FLUTTERFEVER

BASIC FLUTTER

Build your first clean apps with widgets, layout, state and APIs.

```dart
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(home: HomePage());
  }
}
```

Widgets · Layout · State · Navigation · APIs · Debugging

A practical beginner guide for developers starting Flutter the right way.

flutterfever.com
Basic Flutter • Beginner Edition

Start with the Flutter mental model

Flutter is not only a UI toolkit. It is a way of describing the screen as a tree of widgets. The framework compares the new tree with the previous one and updates the visible UI efficiently.

Beginner mental model

1 Widget tree

Every screen is built by composing small widgets into a hierarchy. Think in boxes, rows, columns and state.

2 Rebuilds

A rebuild is not a full app restart. It simply asks widgets to describe the UI again from current data.

3 Hot reload

Hot reload helps you test UI changes quickly while keeping app state whenever possible.

4 Declarative UI

You describe what the screen should look like for a given state; Flutter handles the rendering pipeline.

CODE

```
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(home: HomePage());
  }
}
```

WHY THIS MATTERS

Your first goal is not to memorize 100 widgets. Your goal is to understand how data becomes UI.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Install, run and understand the tools

Beginners usually get stuck because they install Flutter but do not understand what each command is checking. Treat the CLI as your first debugging friend.

1 flutter doctor

Checks SDK, Android toolchain, Chrome, connected devices and editor integrations.

2 flutter create

Generates a project with Android, iOS, web and desktop folders based on enabled platforms.

3 flutter run

Builds and launches the app on a selected emulator, phone, browser or desktop target.

4 pub get

Downloads packages declared in pubspec.yaml and updates the package graph.

CODE

```
$ flutter doctor
$ flutter create my_first_app
$ cd my_first_app
$ flutter run

# When dependencies change
$ flutter pub get
```

WHY THIS MATTERS

If doctor reports a missing Android license or SDK path, fix that before debugging your Flutter code.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Project structure without fear

A new Flutter project looks big because it contains platform folders. Most beginner app work happens inside lib/ and pubspec.yaml.

1 **lib/**

Main Dart source code. Start here and keep your features organized as the app grows.

2 **pubspec.yaml**

Package dependencies, assets, fonts and app metadata live here.

3 **android/ios/**

Native host projects. Touch them only when you need platform configuration.

4 **test/**

Unit and widget tests. Start small by testing pure Dart logic first.

CODE

```
my_app/
  lib/
    main.dart
    screens/
    widgets/
    models/
    services/
  assets/
  pubspec.yaml
  android/
  ios/
```

WHY THIS MATTERS

Do not put every widget in main.dart. Even beginner apps feel professional when screens, widgets and services are separated.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

MaterialApp, Scaffold and the first screen

A typical Flutter screen starts with MaterialApp for app-level setup and Scaffold for page-level structure.

1 MaterialApp

Controls app theme, routes, localization and Material behavior.

2 Scaffold

Gives a page app bar, body, drawer, bottom navigation and floating action button structure.

3 AppBar

Top navigation or title area. Keep it simple for beginner screens.

4 body

The main content area where your layout widgets live.

CODE

```
return MaterialApp(
  theme: ThemeData(useMaterial3: true),
  home: Scaffold(
    appBar: AppBar(title: const Text('Home')),
    body: const Center(
      child: Text('Hello Flutter'),
    ),
  ),
);
```

WHY THIS MATTERS

When your screen feels messy, ask: is this app-level configuration, page structure or body content?

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Widgets: the building blocks

In Flutter, almost everything visible is a widget: text, padding, rows, columns, buttons, images and even invisible layout behavior.

1 Visual widgets

Text, Image, Icon, Card, Container and Button show something visible.

2 Layout widgets

Row, Column, Stack, Expanded, Padding and Align decide where things go.

3 State widgets

StatefulWidget and state managers decide when UI should update.

4 Composition

Professional Flutter is composition: combine small widgets instead of building huge classes.

CODE

```
Widget build(BuildContext context) {
  return Padding(
    padding: const EdgeInsets.all(16),
    child: Column(
      children: const [
        Icon(Icons.flutter_dash),
        Text('Welcome'),
      ],
    ),
  );
}
```

WHY THIS MATTERS

A widget should usually do one clear job. Small widgets are easier to test, reuse and debug.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

StatelessWidget vs StatefulWidget

This is one of the first concepts every beginner must understand. The difference is not visual; it is about whether the widget owns mutable state.

1 StatelessWidget

Use when output depends only on constructor values or external state.

2 StatefulWidget

Use when the widget itself owns changing values like counter, selected tab or form text.

3 setState

Tells Flutter that local state changed and this widget subtree should rebuild.

4 Rule

Do not use StatefulWidget everywhere. Use it when the screen actually owns changing data.

CODE

```
class CounterPage extends StatefulWidget {
  const CounterPage({super.key});

  @override
  State<CounterPage> createState() => _CounterPageState();
}

class _CounterPageState extends State<CounterPage> {
  int count = 0;
  void increment() => setState(() => count++);
}
```

WHY THIS MATTERS

StatefulWidget is not bad. Unclear state ownership is bad.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Build method and BuildContext

The build method describes UI. BuildContext tells a widget where it lives in the tree and gives access to inherited information such as theme, navigator and media size.

1 build()

Should be fast and mostly pure. Do not do network calls directly inside build.

2 context

Represents a location in the widget tree, not a global app object.

3 Theme.of

Reads theme data from above the widget.

4 Navigator

Uses context to find the nearest navigation stack.

CODE

```
@override
Widget build(BuildContext context) {
  final theme = Theme.of(context);

  return Text(
    'Welcome',
    style: theme.textTheme.headlineMedium,
  );
}
```

WHY THIS MATTERS

When a context error appears, the widget may be trying to access something before it exists above that point in the tree.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Layout basics: constraints first

Flutter layout is controlled by constraints. Parents give constraints, children choose a size within those constraints, and parents place children.

Beginner mental model

1 Constraints
A child cannot be bigger than the constraints sent by its parent.

2 Size
A child chooses a size that satisfies its constraints.

3 Position
The parent decides where the child is placed.

4 Overflow
Usually means the child wants more space than the parent allowed.

CODE

```
Column(
  children: [
    const Text('Profile'),
    Expanded(
      child: ListView(children: items),
    ),
  ],
)
```

WHY THIS MATTERS

Most layout bugs become easier when you ask: who is giving constraints and who is asking for too much space?

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Row and Column without overflow

Row and Column are simple but powerful. Most beginner overflow errors come from putting wide text or large children inside Row without flexible space.

1 mainAxis

The direction in which children are placed: horizontal for Row, vertical for Column.

2 crossAxis

The opposite direction. Use alignment carefully.

3 Expanded

Takes available space and prevents many overflow errors.

4 Flexible

Similar to Expanded but allows the child to be smaller if it wants.

CODE

```
Row(
  children: [
    const Icon(Icons.person),
    const SizedBox(width: 12),
    Expanded(
      child: Text(
        user.name,
        overflow: TextOverflow.ellipsis,
      ),
    ),
  ],
)
```

WHY THIS MATTERS

If text overflows in a Row, wrap it with Expanded before reducing font size.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Padding, margin and spacing

Flutter does not have CSS-style margin as a separate property for every widget. You normally use Padding, SizedBox and Container.

1 Padding
Adds space inside the parent area around a child.

2 SizedBox
Creates fixed gaps or constrains a child to a size.

3 Container
Useful when you need decoration, color, constraints or alignment.

4 Consistency
Use repeated spacing values such as 8, 12, 16 and 24.

CODE

```
Card(
  child: Padding(
    padding: const EdgeInsets.all(16),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: const [
        Text('Order #1234'),
        SizedBox(height: 8),
        Text('Delivered'),
      ],
    ),
  ),
)
```

WHY THIS MATTERS

Professional UI often comes from consistent spacing more than from complex animations.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Stack, Align and Positioned

Stack is useful when widgets overlap: badges, banners, gradients, floating labels and image overlays.

1 Stack

Places children on top of each other in order.

2 Positioned

Controls exact edges inside a Stack.

3 Align

Places a child relative to available space.

4 Use lightly

Too many Positioned widgets can make responsive design harder.

CODE

```
Stack(
  children: [
    Image.network(imageUrl),
    Positioned(
      right: 8,
      top: 8,
      child: Badge(label: Text('New')),
    ),
  ],
)
```

WHY THIS MATTERS

Use Stack for overlays, not as a replacement for normal layout.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Lists and scrolling screens

Most real apps display repeated data. ListView.builder creates visible rows lazily, which is better for long lists.

1 ListView

Scrollable list for a fixed or dynamic collection.

2 builder

Creates items only when needed, improving performance.

3 itemBuilder

Returns the widget for each index.

4 Keys

Help Flutter preserve item identity when lists reorder or change.

CODE

```
ListView.builder(
  itemCount: products.length,
  itemBuilder: (context, index) {
    final product = products[index];
    return ListTile(
      title: Text(product.name),
      subtitle: Text('₹${product.price}'),
    );
  },
)
```

WHY THIS MATTERS

Use ListView.builder for API lists, search results, messages and feeds.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Images, assets and icons

Flutter can load local assets, network images and built-in Material icons. The most common mistake is forgetting to declare assets in pubspec.yaml.

1 Assets

Local files bundled with your app such as images, icons and JSON.

2 NetworkImage

Loads images from a URL. Always think about loading and error states.

3 Icons

Material icons are quick for prototypes and clean interfaces.

4 pubspec

Indentation matters. YAML errors often come from spacing.

CODE

```
flutter:
  assets:
    - assets/images/logo.png

Image.asset('assets/images/logo.png')

Image.network(
  imageUrl,
  errorBuilder: (_, __, ___) => const Icon(Icons.error),
)
```

WHY THIS MATTERS

A missing image is usually a path or pubspec indentation issue.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Text, fonts and theme

Do not style every Text widget manually. Use ThemeData and textTheme so the app stays consistent.

1 TextStyle

Controls size, weight, color, height and letter spacing.

2 ThemeData

Central place for colors, typography and component styling.

3 textTheme

Predefined roles like titleLarge, bodyMedium and labelSmall.

4 Consistency

Use theme roles instead of random font sizes everywhere.

CODE

```
final textTheme = Theme.of(context).textTheme;

Text(
  'Dashboard',
  style: textTheme.headlineSmall?.copyWith(
    fontWeight: FontWeight.bold,
  ),
)
```

WHY THIS MATTERS

A good Flutter UI system starts with theme, spacing and reusable components.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Buttons, gestures and user actions

A button should clearly show what will happen. In Flutter, actions usually update state, navigate or call a service.

1 **ElevatedButton**
Good for primary important actions.

2 **TextButton**
Good for low-emphasis actions.

3 **IconButton**
Useful for compact actions like edit, delete or share.

4 **InkWell**
Adds tap behavior to custom UI cards.

CODE

```
ElevatedButton(
  onPressed: isLoading ? null : submitForm,
  child: isLoading
      ? const CircularProgressIndicator()
      : const Text('Continue'),
)
```

WHY THIS MATTERS

Disable buttons while a request is running to avoid duplicate submissions.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Forms and validation

Forms are everywhere: login, signup, search, checkout and profile editing. Use controllers carefully and dispose them when needed.

1 **TextEditingController**
Reads and changes text field values.

2 **Form**
Groups fields for validation and saving.

3 **validator**
Returns an error message or null when valid.

4 **dispose**
Clean controllers in StatefulWidget to avoid leaks.

CODE

```
final emailController = TextEditingController();

TextFormField(
  controller: emailController,
  validator: (value) {
    if (value == null || !value.contains('@')) {
      return 'Enter a valid email';
    }
    return null;
  },
)
```

WHY THIS MATTERS

Validation should guide the user. Avoid vague messages like invalid input.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

setState for simple local state

setState is perfect for small local state: counters, toggles, selected chips and form visibility.

Beginner mental model

1 Local state
State used only by one screen or widget.

2 setState block
Only change state inside the callback; do not run heavy async work inside it.

3 Rebuild scope
Only the current StatefulWidget subtree rebuilds.

4 Limit
When many screens need the same state, use a state management approach.

CODE

```
bool isFavorite = false;

IconButton(
  icon: Icon(
    isFavorite ? Icons.favorite : Icons.favorite_border,
  ),
  onPressed: () {
    setState(() => isFavorite = !isFavorite);
  },
)
```

WHY THIS MATTERS

Start with setState, but learn when state should move upward or into a controller.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Lift state up

When two widgets need the same value, keep state in their closest common parent and pass callbacks down.

1 **Parent owns state**
The parent stores the selected value or changing data.

2 **Child receives value**
Child widgets display what they are given.

3 **Child sends event**
Callbacks tell the parent what happened.

4 **Cleaner flow**
Data goes down, events go up.

CODE

```
class FilterChipItem extends StatelessWidget {
  final bool selected;
  final VoidCallback onTap;

  const FilterChipItem({
    required this.selected,
    required this.onTap,
  });
}
```

WHY THIS MATTERS

This pattern is the foundation of clean Flutter architecture.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Navigation basics

Navigation moves users between screens. Beginners should first learn named routes or MaterialPageRoute before complex routing packages.

1 push
Opens a new page on top of the current one.

2 pop
Returns to the previous page.

3 arguments
Pass simple data to the next screen.

4 routes
Centralize common navigation paths when the app grows.

CODE

```
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (_) => DetailsPage(product: product),
  ),
);

Navigator.pop(context);
```

WHY THIS MATTERS

Keep navigation calls close to user actions, but avoid passing huge objects everywhere.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Passing data between screens

A details screen usually needs an id or model from the previous screen. Prefer passing the minimum useful data.

Beginner mental model

1 Model object

Pass when the next screen needs already loaded data.

2 ID only

Pass an id when the next screen should fetch fresh data.

3 Result back

Use pop(result) when a child screen returns a selection.

4 Avoid globals

Global variables make navigation behavior hard to test.

CODE

```
final selected = await Navigator.push<Product>(
  context,
  MaterialPageRoute(
    builder: (_) => ProductPickerPage(),
  ),
);

if (selected != null) {
  setState(() => product = selected);
}
```

WHY THIS MATTERS

Navigation is easier when data flow is explicit.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

HTTP and JSON basics

Most apps talk to a backend. The beginner-friendly flow is: call API, decode JSON, convert to model, render state.

1 HTTP request
Use a package or service class to make network calls.

2 JSON decode
Convert response text into Map/List structures.

3 Model
A Dart class gives names and types to raw JSON fields.

4 UI state
Show loading, data, empty and error states clearly.

CODE

```
class Product {
  final int id;
  final String name;

  Product({required this.id, required this.name});

  factory Product.fromJson(Map<String, dynamic> json) {
    return Product(id: json['id'], name: json['name']);
  }
}
```

WHY THIS MATTERS

Do not render raw maps directly in UI. Convert API data into typed models.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Async, Future and loading state

API calls take time. Flutter must keep drawing frames while waiting, so async code should update UI states intentionally.

1 Future

Represents a value available later.

2 await

Pauses only the current async function, not the entire app.

3 loading

Show progress while the request is active.

4 finally

Reset loading state even when the request fails.

CODE

```
Future<void> loadProducts() async {
  setState(() => isLoading = true);
  try {
    products = await productService.fetchProducts();
  } catch (e) {
    error = e.toString();
  } finally {
    setState(() => isLoading = false);
  }
}
```

WHY THIS MATTERS

Many beginner bugs are not API bugs. They are missing loading, error or empty states.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Build an API list screen

A real beginner milestone is showing data from an API in a ListView with loading and error handling.

1 **Start loading**
Show a progress indicator before data arrives.

2 **Success**
Render the list from typed models.

3 **Empty**
Tell users when no records exist.

4 **Error**
Explain what failed and offer retry.

CODE

```
if (isLoading) return const Center(child: CircularProgressIndicator());
if (error != null) return ErrorView(onRetry: loadProducts);
if (products.isEmpty) return const EmptyView();

return ListView.builder(
  itemCount: products.length,
  itemBuilder: (_, i) => ProductTile(products[i]),
);
```

WHY THIS MATTERS

A professional app never leaves users staring at a blank screen.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Error, empty and success UI

Handling only success is a beginner mistake. A robust screen describes every important state.

Beginner mental model

1 Loading
Something is happening now.

2 Empty
The request worked, but there is no data.

3 Error
Something failed and the user needs next steps.

4 Success
Data is ready and interactive.

CODE

```
Widget buildBody() {
  switch (status) {
    case PageStatus.loading:
      return const LoadingView();
    case PageStatus.empty:
      return const EmptyView();
    case PageStatus.error:
      return ErrorView(onRetry: reload);
    case PageStatus.success:
      return ProductList(products);
  }
}
```

WHY THIS MATTERS

Even without advanced state management, you can model UI states cleanly.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

pubspec.yaml and packages

Packages extend Flutter. Use them carefully, read their maintenance status and avoid adding a package for every tiny task.

1 dependencies

Packages required by your app at runtime.

2 dev_dependencies

Tools needed only during development or testing.

3 assets

Local files included in the app bundle.

4 versions

Avoid random dependency upgrades right before release.

CODE

```
dependencies:
  flutter:
    sdk: flutter
  http: ^1.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  assets:
    - assets/images/
```

WHY THIS MATTERS

A clean pubspec is part of professional Flutter development.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Responsive design basics

Flutter runs on many screen sizes. Beginners should avoid fixed widths everywhere and learn MediaQuery, LayoutBuilder and flexible widgets.

1 MediaQuery
Reads screen size, padding and device information.

2 LayoutBuilder
Responds to the space a widget actually receives.

3 Wrap
Moves children to the next line when horizontal space is limited.

4 Flexible UI
Use Expanded, Flexible and constraints instead of hard-coded dimensions.

CODE

```
LayoutBuilder(
  builder: (context, constraints) {
    final isWide = constraints.maxWidth > 700;
    return isWide
        ? Row(children: panels)
        : Column(children: panels);
  },
)
```

WHY THIS MATTERS

Responsive design starts with avoiding assumptions about device width.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Debugging beginner Flutter issues

Debugging is a skill. Read the first meaningful error, not only the last line of a long stack trace.

1 Red screen

Runtime error. Read the message carefully; Flutter usually explains the problem.

2 Yellow stripes

Layout overflow. A child wants more room than available.

3 Print/log

Use logs to inspect state, but remove noisy prints later.

4 DevTools

Use Flutter DevTools for layout, performance and widget inspection.

CODE

```
debugPrint('products: ${products.length}');

// Useful during layout debugging
Container(
  color: Colors.red.withOpacity(.1),
  child: child,
)
```

WHY THIS MATTERS

The fastest developers are not error-free. They know how to read errors calmly.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Performance basics for beginners

Most beginner apps do not need advanced optimization, but they do need common sense: avoid expensive work in build and use lazy lists.

1 Avoid build work

Do not parse big JSON, call APIs or do heavy computation inside build.

2 const widgets

Use const when possible for stable static widgets.

3 Lazy lists

Use builder constructors for long lists.

4 Image size

Avoid loading huge images when thumbnails are enough.

CODE

```
const SizedBox(height: 16)
const Text('Welcome')

// Prefer this for long lists
ListView.builder(
  itemCount: items.length,
  itemBuilder: (_, i) => ItemTile(items[i]),
)
```

WHY THIS MATTERS

Performance improves when UI work stays predictable and data work stays outside build.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Clean folder structure

A beginner project can still be clean. Structure by feature when the app grows, and keep services separate from screens.

Beginner mental model

1 screens
Full pages the user navigates to.

2 widgets
Reusable UI pieces used inside screens.

3 models
Typed data classes used across the app.

4 services
API, storage or platform logic that should not live in widgets.

CODE

```
lib/
  main.dart
  features/
    products/
      product_screen.dart
      product_tile.dart
      product_model.dart
      product_service.dart
  shared/
    app_theme.dart
    app_button.dart
```

WHY THIS MATTERS

Organized folders reduce fear when your app grows from 3 files to 300 files.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Build a mini project: Product catalog

The best beginner project is small but complete: list products, open details, handle loading, validate a search field and show errors.

1 Screen 1

Product list with loading, empty, error and success states.

2 Screen 2

Product details with image, price and description.

3 Search

TextField filters the local list or sends query to API.

4 Polish

Use theme, reusable tiles and consistent spacing.

CODE

```
class ProductCatalogApp extends StatelessWidget {
  const ProductCatalogApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: buildAppTheme(),
      home: const ProductListScreen(),
    );
  }
}
```

WHY THIS MATTERS

A small finished app teaches more than ten unfinished tutorial screens.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Beginner mistakes to avoid

Most Flutter beginner mistakes are predictable. Avoiding them makes your code easier to grow and debug.

1 Huge main.dart

Split screens and reusable widgets early.

2 API in build

Call APIs from lifecycle methods, controllers or state management, not directly in build.

3 Hard-coded UI

Avoid fixed sizes where flexible layout is needed.

4 No states

Always handle loading, empty, error and success.

CODE

```
// Avoid
Widget build(context) {
  fetchProducts(); // bad place for API call
  return ProductList();
}

// Better
@override
void initState() {
  super.initState();
  fetchProducts();
}
```

WHY THIS MATTERS

Good habits early save months of refactoring later.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

30-day learning roadmap

Do not learn randomly. Move from widgets to layout, then state, navigation, APIs and finally app structure.

1 Week 1

Dart basics, widgets, MaterialApp, Scaffold, Text, Button and simple layout.

2 Week 2

Rows, columns, lists, forms, validation, assets and themes.

3 Week 3

Navigation, API calls, JSON models, loading/error states and local state.

4 Week 4

Mini project, cleanup, reusable widgets, debugging and deployment basics.

CODE

```
Day 1-5: widgets and layout
Day 6-10: forms, theme and assets
Day 11-17: state and navigation
Day 18-24: APIs and JSON
Day 25-30: mini project and polish
```

WHY THIS MATTERS

Consistency beats speed. Build one useful mini app before jumping to advanced packages.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

Quick reference: beginner Flutter essentials

Keep this page as a quick mental checklist when building a new screen.

1 Screen base

MaterialApp -> Scaffold -> body.

2 Layout

Use Padding, Column, Row, Expanded, ListView and Stack intentionally.

3 State

Start with setState for local state; lift state when siblings need it.

4 Data

API -> JSON -> model -> UI state -> widget tree.

CODE

```
// New screen checklist
1. What data does this screen need?
2. What states can it show?
3. Which widgets are reusable?
4. Does layout work on small screens?
5. What happens when API fails?
```

WHY THIS MATTERS

When you can answer these questions, you are no longer just copying Flutter code. You are designing screens.

BEGINNER LAB

Use this page in a real mini app

COMMON MISTAKE

Trying to solve everything inside one big widget. Split UI, state and service code step by step.

PROFESSIONAL HABIT

Name widgets by responsibility: ProductTile, LoginForm, EmptyState, ErrorView, not random UI names.

PRACTICE TASK

Create a tiny screen for this topic and add loading, empty and error behavior before adding decoration.

CHECK BEFORE MOVING ON

Can I explain this without looking at code? Can I build one example? Can I debug the most common error?

YOUR FIRST FLUTTER FOUNDATION.

**Learn the ideas that make every Flutter screen work:
widgets, constraints, state, routes, async data and clean project stru**

WHAT THIS GUIDE HELPS YOU DO

- Understand the Flutter widget tree instead of memorizing random widgets
- Build clean layouts with Row, Column, Stack, ListView and constraints
- Handle buttons, forms, navigation, loading states and API responses
- Avoid beginner mistakes that create messy rebuilds and broken UI
- Move from copy-paste tutorials to confident app structure

Continue learning with FlutterFever

flutterfever.com

Flutter tutorials • Developer guides • Tools