#18 | Flutter Fundamental - Bagian 3

| **Summary** | - Mahasiswa mampu menerapkan jenis-jenis gesture  - Mahasiswa mampu menerapkan input widget dan controllernya  - Mahasiswa mampu menerapkan custom input dan FormField widget |
| --- | --- |
| **URL** | 18-flutter-fundamental-bagian-3 |
| **Category** | Mobile Flutter |
| **Environment** | web |
| **Status** | Published |
| **Feedback Link** | https://github.com/jti-polinema/flutter-codelab/issues |
| **Author** | Tim Pengajar Mobile TSA 2022 |
| **Author LDAP** | hbb-polinema |
| **Analytics Account** | G-H7507PFEJB |

[Flutter Fundamental - Bagian 3](#_ok7k5uux6)

[Tujuan Praktikum](#_21yzqg98x7h6)

[Tujuan Visual](#_hs6hf8vnv2d6)

[Praktikum 1](#_xi2w31ib0f03)

[Praktikum 2](#_wpbsezku4ioh)

[Praktikum 3](#_ued7s4ffx9fq)

[Sumber Daya yang Dibutuhkan](#_dusjnkkj26ob)

[Daftar Perintah](#_6mgk7cf2soh8)

[Bantuan](#_f4c7kiz5py9x)

[Pengetahuan yang Anda harus Miliki](#_319h1e1sjzdz)

[Menangani User Gestures](#_t0tqoph2ohlh)

[Pointers](#_z4yzw1hgb1k2)

[Gestures](#_pt71gc6jnylb)

[Gestures di material widgets](#_xxl9wp76n58b)

[Memahami stateful widget life cycle](#_g7nutupwvu6v)

[Key life cycle states](#_89gt0c2q7m8e)

[Creation of the state](#_5w4vw82pp9b0)

[Initializing the state](#_oqkju35smcmf)

[Build](#_gkwhsy46pizn)

[Disposing of the state](#_hqa3lu1pn7e0)

[Mounted](#_7uyvqx1t47)

[Input widgets and forms](#_sl1nr883oe3v)

[Getting input through a controller](#_ndqn33mzkwzq)

[FormField and TextField](#_y4w796pa4qp3)

[Accessing the FormField widget's state](#_mpg6x3enher)

[Form](#_s4fuwn64dog5)

[Accessing the Form widget's state](#_gx5hx3cpzhfw)

[Using a key](#_auyyiycryknw)

[Using InheritedWidget](#_cxyxs6lfwvl6)

[Validating user input](#_qx6uhmge0oss)

[Custom input and FormField widgets](#_el89ede9berg)

[Creating custom inputs](#_lc4ug7jk1eko)

[Custom input widget example](#_od6s7wl3f9jr)

[Praktikum 1: Menerapkan Gesture Detector](#_vmpz7ap0erhd)

[Langkah 1: Buat Project Baru](#_spr93e7x3yo2)

[Langkah 2: Menambahkan GestureDetector](#_g8csgggwssw)

[Langkah 3: Buat Dokumentasi atau Laporan Praktikum](#_2q85gi9dhaiq)

[Langkah 4: Eksperimen Jenis Gesture Lain](#_p0t5r1yzb5qp)

[Praktikum 2: Menerapkan Input Widget dan Forms](#_86j27ymyw6sc)

[Langkah 1: Buat class FormContoh](#_z4xb79s85ro)

[Langkah 2: Buat class \_FormContohState](#_x89yqkiqmitd)

[Langkah 3: Tambah Widget FormContoh di main.dart](#_uvblkrqosg3y)

[Praktikum 3: Menerapkan Custom Input dan FormField Widget](#_gz7tbatl2uh8)

[Langkah 1: Buat class VerificationCodeInput](#_h9m2druzjfj8)

[Langkah 2: Buat class \_VerificationCodeInputState](#_2nzf86f8qvki)

[Langkah 3: Buat class VerificationCodeFormField](#_gxsz8i7mkdo3)

[Langkah 4: Buat class \_VerificationCodeFormFieldState](#_qp3pp1metiux)

[Langkah 5: Tambah variabel \_controller](#_ybah5qq959kt)

[Langkah 6: Tambah Widget VerificationCodeFormField](#_e4m3gfagvjh)

[Tugas Praktikum](#_dz35nqtk0wjb)

[Selamat!](#_s22qmcj4zu61)

[Apa selanjutnya?](#_9mqifffit2ew)

[Referensi](#_an95hoqxrlhg)

# Flutter Fundamental - Bagian 3



**Terakhir diperbarui:** 30 Agustus 2022

**Penulis:** Tim Pengajar Mobile Flutter

Pada codelab ini, Anda akan mempelajari konsep dan praktik untuk dasar-dasar framework Flutter dengan menerapkan jenis-jenis gesture, input widget dan controller-nya serta melakukan custom input dan FormField widget.

## Tujuan Praktikum

Setelah menyelesaikan codelab ini Anda akan mampu untuk:

* menerapkan jenis-jenis gesture
* menerapkan input widget dan controllernya
* menerapkan custom input dan FormField widget

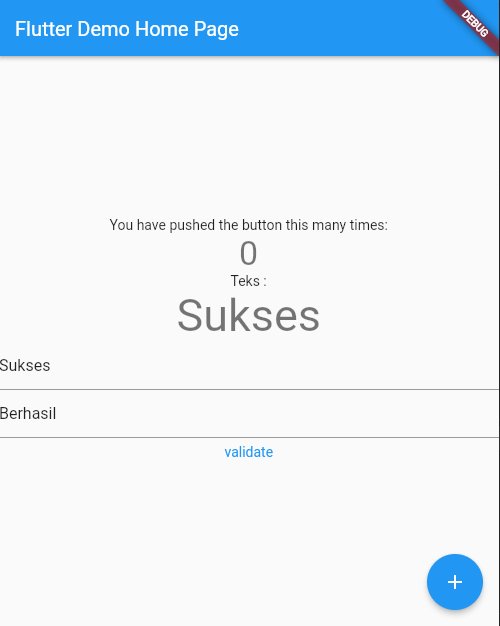
| **Catatan:** Materi ini diadaptasi dari Buku Flutter for Beginners - second edition dan Buku Ajar Pemrograman Mobile Flutter JTI Polinema serta sumber referensi lainnya. |
| --- |

## Tujuan Visual

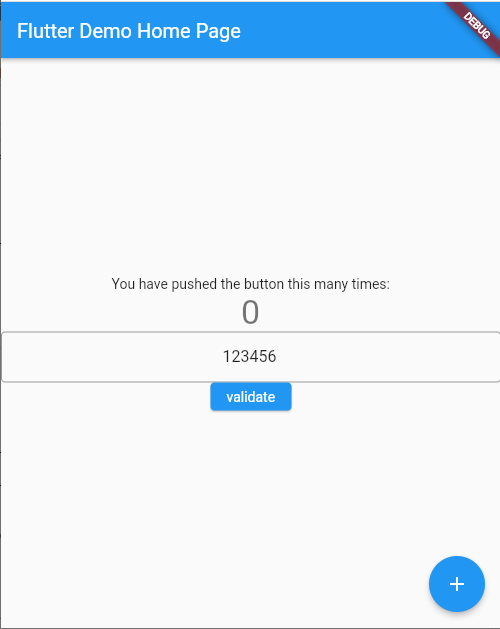
### Praktikum 1



### Praktikum 2



### Praktikum 3



## Sumber Daya yang Dibutuhkan

Berikut merupakan sumber daya yang diperlukan untuk menyelesaikan praktikum ini:

* PC atau Laptop dengan spesifikasi minimum RAM 4GB Processor Core i3
* Koneksi internet
* Chrome Browser
* Perangkat mobile (Android/iOS)

## Daftar Perintah

Tidak ada.

## Bantuan

Tidak ada.

## Pengetahuan yang Anda harus Miliki

* Algoritma dan Pemrograman Dasar
* Pemrograman Berorientasi Objek
* Bahasa Pemrograman Dart

# Menangani User Gestures

Duration: 10:00

Mobile Application akan sangat terbatas tanpa interaktivitas. Flutter framework memungkinkan untuk menangani user gestures dengan segala cara, dari simple taps hingga drag dan pan gestures. Event pada layar dalam sistem gesture Flutter dipisahkan menjadi dua layer, sebagai berikut:

**Pointers layer:** Layer ini menyimpan informasi mentah tentang bagaimana penunjuk (misalnya, sentuhan, mouse, atau stylus) berinteraksi dengan layar. Data mentah ini akan mencakup lokasi dan pergerakan pointer.

**Gestures layer:** Layer ini mengambil beberapa tindakan penunjuk dan mencoba memberi beberapa arti sebagai tindakan pengguna. Tindakan semantik ini (misalnya, tap, drag, atau scale) seringkali lebih berguna untuk aplikasi, dan ini adalah cara paling umum untuk menerapkan penanganan input pengguna.

## Pointers

Flutter memulai penanganan input layar di lapisan penunjuk level rendah. Secara umum, tidak perlu menggunakan event dari lapisan ini dalam aplikasi Anda, tetapi jika Anda perlu melakukan penanganan masukan yang dipesan lebih dahulu, maka Anda dapat menggunakan lapisan ini untuk menerima peristiwa pada setiap pembaruan penunjuk dan memutuskan bagaimana mengontrolnya. Misalnya, jika Anda membuat kode game, maka Anda mungkin memerlukan detail yang tepat pada setiap pembaruan penunjuk daripada mengandalkan peristiwa gerakan tingkat yang lebih tinggi. Kerangka kerja Flutter mengimplementasikan pengiriman peristiwa penunjuk pada pohon widget dengan mengikuti urutan peristiwa:

**PointerDownEvent** adalah tempat interaksi dimulai, dengan penunjuk bersentuhan dengan lokasi tertentu pada layar perangkat. Di sini, kerangka kerja mencari pohon widget untuk widget yang ada di lokasi penunjuk di layar. Tindakan ini disebut tes hit.

Setiap event berikut dikirim ke widget terdalam yang cocok dengan lokasi dan kemudian memunculkan pohon widget dari widget induk ke root. Penyebaran tindakan peristiwa ini tidak dapat diinterupsi. Peristiwa tersebut dapat berupa PointerMoveEvent, di mana lokasi penunjuk diubah, PointerUpEvent, yang menunjukkan bahwa penunjuk tidak lagi bersentuhan dengan layar, atau PointerCancelEvent, di mana penunjuk masih aktif di perangkat tetapi tidak lagi berinteraksi dengan aplikasi Anda.

Interaksi akan selesai dengan salah satu event PointerUpEvent atau PointerCancelEvent. Flutter menyediakan kelas Listener, yang dapat digunakan untuk mendeteksi peristiwa interaksi pointer yang terdaftar sebelumnya. Anda dapat membungkus pohon widget dengan widget ini untuk menangani peristiwa penunjuk pada subpohon widgetnya.

## Gestures

Meskipun memungkinkan, tidak selalu praktis untuk menangani sendiri event pointer menggunakan widget Pendengar. Sebaliknya, acara dapat ditangani pada lapisan kedua

sistem isyarat. Gerakan dikenali dari beberapa peristiwa penunjuk, dan bahkan beberapa pointer individu (*multitouch*). Ada beberapa jenis gerakan yang bisa ditangani:

**Tap**: Satu ketukan/sentuhan pada layar perangkat.

**Double-tap**: Ketuk dua kali dengan cepat pada lokasi yang sama di layar perangkat.

**Press and long-press**: Tekan pada layar perangkat, mirip dengan ketuk, tetapi memiliki kontak dengan layar untuk jangka waktu yang lebih lama sebelum dilepaskan.

**Drag**: Tekan yang dimulai dengan penunjuk yang bersentuhan dengan layar di beberapa lokasi, yang kemudian dipindahkan, dan berhenti berhubungan di lokasi lain di layar perangkat.

**Pan**: Mirip dengan peristiwa tarik. Di Flutter, arahnya berbeda; gerakan pan mencakup drag horizontal dan vertikal.

**Scale**: Dua penunjuk digunakan untuk gerakan seret untuk menggunakan isyarat skala. Ini juga mirip dengan gerakan zoom.

Seperti widget Listener untuk event pointer, Flutter menyediakan widget GestureDetector, yang berisi callback untuk semua event sebelumnya. Anda akan menggunakannya sesuai dengan interaksi yang ingin Anda izinkan. Mari kita lihat beberapa contoh GestureDetector.

## Gestures di material widgets

Meskipun GestureDetector adalah widget yang sangat berguna, sebagian besar waktu Anda tidak perlu menggunakannya karena widget bawaan sudah memiliki manajemen gerakan yang terpasang di dalamnya. Desain Material dan widget Cupertino iOS memiliki banyak gerakan yang diabstraksikan ke parameter konstruktor dengan menggunakan widget GestureDetector secara internal dalam kodenya. Misalnya, widget material seperti ElevatedButton menyematkan widget khusus bernama InkWell yang selain memberikan akses ke acara gerakan tap, juga akan membuat efek percikan pada widget target. Properti onPressed dari ElevatedButton memaparkan fungsionalitas tap yang dapat digunakan untuk mengimplementasikan aksi tombol.

# Memahami stateful widget life cycle

Duration: 15:00

Di pertemuan sebelumnya, kita melihat bagaimana widget stateful berbeda dari widget stateless dan bagaimana metode build() dapat dipanggil beberapa kali, dipicu oleh metode setState().

Namun, ada beberapa bagian tambahan dari siklus hidup widget stateful yang akan kita jelajahi saat ini karena hal itu penting untuk cara kita mengelola data input dan juga menjadi semakin penting di sepanjang sisa buku ini saat kita melihat widget yang lebih canggih interaksi.

## Key life cycle states

Ada beberapa status siklus hidup yang dapat dilalui oleh widget stateful. Di bagian ini, kita akan melihat status yang akan Anda perlukan di sebagian besar situasi. Nanti di buku ini, kami akan memperkenalkan status siklus hidup tambahan untuk skenario dan kasus sudut tertentu.

### Creation of the state

Pembuatan status terjadi di awal siklus hidup widget stateful, tepat setelah konstruktor dipanggil. Widget stateful membuat objek State pendamping untuk menahan state yang bisa berubah dengan memanggil metode createState() dan meneruskan sebuah instance dari objek State pendamping. Ini adalah langkah yang diperlukan dalam siklus hidup; jika tidak, widget stateful tidak akan memiliki status.

Kita melihat contoh metode createState() di pertemuan sebelumnya:

| class MyHomePage extends StatefulWidget {  MyHomePage({Key key, this.title}) : super(key: key);  final String title;    @override  \_MyHomePageState createState() => \_MyHomePageState();  } |
| --- |

### Initializing the state

Instance Status dapat menginisialisasi variabel statusnya atau persyaratan infrastruktur lainnya (seperti koneksi database) dalam metode initState(). Metode ini hanya dipanggil sekali saat widget ditambahkan ke pohon widget untuk pertama kalinya (yaitu, terlihat oleh pengguna) dan bersifat opsional.

Kita akan melihat beberapa contohnya nanti di bab ini, tetapi contoh dasar dari metode initState() terlihat sebagai berikut:

| @override  void initState() {  super.initState();  // Custom initialization logic here  } |
| --- |

Dalam contoh ini, kita melihat bahwa baris pertama metode harus berupa panggilan untuk menginisialisasi status kelas super. Ini diikuti oleh logika kustom apa pun yang diperlukan untuk menginisialisasi widget.

### Build

Seperti yang Anda lihat di bab sebelumnya, metode build dipanggil saat widget akan digambar ke layar. Ini dipanggil setelah initState() dan kemudian dipanggil setiap kali setState() dipicu.

### Disposing of the state

Saat widget dihapus dari pohon widget, metode buang() dipanggil. Pembersihan infrastruktur apa pun yang diperlukan, umumnya untuk aktivitas yang terjadi selama initState(), seperti menyiapkan pendengar basis data atau koneksi internet, akan dilakukan dalam metode buang().

Sekali lagi, kita akan melihat contohnya nanti di bagian ini, tetapi di sini adalah struktur kerangka dari metode ini:

| @override  void dispose() {  // Custom clean-up code here  super.dispose();  } |
| --- |

Dalam contoh ini, kita melihat bahwa baris terakhir dari metode harus berupa panggilan untuk membuang status kelas super. Namun, sebelum ini, logika khusus apa pun yang diperlukan untuk membuang widget dapat ditempatkan.

## Mounted

Selain status siklus hidup, ada bidang penting yang tersedia untuk Anda, dari kelas induk Stateful widget Anda, yang disebut mount. Ini akan memberi tahu Anda apakah widget masih terpasang ke pohon widget. Secara khusus, ketika initState dipanggil, maka mount ditandai sebagai benar, dan ketika buang dipanggil, mount ditandai sebagai palsu. Anda akan menggunakan ini untuk situasi seperti mendengarkan di database atau koneksi internet. Jika perubahan dalam database atau status koneksi internet dikodekan untuk memicu pembaruan widget (mungkin melalui setState()), maka sebaiknya tambahkan pemeriksaan terpasang sebelum memanggil setState() karena widget mungkin telah dihapus dari widget pohon antara waktu Anda menyiapkan pendengar dan waktu menerima pembaruan.

Mari kita lihat contoh sederhana:

| if (mounted) {  setState(() {  // Change state here  });  } |
| --- |

Dalam contoh ini, kami telah membungkus panggilan setState() dalam pemeriksaan terpasang untuk memastikan widget masih berada di pohon widget dan dapat digambar ulang.

Jadi, sekarang Anda tahu lebih banyak tentang siklus hidup widget stateful, dan memiliki pengetahuan tentang input pengguna melalui gerakan, mari kita lihat cara umum lainnya untuk menerima input pengguna, melalui widget dan formulir input.

# Input widgets and forms

Duration: 20:00

Kemampuan aplikasi Anda untuk mengelola gerakan adalah titik awal yang baik untuk interaksi dengan pengguna, tetapi untuk banyak aplikasi, Anda juga memerlukan cara untuk mendapatkan jenis masukan lain dari pengguna.

Mendapatkan data pengguna adalah hal yang menambahkan konten khusus dan penyesuaian ke banyak aplikasi.

Flutter menyediakan banyak widget data input untuk membantu developer mendapatkan berbagai jenis informasi dari pengguna. Kita telah melihat beberapa di antaranya, Widget – Membangun Tata Letak di Flutter, termasuk TextField, dan berbagai jenis widget Selector dan Picker.

Widget TextField memungkinkan pengguna memasukkan teks dengan keyboard. Widget TextField memperlihatkan metode onChanged, yang dapat digunakan untuk mendengarkan perubahan pada nilainya saat ini, seperti yang telah kita lihat sebelumnya dengan widget TextField. Namun, cara lain untuk mendengarkan perubahan adalah dengan menggunakan pengontrol.

## Getting input through a controller

Saat menggunakan widget TextField standar, kita perlu menggunakan properti pengontrolnya untuk mengakses nilainya. Ini dilakukan dengan kelas TextEditingController:

| final \_controller = TextEditingController.fromValue(  TextEditingValue(text: "Initial value"),  ); |
| --- |

Seperti yang Anda lihat, dengan menyetel properti teks dari pengontrol, kita dapat menentukan nilai awal dari widget yang dikontrolnya. Setelah membuat instance TextEditingController, kami mengatur properti controller dari widget TextField sehingga "mengontrol" widget:

| TextField(  controller: \_controller,  ); |
| --- |

TextEditingController diberi tahu setiap kali widget TextField memiliki nilai baru. Untuk mendengarkan perubahan, kita perlu menambahkan pendengar ke \_controller kita:

| \_controller.addListener(() {  this.setState(() {  \_textValue = \_controller.text;  });  }); |
| --- |

Kita harus menentukan fungsi panggilan balik yang akan dipanggil setiap kali widget TextField berubah. Dalam hal ini, kami telah membuat fungsi sebaris sederhana yang menetapkan variabel status \_textValue ke nilai teks di TextField seperti yang diambil melalui properti teks pada pengontrol.

Pendekatan serupa digunakan untuk widget input lainnya. Namun, seringkali, Anda ingin membuat formulir yang menampung sekelompok widget data input dan memiliki validasi serta umpan balik untuk pengguna yang bekerja di seluruh formulir.

## FormField and TextField

Flutter menyediakan dua widget untuk membantu mengatur penyimpanan data input, validasinya, dan memberikan umpan balik segera kepada pengguna. Ini adalah widget Form dan FormField.

Widget FormField berfungsi sebagai kelas dasar untuk membuat bidang input kita sendiri di dalam formulir.

Fungsinya adalah sebagai berikut:

• Untuk membantu proses pengaturan dan pengambilan nilai input saat ini

• Untuk memvalidasi nilai input saat ini

• Untuk memberikan validasi formulir umpan balik

Widget FormField sering kali memiliki widget Form sebagai ancestor, tetapi dalam beberapa kasus, ini tidak diperlukan. Misalnya, ketika kita memiliki satu FormField untuk menerima input, mungkin tidak perlu widget Formulir untuk mengelola pembaruan formulir.

Banyak widget input bawaan dari Flutter hadir dengan implementasi widget FormField yang sesuai. Salah satu contohnya adalah widget TextField, yang memiliki widget TextFormField khusus bentuk. Widget TextFormField membantu dengan akses ke nilai TextField dan juga menambahkan perilaku terkait formulir ke dalamnya, seperti validasi.

## Accessing the FormField widget's state

Jika kita menggunakan widget TextFormField, maka ada pendekatan alternatif untuk mengakses data input menggunakan status widget FormField:

final \_key = GlobalKey<FormFieldState<String>>();

...

TextFormField(

key: \_key,

);

Kita dapat menambahkan kunci ke TextFormField yang nantinya dapat digunakan untuk mengakses status widget saat ini melalui nilai key.currentState, yang akan berisi nilai bidang yang diperbarui.

Jenis kunci khusus mengacu pada jenis data yang digunakan bidang input. Pada contoh sebelumnya, ini adalah String, karena merupakan widget TextField, jadi jenis kuncinya tergantung pada widget tertentu yang digunakan.

Kelas FormFieldState<String> juga menyediakan metode dan properti lain yang berguna untuk menangani FormField:

• validate() akan memanggil callback validator widget, yang akan memeriksa nilainya saat ini dan mengembalikan pesan kesalahan, atau null jika valid.

• hasError dan errorText hasil dari validasi sebelumnya menggunakan fungsi sebelumnya. Dalam widget material, misalnya, ini menambahkan beberapa teks kecil di dekat bidang, memberikan umpan balik yang tepat kepada pengguna tentang kesalahan tersebut.

• save() akan memanggil callback onSaved widget.

• reset() akan menempatkan bidang dalam keadaan awalnya, dengan nilai awal (jika ada) dan kesalahan validasi yang jelas.

## Form

Memiliki FormFieldWidget membantu kami mengakses dan memvalidasi informasinya satu per satu. Tetapi ketika kita memiliki satu set widget input dalam struktur formulir, maka kita dapat menggunakan widget Formulir. Widget Formulir mengelompokkan instance FormFieldWidget secara logis, memungkinkan kita untuk melakukan operasi termasuk mengakses informasi bidang dan memvalidasi seluruh rangkaian bidang dengan cara yang lebih terstruktur.

Widget Form memungkinkan kita untuk menjalankan metode berikut di semua bidang turunan dengan mudah:

• save(): Ini akan memanggil metode simpan semua instance FormField, menyimpan semua data formulir di bidang sekaligus.

• validate(): Ini akan memanggil metode validasi semua instance FormField, menyebabkan semua kesalahan muncul sekaligus.

• reset(): Ini akan memanggil metode reset semua instance FormField, menyetel ulang seluruh formulir ke keadaan awalnya.

### Accessing the Form widget's state

Aplikasi Anda harus dapat mengakses status widget Formulir, sama seperti kami mengakses status widget FormField, sehingga Anda dapat menjalankan validasi, penyimpanan data, dan penyetelan ulang dari bagian lain antarmuka pengguna, tidak hanya di dalam pohon widget formulir . Sebagai contoh, Anda mungkin memiliki tombol tindakan mengambang yang memungkinkan Anda menyimpan formulir, atau tombol bilah aplikasi untuk mengatur ulang formulir.

Mari kita lihat dua cara untuk mengakses status formulir.

#### Using a key

Widget Formulir harus digunakan dengan kunci tipe FormState. FormState berisi pembantu untuk mengelola semua anak FormField:

final \_key = GlobalKey<FormFieldState<String>>();

...

Form(

key: \_key,

child: Column(

children: <Widget>[

TextFormField(),

TextFormField(),

],

),

);

Dalam contoh ini, kami memiliki Formulir dengan kunci global dan, secara tidak langsung, dua widget TextFormField sebagai turunan.

Kami kemudian dapat menggunakan kunci untuk mengambil status terkait widget Formulir dan memanggil validasinya dengan \_key.currentState.validate().

Sebagian besar waktu ini adalah cara terbaik untuk mengakses widget Formulir, tetapi jika Anda memiliki pohon widget yang kompleks, maka ada opsi lain. Mari kita lihat opsi alternatif ini.

#### Using InheritedWidget

Widget Formulir hadir dengan kelas yang berguna untuk menghilangkan kebutuhan untuk menambahkan kunci ke dalamnya dan tetap mendapatkan manfaatnya.

Setiap widget Formulir di pohon memiliki InheritedWidget terkait dengannya. Formulir dan banyak widget lain mengekspos ini dalam metode statis yang disebut of(), di mana kita melewati BuildContext, dan itu mencari pohon untuk menemukan status yang sesuai yang kita cari. Mengetahui hal ini, jika kita perlu mengakses widget Formulir di suatu tempat di bawahnya di pohon, kita bisa menggunakan Form.of(), dan kita mendapatkan akses ke fungsi yang sama seperti yang kita miliki jika kita menggunakan properti key:Setiap widget Form di pohon memiliki InheritedWidget yang terkait dengannya. Formulir dan banyak widget lain mengekspos ini dalam metode statis yang disebut of(), di mana kita melewati BuildContext, dan itu mencari pohon untuk menemukan status yang sesuai yang kita cari. Mengetahui hal ini, jika kita perlu mengakses widget Form di suatu tempat di bawahnya di pohon, kita dapat menggunakan Form.of(), dan kita mendapatkan akses ke fungsi yang sama seperti yang kita miliki jika kita menggunakan properti key:

| Widget build(BuildContext topContext) {  return Form(  child: Column(  mainAxisSize: MainAxisSize.min,  children: <Widget>[  TextFormField(  validator: (String value) {  return value.isEmpty ? "Not empty" : null;  },  ),  TextFormField(),  Builder(  builder: (BuildContext subContext) => TextButton(  onPressed: () {  final valid = Form.of(subContext).validate();  print("valid: $valid");  },  child: Text("validate"),  ),  )  ],  ),  );  } |
| --- |

Berikan perhatian khusus pada widget Builder yang digunakan untuk merender TextButton. Seperti yang telah kita lihat sebelumnya, widget yang diwarisi dapat digunakan untuk mencari pohon widget. Saat kita menggunakan Form.of(subContext), ia menggunakan BuildContext dari Builder, yang lebih rendah di bawah pohon widget daripada widget Form. Oleh karena itu, Form.of(subContext) akan mencari pohon widget dan menemukan Form.

Jika builder tidak ada dan kita menggunakan konteks dari metode build, maka Form.of(topContext) akan memulai pencarian pada pohon widget di atas widget Form dan tidak akan menemukan widget Form selama pencarian tersebut.

### Validating user input

Memvalidasi input pengguna adalah salah satu fungsi utama widget Formulir. Untuk memastikan data yang dimasukkan oleh pengguna valid, sangat penting untuk menjalankan pemeriksaan validasi karena pengguna mungkin tidak mengetahui semua nilai yang diizinkan atau mungkin telah melakukan kesalahan. Widget Formulir, dikombinasikan dengan instans FormField, membantu Anda menampilkan pesan kesalahan yang sesuai jika nilai input perlu diperbaiki, sebelum menyimpan data formulir melalui fungsi save()-nya.

Kita telah melihat, dalam contoh Form sebelumnya, bagaimana memvalidasi nilai bidang form. Mari kita lihat aliran sebenarnya:

1. Buat widget Form dengan FormField di atasnya.
2. Tentukan logika validasi pada setiap properti konstruktor validator FormField dengan melewatkan fungsi validasi sebagai argumen. Berikut adalah contoh fungsi inline:

| TextFormField(  validator: (String value) {  return value.isEmpty ? "Cannot be empty" : null;  },  ) |
| --- |

1. Saat pengguna memilih untuk mengirimkan formulir, panggil validasi() pada FormState dengan menggunakan kuncinya, atau metode Form.of yang telah dibahas sebelumnya.
2. Setiap FormField yang merupakan anak dari formulir akan memiliki validasi() metode yang disebut:
3. Jika validasi tidak berhasil, beberapa teks kesalahan dikembalikan sebagai string. Teks kesalahan ini kemudian ditampilkan di FormField kepada pengguna sehingga mereka dapat memperbaiki masalah dan mengirimkan formulir lagi.
4. Jika validasi berhasil, nilai null dikembalikan.
5. Jika validasi berhasil, metode save() dapat dipanggil pada FormState untuk menyimpan semua data dari kolom input.

Sekarang setelah Anda memahami formulir, mari kita lihat bagaimana kita bisa masuk lebih dalam tentang kustomisasi input formulir kita.

# Custom input and FormField widgets

Duration: 10:00

Kita telah melihat bagaimana widget Form dan FormField membantu manipulasi dan validasi input. Selain itu, kita tahu bahwa Flutter hadir dengan serangkaian widget input yang merupakan varian FormField yang berisi fungsi pembantu seperti simpan dan validasi.

Ekstensibilitas dan fleksibilitas Flutter ada di mana-mana dalam kerangka kerja, jadi membuat bidang masukan khusus Anda sendiri sangat mungkin.

## Creating custom inputs

Membuat input kustom di Flutter semudah membuat widget normal dan menyertakan metode yang dijelaskan sebelumnya. Kami biasanya melakukan ini dengan memperluas widget FormField<inputType>, di mana inputType adalah tipe nilai dari widget input.

Jadi, prosesnya adalah sebagai berikut:

1. Buat widget khusus yang extends dengan StatefulWidget (untuk melacak value) dan menerima input dari pengguna dengan mengenkapsulasi widget input lain, atau dengan menyesuaikan seluruh proses, seperti dengan menggunakan gesture.
2. Buat widget yang extends dengan FormField yang pada dasarnya menampilkan widget input yang dibuat pada langkah sebelumnya dan juga menampilkan bidangnya.

## Custom input widget example

Nanti, di pertemuan berikutnya, Plugin Pihak Ketiga, kita akan melihat cara menggunakan plugin untuk menambahkan autentikasi ke aplikasi kita. Untuk saat ini, kita akan membuat widget khusus yang akan mirip dengan yang digunakan pada langkah itu.

Dalam contoh ini, kami akan meminta nomor telepon pengguna dan kemudian berpura-pura bahwa mereka telah dikirimi kode verifikasi enam digit. Kami kemudian akan meminta mereka untuk memasukkan kode verifikasi, yang harus sesuai dengan nilai server agar berhasil masuk.

Untuk saat ini, itu saja informasi yang perlu kita ketahui untuk pembuatan widget input kustom. Anda akan melihat seperti berikut:



Widget akan memulai widget input enam digit sederhana, yang nantinya akan menjadi widget FormField dan menampilkan metode save(), reset(), dan validation().

# Praktikum 1: Menerapkan Gesture Detector

Duration: 10:00

Selesaikan langkah-langkah praktikum berikut ini menggunakan editor Visual Studio Code (VS Code) atau Android Studio atau code editor lain kesukaan Anda.

| **Perhatian:** Diasumsikan Anda telah berhasil melakukan setup environment Flutter SDK, VS Code, Flutter Plugin, dan Android SDK pada pertemuan pertama. |
| --- |

## Langkah 1: Buat Project Baru

Buatlah sebuah project flutter baru dengan nama **flutter\_fundamental\_3**. Lalu jadikan repository di GitHub Anda dengan nama **flutter-fundamental-part3.**

## Langkah 2: Menambahkan GestureDetector

Buka file main.dart lalu ganti bagian body dengan kode berikut. Untuk MyImageWidget() dapat Anda ganti dengan widget milik Anda sendiri.

| body: Center(  child: GestureDetector(  onTap: \_incrementCounter,  child: Column(  mainAxisAlignment: MainAxisAlignment.center,  children: <Widget>[  const MyImageWidget(),  Text(  '$\_counter',  style: Theme.of(context).textTheme.headline4,  ),  ],  )),  ), |
| --- |

## Langkah 3: Buat Dokumentasi atau Laporan Praktikum

Simpan lalu coba untuk **Run** project Anda. Untuk kode MyImageWidget() di sini menampilkan logo Polinema seperti gambar berikut. Jika Anda coba klik/tap pada gambar, maka angka di bawah akan terus bertambah. Mengapa demikian? Jelaskan dalam laporan README.md! Jangan lupa kode dan hasil tampilannya di screenshot.



## Langkah 4: Eksperimen Jenis Gesture Lain

Sekarang Anda ganti kode bagian onTap dari Langkah 2 dengan onDoubleTap dan onLongPress. Lalu screenshot masing-masing hasil tampilannya dan jelaskan fungsinya!

# Praktikum 2: Menerapkan Input Widget dan Forms

Duration: 30:00

Selesaikan langkah-langkah praktikum berikut ini dengan melanjutkan dari praktikum sebelumnya.

## Langkah 1: Buat class FormContoh

Buatlah file baru dengan nama forms.dart lalu buat class FormContoh seperti kode berikut

| class FormContoh extends StatefulWidget {  const FormContoh({Key? key}) : super(key: key);  @override  \_FormContohState createState() => \_FormContohState();  } |
| --- |

## Langkah 2: Buat class \_FormContohState

Kemudian buat class state-nya

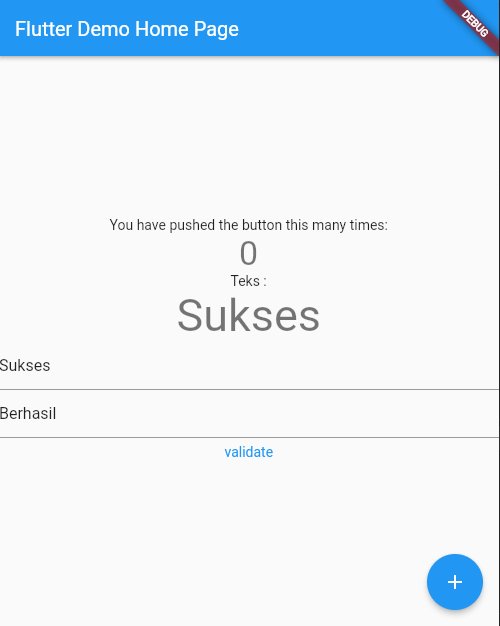
| class \_FormContohState extends State<FormContoh> {  final \_controller = TextEditingController.fromValue(  const TextEditingValue(text: "Initial value"),  );  final \_key = GlobalKey<FormFieldState<String>>();  String \_textValue = "";  @override  void initState() {  \_controller.addListener(() {  setState(() {  \_textValue = \_controller.text;  });  });  super.initState();  }  @override  Widget build(BuildContext topContext) {  return Form(  key: \_key,  child: Column(  mainAxisSize: MainAxisSize.min,  children: <Widget>[  const Text(  'Teks :',  ),  Text(  \_textValue,  style: Theme.of(context).textTheme.headline3,  ),  TextFormField(  controller: \_controller,  validator: (String? value) {  return value == null || value.isEmpty  ? "Tidak Boleh Kosong"  : null;  },  ),  TextFormField(),  Builder(  builder: (BuildContext subContext) => TextButton(  onPressed: () {  final valid = Form.of(subContext)!.validate();  if (kDebugMode) {  print("valid: $valid");  }  if (valid) {  setState(() {  \_textValue = \_controller.text;  });  }  },  child: const Text("validate"),  ),  )  ],  ),  );  }  } |
| --- |

## Langkah 3: Tambah Widget FormContoh di main.dart

Lalu pindah ke file main.dart dan panggil widget FormContoh tersebut yang telah dibuat.

| children: <Widget>[  const Text(  'You have pushed the button this many times:',  ),  Text(  '$\_counter',  style: Theme.of(context).textTheme.headline4,  ),  const FormContoh(),  . . . |
| --- |

Jangan lupa sesuaikan kode dan import di file main.dart kemudian akan tampil gambar seperti berikut. Jika terdapat error atau warning, silakan diperbaiki.



# Praktikum 3: Menerapkan Custom Input dan FormField Widget

Duration: 30:00

Selesaikan langkah-langkah praktikum berikut ini dengan melanjutkan dari praktikum sebelumnya. Anda dapat melakukan *comment* terhadap widget dari praktikum 2.

## Langkah 1: Buat class VerificationCodeInput

Buat file input\_fields.dart lalu isi kode seperti berikut.

| class VerificationCodeInput extends StatefulWidget {  final BorderSide borderSide;  final void Function(String)? onChanged;  final TextEditingController? controller;  const VerificationCodeInput({  Key? key,  this.controller,  this.borderSide = const BorderSide(),  this.onChanged,  }) : super(key: key);  @override  \_VerificationCodeInputState createState() => \_VerificationCodeInputState();  } |
| --- |

## Langkah 2: Buat class \_VerificationCodeInputState

Kemudian dibawahnya buat class \_VerificationCodeInputState yang di-extends dengan State.

| class \_VerificationCodeInputState extends State<VerificationCodeInput> {  @override  Widget build(BuildContext context) {  return TextField(  controller: widget.controller,  inputFormatters: [  FilteringTextInputFormatter.allow(RegExp("[0-9]")),  LengthLimitingTextInputFormatter(6),  ],  textAlign: TextAlign.center,  decoration: InputDecoration(  border: OutlineInputBorder(  borderSide: widget.borderSide,  ),  ),  keyboardType: TextInputType.number,  onChanged: widget.onChanged,  );  }  } |
| --- |

## Langkah 3: Buat class VerificationCodeFormField

Masih di file input\_fields.dart, buatlah class VerificationCodeFormField seperti berikut yang diletakkan dibawahnya.

| /// FormField version of the VerificationCodeInput widget  class VerificationCodeFormField extends FormField<String> {  final TextEditingController controller;  VerificationCodeFormField({  Key? key,  FormFieldSetter<String>? onSaved,  required this.controller,  FormFieldValidator<String>? validator,  }) : super(  key: key,  validator: validator,  builder: (FormFieldState<String> field) {  \_VerificationCodeFormFieldState state = field as \_VerificationCodeFormFieldState;  return VerificationCodeInput(  controller: state.\_controller,  );  },  );  @override  FormFieldState<String> createState() => \_VerificationCodeFormFieldState();  } |
| --- |

## Langkah 4: Buat class \_VerificationCodeFormFieldState

Selanjutnya buat class state-nya seperti berikut.

| class \_VerificationCodeFormFieldState extends FormFieldState<String> {  final TextEditingController \_controller = TextEditingController(text: "");  @override  void initState() {  super.initState();  \_controller.addListener(\_controllerChanged);  }  @override  void reset() {  super.reset();  \_controller.text = "";  }  void \_controllerChanged() {  didChange(\_controller.text);  }  @override  void dispose() {  \_controller.removeListener(\_controllerChanged);  super.dispose();  }  } |
| --- |

## Langkah 5: Tambah variabel \_controller

Buka file main.dart lalu tambahkan variabel \_controller di dalam class extends State

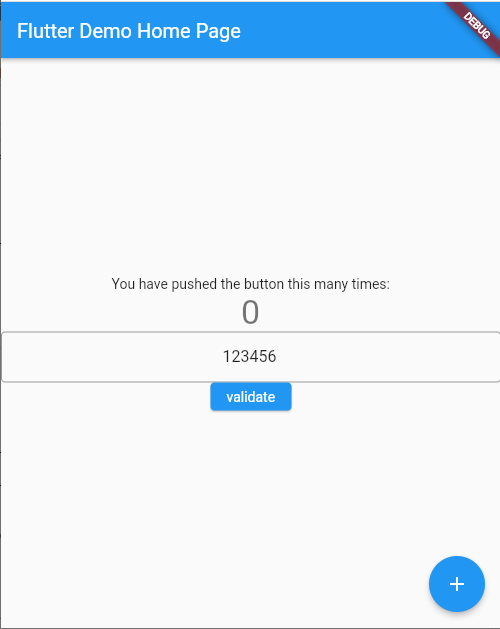
| final TextEditingController \_controller = TextEditingController.fromValue(const TextEditingValue(text: "isi angka saja")); |
| --- |

## Langkah 6: Tambah Widget VerificationCodeFormField

Lalu masuk ke method Widget build pada bagian children: <Widget> tambahkan Form seperti kode berikut. Lakukan import dari file input\_fields.dart untuk VerificationCodeFormField.

| Form(  child: Column(  mainAxisSize: MainAxisSize.min,  children: <Widget>[  VerificationCodeFormField(controller: \_controller),  Builder(  builder: (BuildContext subContext) => ElevatedButton(  onPressed: () {  final valid = Form.of(subContext)?.validate();  if (kDebugMode) {  print("valid: $valid");  }  },  child: const Text("validate"),  ),  )  ],  ),  ), |
| --- |

Maka hasilnya akan seperti berikut. Silakan coba isi form field dengan mengetikkan huruf dan angka. Apa yang terjadi ? Jelaskan dalam laporan praktikum Anda!



# Tugas Praktikum

Duration: 90:00

1. Selesaikan Praktikum 1 sampai 3, lalu dokumentasikan dan push ke repository Anda berupa screenshot setiap hasil pekerjaan beserta penjelasannya di file README.md!
2. Kumpulkan berupa link repository GitHub Anda ke LMS!

# Selamat!

Duration: 0:00

Selamat Anda telah menyelesaikan Codelab ini. Anda telah mempelajari terkait Flutter Fundamental untuk penanganan User Input, Gesture, dan FormField.

Pada codelab berikutnya, Anda akan mempelajari tentang Flutter Fundamental Bagian 4 terkait ListMap dan aplikasi konversi suhu dengan stateful widget.

## Apa selanjutnya?

Silakan cek beberapa sumber belajar lainnya…

* [FlutLab is a modern Flutter online IDE](https://flutlab.io/)
* [Flutter System Architecture](https://docs.google.com/presentation/d/1cw7A4HbvM_Abv320rVgPVGiUP2msVs7tfGbkgdrTy0I/edit#slide=id.p)

## Referensi

* [Flutter for Beginners Second Edition](https://drive.google.com/file/d/1hJbctx-_O3B6kUeG70bJD5xfbnsWwBXY/view?usp=sharing)
* <https://www.youtube.com/flutterdev>
* <https://github.com/PacktPublishing/Flutter-for-Beginners-Second-Edition>
* Buku Ajar Pemrograman Mobile Dengan Flutter, Jurusan Teknologi Informasi, Politeknik Negeri Malang. 2021.