformations or logic.

Model-View-Controller (MVC)

Purpose: Separation of concerns. Model (data), View (UI), Controller (handles interactions)

When to Use: When you prefer the familiar MVC structure but use a Flutter specific implementation.

Scoped Model

Purpose: Similar to Provider, but with a focus on rebuilding only the widgets that depend on specific parts of the model.

When to Use: When you want easy state management with some optimization features.

Explanation: Scoped model example - video

Challenge: N/A

Memory management

Core topics (the current module will get you up to speed with these topics):

- Memory management best pactices - article
- Using the memory view - doc
- Leak detector package – tool
- Flutter load sequence, performance and memory – doc

Flutter memory management summary:

Key Principles

- Automatic Garbage Collection: Trust Dart's garbage collector to reclaim unused memory for you.
- State Management: Choose the right state management solution (Provider, BLoC, etc.) to prevent excessive object creation and unnecessary widget rebuilds.
- Dispose of Resources: Religiously call dispose () on widgets (like StatefulWidgets and controllers) and other objects when they are no longer needed.

Best Practices

- Immutability: Use immutable data structures whenever possible for efficient memory sharing.
- Weak References: Break reference cycles that prevent GC using WeakReference in appropriate scenarios.
- Minimize Widget Tree Footprint: Avoid unnecessarily storing large data within the widget tree.
- Optimize Image Handling: Use image caching libraries like cached_network_image to avoid repeated downloads.
- Manage Streams: Close or cancel Stream subscriptions when no longer needed.

Tools & Techniques

Tool/Technique	Usage
DevTools Memory Profiler	Track memory usage over time, identify potential leaks or hotspots.
Dart Observatory	Deep dive into memory, heap snapshots, object inspection.
dispose () method	Release resources (timers, controllers, etc.) associated with widgets and objects.
Stream cancel ()/close ()	Prevent StreamControllers and subscriptions from holding onto references unnecessarily.

Common Optimization Scenarios

- Large Lists: Use ListView.builder to render only visible list items. Consider pagination for huge datasets.
- Complex Animations: Be mindful of how frequently animations rebuild widgets. Pre-calculate where possible.
- Offscreen State: If data is needed only when visible, consider removing widgets from the tree when offscreen to aid GC.

Remember

- In most cases, Dart's garbage collector does a good job. Focus on addressing the major causes of memory issues.
- If you encounter memory issues, use Flutter DevTools and the Observatory to pinpoint the root cause.

Serverpod, the Flutter backend solution

Module description: Serverpod is an open-source app server designed specifically for the Flutter development community. It offers several key features that make it attractive for building backend functionality for Flutter apps.

Key features:

- **Written in Dart**: Serverpod leverages the Dart language, and this allows for a consistent development experience and simplifies code sharing between your front-end and back-end.
- **Automatic Code Generation**: Serverpod automatically generates code for your server-side APIs based on your server code.

- S**eamless Database Integration**: Serverpod integrates well with popular databases, allowing you to easily connect your backend to store and retrieve data.
- **Built-in Features:** Serverpod includes various functionalities out of the box, such as user authentication, file uploads, caching, and WebSocket support, eliminating the need to implement these features from scratch.
- **Focus on Flutter familiarity**: Serverpod is specifically designed with Flutter development in mind. following Flutter best practices and conventions.

Core topics (the current module will get you up to speed with these topics):

- Installing and get started - doc
- CRUD operations with Serverpod - article
- Mastering API development with Serverpod - article
- Authentication - doc
- Authentication step-by-step guide - article

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Serverpod source code - github
- Initial Setup - video
- Data CRUD - video
- Building a data layer - video
- Diving into Serverpod - video
- Creating a journal app - video
- Serverpod on github
- Serverpod VSCode plugin – tool

Cheat sheet/summary (a quick reference to what you have learned here):

Installation

- Install Serverpod: pub global activate serverpod
- Create a Project: serverpod create my_project

Project Structure

- endpoints/: Contains your endpoint files (.dart files).
- generated/: Holds Serverpod-generated code for communication and serialization.
- lib/: Your standard Flutter project structure within the Serverpod project.
- pubspec.yaml: Dependencies, including serverpod.
- serverpod.yaml: Serverpod configuration.

Key Commands

- serverpod run: Starts your development Serverpod server.
- serverpod generate: (Re)generates serialization and communication code.
- serverpod install: Installs Serverpod's database schema into your configured PostgreSQL database.

Important Notes

- **Authentication**: Serverpod has built-in mechanisms for authentication and authorization.
- **Caching**: Explore Serverpod's caching system for performance optimization.
- **Deployment**: Serverpod can be deployed to various cloud providers or even run on self-hosted machines.

Challenge (consider implementing any of these):

- **Quiz App**: Store questions, answers, and categories in the database. Implement endpoints to fetch questions, check answers, and track scores.
- **Simple Multiplayer Game**: A basic board game like Tic-Tac-Toe . Serverpod maintains game state and handles player turns.

Direct native interaction with JNI and FFI

Core topics (the current module will get you up to speed with these topics):

- Understanding Use Cases
- Setting Up the Native Side
- JNI
    - Building jnigen - article
- FFI
    - Foreign Function Interface - article
- Bridging the Language Gap
- Complexity and Considerations
- Choosing the Right Approach

Additional recommended topics (we recommend you read up on this to do a deeper dive):

Cheat sheet/summary (a quick reference to what you have learned here):

JNI (Java Native Interface)

Purpose: Interacting with existing native libraries written in C/C++ from Flutter (Dart) code.

Pros:

- Access mature, performance-critical native libraries.
- Potentially better performance for computationally heavy tasks within the native domain.

Cons:

- More complex setup - involves writing bindings with the Java Native Development Kit (NDK), working across languages.
- Steeper learning curve, especially if unfamiliar with C/C++.
- Adds overhead on every call between Dart and native code.

FFI (Foreign Function Interface)

Purpose: Directly calling native C libraries from Dart code.

Pros:

- Simpler setup than JNI as it works directly with Dart.
- Can be faster for simple interactions, as it avoids some JNI overhead.
- Easier if already familiar with C.

Cons:

- Less mature in Flutter, potential for platform compatibility issues.
- Limited support for complex data structures, making it trickier for object-oriented paradigms.

Comparison sheet:

Feature	JNI	FFI
Use Case	Access to large, mature C/C++ libraries	Calling simple C functions or smaller libraries
Setup	More involved, uses the NDK	Simpler, dart:ffi library
Overhead	Higher for each call due to language switching	Lower for simple interactions
Complexity	Steeper learning curve	Easier for smaller integrations

When to Choose Which

JNI: Choose JNI if you need to leverage well-established, potentially performance-critical native libraries (like OpenCV for image processing).

FFI: opt for FFI when making simple calls to C code, or prototyping interactions quickly.

Important Notes:

Performance Varies: Real-world performance depends on the specific use case. Profile your implementation.

Alternatives: Consider cross-platform plugins written in Dart if a suitable one exists; they may offload most of the native interaction complexity.

Challenge: Modified BMI Calculator that has generated Dart code that calls native classes via JNI and FFI to calculate BMI

Creating Plugins for Flutter

Module description: This module dives into the specifics of Flutter plugin development. You'll learn how to extend Flutter's capabilities by creating custom plugins that communicate with native Android and iOS code. Get ready to bridge the gap between Dart and the platform-specific world.

Learning Objectives:

Flutter Plugin Fundamentals:

- Understand the architecture of Flutter plugins
- The role of platform channels and method invocations
- Dart/Native code interaction
- Plugin Development Setup:
- Project creation using Flutter CLI
- Structuring your plugin code

Android Integration (Java/Kotlin):

- Writing Android-specific code
- Exposing methods to Flutter

- Handling data communication

iOS Integration (Swift/Objective-C):

- Writing iOS-specific code
- Exposing methods to Flutter
- Handling data communication

Platform Interfaces:

- Defining a common interface for consistent cross-platform usage

Core topics (the current module will get you up to speed with these topics):

- Flutter plugin architecture explained - article
- Plugin development fundamentals - doc
- Mastering Flutter: Create a plugin - article
- Packages and Plugins in Flutter - video
- Create and publish packages and plugins - video

Additional recommended topics (we recommend you read up on this to do a deeper dive):

- Flutter plugins, comprehensive guide - article
- Plugin example - gitlab

Specific technical topics (must have, you need this to complete this module):

Additional technical topics (you are encouraged to look at these to enhance knowledge):

Cheat sheet/summary (a quick reference to what you have learned here):

Challenge: BMI Calculator app, that hosts the calculation login in native plugins

GenAI integration for Flutter

Core topics (the current module will get you up to speed with these topics):

- Gemini API in Dart/Flutter - doc
- Google AI Dart SDK – pub.dev package
- Harness the Gemini API - article
- Flutter GenAI packages - collection
- Example project using Flutter and Google AI Dart SDK - github proj
- API reference for Gemini

Challenges:

- **Language Learning Buddy**: An app where users converse with an AI to develop their language fluency.
- **Explainer:** An app where users ask questions about complex concepts, and the AI provides summarized explanations.
- **Summarizer**: Summarize complex articles, research papers, or meeting notes into easily digestible bullet points.
- **Stock analyzer:** A bulletpoint based stock analyzer: e.g. Analyze XY stock in 10 bullet points.

Conferences, certifications, recommended courses

Courses

- Flutter beginner courses

    - Variant 1 - video
    - Variant 2 - free Udemy
    - Variant 3 - article
    - Variant 4 - video
- Flutter state management
    - State management 1 - video
- Flutter animations
    - Animations 1 - video

Youtube channels

- Flutter
- Reso Coder
- Code With Andrea
- Flutter Explained
- Nick Manning
- Creative Bracket
- Robert Brunhage
- Tadas Petra
- Fireship
- Super Declarative
- Flutter Community
- creativecreatorormaybenot
- Learn Flutter Code
- Techie Blossom
- Fun With Flutter
- London App Brewery
- The Net Ninja
- Jedi Pixels
- The Flutter Way

Alternative learning plans

- Flutter learning flow – github doc
- Flutter learning roadmap – github doc

- Codelabs | Flutter

Books

- Cookbook
- Flutter in Action
- Flutter Succinctly, Syncfusion
- Flutter Tutorial
- Flutter Tutorials Handbook
- Flutter UI Succinctly, Syncfusion

Other resources

- Official Flutter feature roadmap - doc
- Flutter Cookbook - doc
- Flutter API reference - doc
- 10 must have library recommendations - article
- Awesome Flutter – github collection
- Flutter Samples - collection

Conferences

- Flutter Global summit
- Flutter Vikings

//Todo

Bootcamp Topics & App Challenges

by Marvin

Dart Language Comparison for Developers

Expression Similarities and Distinctions

- Constructor initialization order
- Return operator
- Getter and Setter Properties
- Functional Programming with Closures
- Error and Exception handling

Function Name Overloading vs Optional Function Parameters

- Language feature completeness comparison
- Null Safety introduction

Built-in Language Features and First Party Library Support

- Factory Pattern Constructor
- Builder Pattern with Method Chaining
- Extension Methods
- Pattern Matching?
- Automatic Type Promotions
- Unit and Integration Tests
- Standard Libraries and pub.dev
- Flutter

Just in Time vs Ahead of Time Compilation

- Tree Shaking Algorithm for unused Code and Build Variants
- Hot Reloading

Common Project Setups and Code Best Practices

- Stateless vs Stateful Widgets with setState
- Package, Plugin, App difference
- Library Import and Export
- FFI vs Method Channels

Example Project to Fix Bugs with Widget Inspector