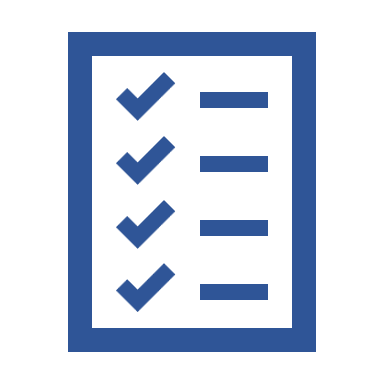
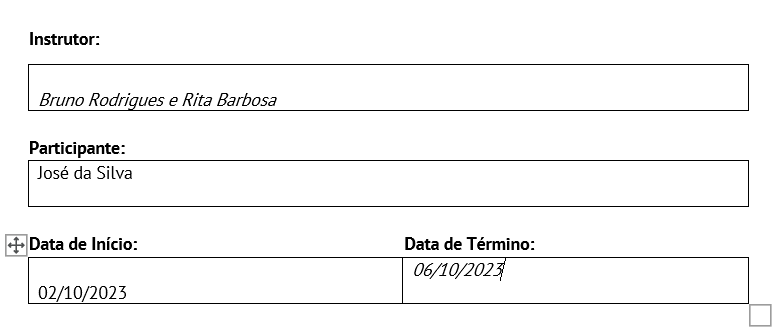
|  |
| --- |
| **Relatório do Miniprojeto do Curso: Explorando a Integração de IoT com *Flutter* para o *Smart Grid*** |

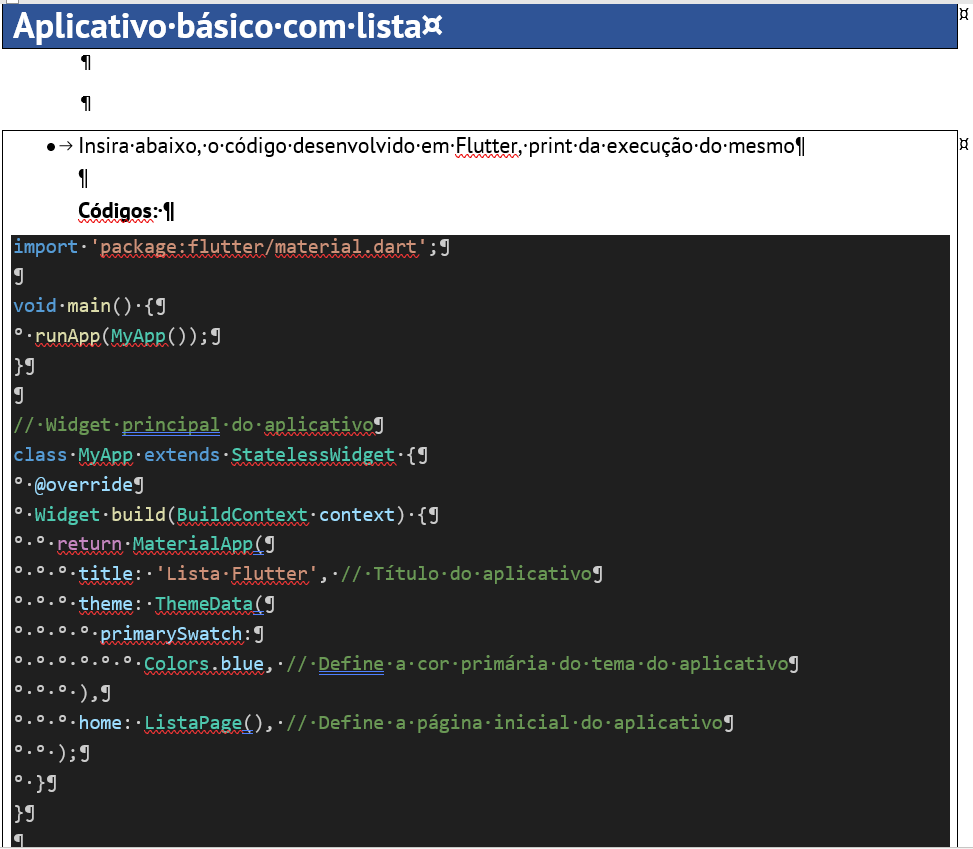
Esta é a estrutura do modelo do relatório, do qual deve ser preenchido e entregue via Plataforma de Educação do Fit como comprovação de desempenho no curso.

**Instruções** 

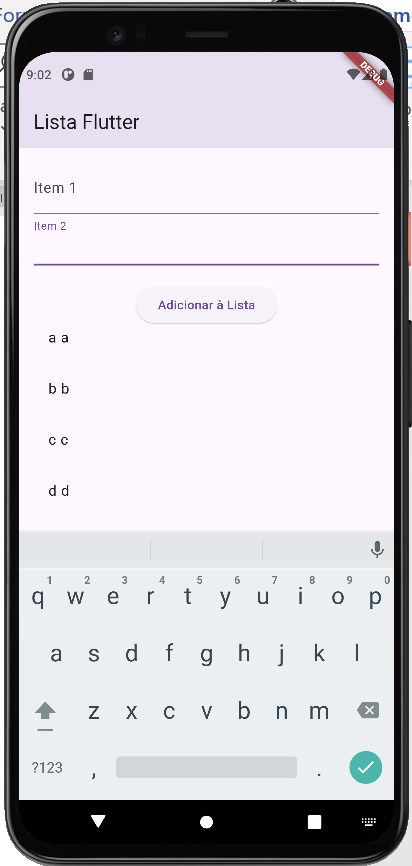
1. Preencha as informações nos campos, conforme exemplo abaixo:



1. Nesta questão, o aluno deverá colocar as evidências dos 03 exercícios realizados em aula, sendo:
   1. Aplicativo básico com lista (Inserir o código e print da execução do aplicativo em *Flutter*).
   2. Aplicativo MQTT básico (Inserir o código e print da execução do aplicativo em *Flutter*).
   3. Aplicativo para criar um gráfico de linha (Inserir o código e print da execução do aplicativo em *Flutter*).
   4. *Flutter* Gráfico e MQTT: Visualização Dinâmica de Dados em Ação (Inserir o código e print da execução do aplicativo em *Flutter*).
2. Você deve inserir o código e print da execução do aplicativo em *Flutter.* Segue abaixo exemplos de um dos 04 exercícios sobre Aplicativo básico com lista.

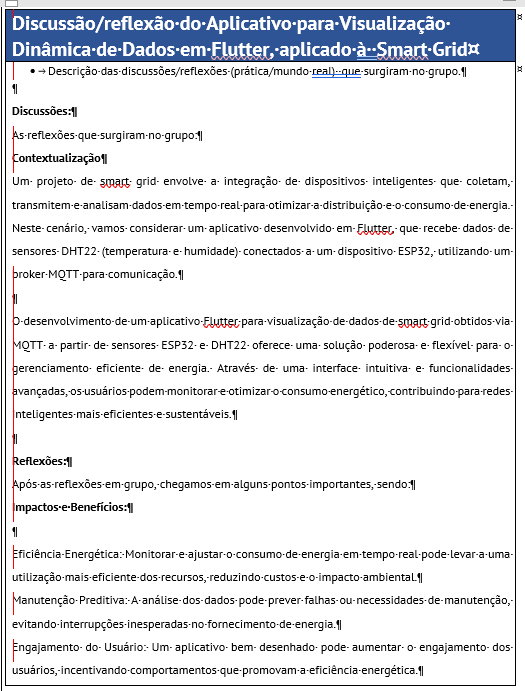


**Print Execução**



1. Discussão/reflexão do Aplicativo para Visualização Dinâmica de Dados em *Flutter:* aplicado à Smart Grid.

* No campo Discussão: descreva os tópicos que surgiram na discussão em grupo;
* No campo Reflexão: descreva os passos definidos em grupo para a solução, conforme exemplo abaixo:



* Salve o arquivo em formato .PDF e faça a entrega deste arquivo na Plataforma de Educação do Fit. Os instrutores lhe auxiliaram para a entrega do arquivo.

**Instrutor(s):**

|  |
| --- |
| *Bruno Rodrigues*  *Rita Barbosa* |

**Participante:**

|  |
| --- |
| Thayná Almeida |

**Data de Início:** **Data de Término:**

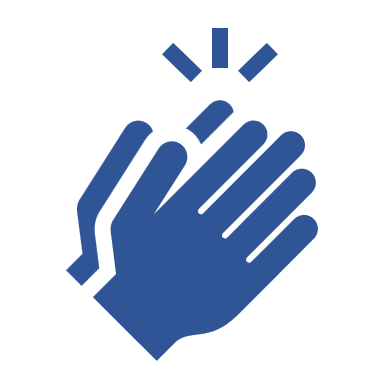
|  |  |
| --- | --- |
| 29/07/2024 | *05/08/2024* |

|  |
| --- |
| **Aplicativo básico com lista** |
| * Insira abaixo o código desenvolvido em *Flutter*, com o *print* da execução do mesmo:   **Códigos:**  // PROJETO 4 - TRABALHANDO COM LISTAS (ADICIONAR E REINICIAR LISTA DE ITENS)  import 'package:flutter/material.dart';  void main() {    runApp(      listApp(),    );  }  class listApp extends StatelessWidget {    const listApp({super.key});    @override    Widget build(BuildContext context) {      return MaterialApp(        home: Scaffold(          appBar: AppBar(            backgroundColor: Colors.lightBlueAccent,            title: Text(              "INCREMENTAL LIST",              style: TextStyle(color: Colors.white, fontWeight: FontWeight.bold),            ),          ),          body: content(),        ),      );    }  }  int quantidade = 0;  class content extends StatefulWidget {    const content({super.key});    @override    State<content> createState() => \_contentState();  }  class \_contentState extends State<content> {    @override    Widget build(BuildContext context) {      return Center(        child: Column(          children: [            ElevatedButton(              style: ButtonStyle(                  backgroundColor: WidgetStateProperty.resolveWith((states) {                if (states.contains(WidgetState.pressed)) {                  return Colors.green;                }                return Colors.greenAccent;              })),              onPressed: () {                //atualiza a tela com a nova quantidade                quantidade = quantidade + 1;                setState(() {});              },              child: Text(                "ADD",              ),            ),            ElevatedButton(              style: ButtonStyle(                  backgroundColor: WidgetStateProperty.resolveWith((states) {                if (states.contains(WidgetState.pressed)) {                  return Colors.amber;                }                return Colors.amberAccent;              })),              onPressed: () {                quantidade = 0;                setState(() {});              },              child: Text("RESTART"),            ),            Expanded(              child: ListView.builder(                  itemCount: quantidade,                  itemBuilder: (context, index) {                    return ListTile(                      title: Text("Thayná: $index"),                    );                  }),            ),          ],        ),      );    }  }  **Print Execução** |
|  |
| **Aplicativo MQTT básico** |
| * Insira abaixo o código desenvolvido em *Flutter,* com o *print* da execução do mesmo:   **Códigos:**  import 'package:flutter/material.dart';  import 'package:mqtt\_client/mqtt\_client.dart';  import 'package:mqtt\_client/mqtt\_server\_client.dart';  void main() {    runApp(      mqtt(),    );  }  class mqtt extends StatelessWidget {    const mqtt({super.key});    @override    Widget build(BuildContext context) {      return MaterialApp(        home: Scaffold(          appBar: AppBar(            backgroundColor: Colors.amberAccent,            title: Text("MQTT Client"),          ),          body: conteudo(),        ),      );    }  }  class conteudo extends StatefulWidget {    const conteudo({super.key});    @override    State<conteudo> createState() => \_conteudoState();  }  class \_conteudoState extends State<conteudo> {    // Controladores de texto para capturar a entrada do usuário.    final TextEditingController \_topicController = TextEditingController();    final TextEditingController \_brokerController = TextEditingController();    // Variáveis para armazenar o status da conexão e a mensagem recebida.    String \_status = "Disconnected";    String \_receivedMessage = "";    // Cliente MQTT.    late MqttServerClient \_client;    // Função que realiza a conexão ao broker MQTT.    void \_connect() async {      // Inicializa o cliente MQTT com o endereço do broker.      \_client = MqttServerClient(\_brokerController.text, '');      // Habilita o log para depuração      \_client.logging(on: true);      // Define os callbacks para eventos de desconexão, conexão e inscrição.      \_client.onDisconnected = \_onDisconnected;      \_client.onConnected = \_onConnected;      \_client.onSubscribed = \_onSubscribed;      // Mensagem de conexão com configurações adicionais.      final connMessage = MqttConnectMessage()          // Identificador do cliente.          .withClientIdentifier('flutter\_client')          .startClean()          .withWillQos(              // Define a qualidade de serviço (QoS).              MqttQos.atLeastOnce);      \_client.connectionMessage = connMessage;      try {        // Tenta conectar ao broker.        await \_client.connect();      } catch (e) {        // Em caso de falha na conexão, atualiza o status e desconecta.        setState(() {          \_status = 'Connection Failed: $e';          \_client.disconnect();        });        return;      }      // Listener para mensagens recebidas.      \_client.updates!.listen((List<MqttReceivedMessage<MqttMessage>> c) {        final MqttPublishMessage recMess = c[0].payload as MqttPublishMessage;        final String pt =            MqttPublishPayload.bytesToStringAsString(recMess.payload.message);        // Atualiza a mensagem recebida no estado do widget.        setState(() {          \_receivedMessage = pt;        });      });      // Inscreve-se no tópico fornecido pelo usuário.      \_client.subscribe(\_topicController.text, MqttQos.atLeastOnce);    }    // Callback para quando o cliente é desconectado.    void \_onDisconnected() {      setState(() {        \_status = 'Disconnected';      });    }    // Callback para quando o cliente é conectado.    void \_onConnected() {      setState(() {        \_status = 'Connected';      });    }    // Callback para quando o cliente se inscreve em um tópico.    void \_onSubscribed(String topic) {      setState(() {        \_status = 'Subscribed at Topic: $topic';      });    }    @override    Widget build(BuildContext context) {      return Padding(        padding: EdgeInsets.all(16),        child: Column(          children: [            TextField(              controller: \_brokerController,              decoration: InputDecoration(                  labelText: "MQTT Broker",                  labelStyle: TextStyle(color: Colors.blueGrey)),            ),            TextField(              controller: \_topicController,              decoration: InputDecoration(                  labelText: "Topic",                  labelStyle: TextStyle(color: Colors.blueGrey)),            ),            SizedBox(              height: 16,            ),            ElevatedButton(              style: ButtonStyle(                  backgroundColor: WidgetStateProperty.resolveWith((states) {                if (states.contains(WidgetState.hovered)) {                  return Colors.green;                }                return Colors.greenAccent;              })),              onPressed: () {                \_connect();              },              child: Text("CONNECT"),            ),            SizedBox(              height: 16,            ),            Text(              "[Status] $\_status",              style: TextStyle(backgroundColor: Colors.amberAccent),            ),            Text("Received Message: $\_receivedMessage"),          ],        ),      );    }  }  **Print Execução :** |
|  |

|  |
| --- |
| **Aplicativo para criar um gráfico de linha** |

|  |
| --- |
| * Insira abaixo o código desenvolvido em *Flutter,* com o print da execução do mesmo:   **Códigos:**  // PROJETO 7 - GRAFICO DE LINHAS COM INSERÇÃO MANUAL DE VALORES  import 'dart:convert';  import 'dart:math';  import 'package:flutter/material.dart';  import 'package:syncfusion\_flutter\_charts/charts.dart';  void main() {    runApp(      chart\_app(),    );  }  class chart\_app extends StatelessWidget {    const chart\_app({super.key});    @override    Widget build(BuildContext context) {      return MaterialApp(        home: Scaffold(          appBar: AppBar(            backgroundColor: Colors.amberAccent,            title: Text(              "LINE CHART",              style: TextStyle(fontWeight: FontWeight.bold),            ),          ),          body: contudo(),        ),      );    }  }  class contudo extends StatefulWidget {    const contudo({super.key});    @override    State<contudo> createState() => \_contudoState();  }  class \_contudoState extends State<contudo> {    final TextEditingController \_xController =        // Controlador para o campo de texto do eixo X.        TextEditingController();    final TextEditingController \_yController =        // Controlador para o campo de texto do eixo Y.        TextEditingController();    // Lista para armazenar os pontos de dados inseridos.    List<\_DataPoint> \_dataPoints = [];    @override    Widget build(BuildContext context) {      return Padding(        padding: EdgeInsets.all(16),        child: Column(          children: [            Row(              children: [                Expanded(                  child: TextField(                    controller: \_xController,                    decoration: InputDecoration(                      labelText: "(X) VALUE",                      labelStyle: TextStyle(color: Colors.blueGrey),                    ),                  ),                ),                SizedBox(                  width: 16,                ),                Expanded(                  child: TextField(                    controller: \_yController,                    decoration: InputDecoration(                      labelText: "(Y) VALUE",                      labelStyle: TextStyle(color: Colors.blueGrey),                    ),                  ),                ),              ],            ),            SizedBox(              height: 16,            ),            ElevatedButton(              style: ButtonStyle(                  backgroundColor: WidgetStateColor.resolveWith((states) {                if (states.contains(WidgetState.pressed)) {                  return Colors.green;                }                return Colors.greenAccent;              })),              onPressed: () {                setState(() {                  // Tenta converter o texto do campo X para um número.                  double? x = double.tryParse(\_xController.text);                  // Tenta converter o texto do campo Y para um número.                  double? y = double.tryParse(\_yController.text);                  if (x != null && y != null) {                    // Atualiza o estado do widget para adicionar o novo ponto de dados.                    \_dataPoints.add(\_DataPoint(x, y));                  }                  \_xController.clear(); // Limpa o campo de texto do eixo X.                  \_yController.clear(); // Limpa o campo de texto do eixo Y.                });              },              child: Text("ADD"),            ),            ElevatedButton(              style: ButtonStyle(                  backgroundColor: WidgetStateColor.resolveWith((states) {                if (states.contains(WidgetState.pressed)) {                  return Colors.red;                }                return Colors.redAccent;              })),              onPressed: () {                setState(() {                  // Limpa a lista de pontos de dados.                  \_dataPoints.clear();                });              },              child: Text(                "DELETE CHART",                style: TextStyle(color: Colors.white),              ),            ),            SizedBox(              height: 16,            ),            Expanded(              child: SfCartesianChart(                primaryXAxis: NumericAxis(), // Define o eixo X como numérico.                primaryYAxis: NumericAxis(), // Define o eixo Y como numérico.                series: <CartesianSeries<dynamic, dynamic>>[                  LineSeries<\_DataPoint, double>(                    dataSource:                        \_dataPoints, // Fonte de dados para a série de linha.                    xValueMapper: (\_DataPoint point, \_) =>                        point.x, // Mapeia o valor X de cada ponto de dados.                    yValueMapper: (\_DataPoint point, \_) =>                        point.y, // Mapeia o valor Y de cada ponto de dados.                  ),                ],              ),            ),          ],        ),      );    }  }  class \_DataPoint {    // Classe para representar um ponto de dados.    \_DataPoint(this.x, this.y); // Construtor que inicializa os valores X e Y.    final double x; // Valor X do ponto de dados.    final double y; // Valor Y do ponto de dados.  }  **Print Execução:** |
|  |
| ***Flutter* Gráfico e MQTT: Visualização Dinâmica de Dados em Ação** |
| * Insira abaixo o código desenvolvido em *Flutter*, com o print da execução do mesmo:   **Códigos:**  import 'package:flutter/material.dart';  import 'package:syncfusion\_flutter\_charts/charts.dart';  import 'package:mqtt\_client/mqtt\_client.dart';  import 'package:mqtt\_client/mqtt\_server\_client.dart';  import 'package:intl/intl.dart'; // Importação do pacote intl  void main() {    runApp(      mqtt\_grafico(),    );  }  class mqtt\_grafico extends StatelessWidget {    const mqtt\_grafico({super.key});    @override    Widget build(BuildContext context) {      return MaterialApp(        home: Scaffold(          appBar: AppBar(            title: Text(              "MQTT Chart",              style: TextStyle(color: Colors.white),            ),            backgroundColor: Colors.orangeAccent,          ),          body: conteudo(),        ),      );    }  }  class conteudo extends StatefulWidget {    const conteudo({super.key});    @override    State<conteudo> createState() => \_conteudoState();  }  class \_conteudoState extends State<conteudo> {    final TextEditingController \_brokerController =        TextEditingController(); // Controlador para o TextField do broker    final TextEditingController \_topicController =        TextEditingController(); // Controlador para o TextField do tópico    List<\_ChartData> \_chartData = []; // Lista para armazenar os dados do gráfico    MqttServerClient? \_client; // Cliente MQTT    late ChartSeriesController        \_chartSeriesController; // Controlador para a série do gráfico  // Função para conectar ao broker MQTT    void \_connect() async {      final String broker = \_brokerController.text; // Obtém o broker do TextField      final String topic = \_topicController.text; // Obtém o tópico do TextField      // Inicializa o cliente MQTT      \_client = MqttServerClient(broker, '');      \_client!.logging(on: true);      \_client!.onConnected = \_onConnected;      \_client!.onDisconnected = \_onDisconnected;      \_client!.onSubscribed = \_onSubscribed;      // Configuração da mensagem de conexão      final connMessage = MqttConnectMessage()          .withClientIdentifier('Mqtt\_Identifier')          .keepAliveFor(60)          .withWillTopic('willtopic')          .withWillMessage('My Will message')          .startClean()          .withWillQos(MqttQos.atLeastOnce);      \_client!.connectionMessage = connMessage;      try {        // Tenta conectar ao broker MQTT        await \_client!.connect();      } catch (e) {        print('Exception: $e');        \_client!.disconnect();      }      // Verifica se a conexão foi bem-sucedida      if (\_client!.connectionStatus!.state == MqttConnectionState.connected) {        print('MQTT client connected');        \_client!.subscribe(topic, MqttQos.atLeastOnce); // Inscreve no tópico      } else {        print(            'ERROR: MQTT client connection failed - disconnecting, state is ${\_client!.connectionStatus!.state}');        \_client!.disconnect();      }    }    // Função chamada quando a conexão é estabelecida    void \_onConnected() {      print('Connected');    }    // Função chamada quando a conexão é perdida    void \_onDisconnected() {      print('Disconnected');    }  // Função chamada quando a inscrição no tópico é bem-sucedida    void \_onSubscribed(String topic) {      print('Subscribed to $topic');      \_client!.updates!.listen((List<MqttReceivedMessage<MqttMessage>> c) {        final MqttPublishMessage recMess = c[0].payload as MqttPublishMessage;        final pt =            MqttPublishPayload.bytesToStringAsString(recMess.payload.message);        print('Received message: $pt from topic: ${c[0].topic}>');        final now = DateTime.now(); // Captura a data e hora atuais        final formattedTime =            DateFormat('HH:mm').format(now); // Formata a hora como HH:mm        setState(() {          \_chartData.add(\_ChartData(              now, double.tryParse(pt) ?? 0)); // Adiciona os dados ao gráfico          \_chartSeriesController.updateDataSource(            addedDataIndexes: [              \_chartData.length - 1            ], // Atualiza o gráfico com os novos dados          );        });      });    }    @override    Widget build(BuildContext context) {      return Padding(        padding: EdgeInsets.all(16),        child: Column(          children: [            TextField(              controller: \_brokerController,              decoration: InputDecoration(                border: OutlineInputBorder(),                labelText: "MQTT Broker",              ),            ),            SizedBox(              height: 10,            ),            TextField(              controller: \_topicController,              decoration: InputDecoration(                border: OutlineInputBorder(),                labelText: "Topic",              ),            ),            SizedBox(              height: 16,            ),            ElevatedButton(              onPressed: () {                \_connect(); // Chama a função de conexão ao broker MQTT              },              child: Text("CONNECT"),            ),            SizedBox(              height: 16,            ),            Expanded(              child: SfCartesianChart(                primaryXAxis: DateTimeAxis(                  dateFormat:                      DateFormat.Hm(), // Formatação para exibir hora e minutos                  intervalType: DateTimeIntervalType                      .minutes, // Tipo de intervalo como minutos                  title: AxisTitle(text: 'HOUR'), // Título do eixo X                ),                primaryYAxis: NumericAxis(                  title: AxisTitle(text: 'VALUE'), // Título do eixo Y                ),                series: <LineSeries<\_ChartData, DateTime>>[                  LineSeries<\_ChartData, DateTime>(                    dataSource: \_chartData, // Fonte de dados do gráfico                    xValueMapper: (\_ChartData data, \_) =>                        data.time, // Mapeia o eixo X para a hora                    yValueMapper: (\_ChartData data, \_) =>                        data.value, // Mapeia o eixo Y para o valor                    onRendererCreated: (ChartSeriesController controller) {                      \_chartSeriesController =                          controller; // Inicializa o controlador da série                    },                  )                ],              ),            ),          ],        ),      );    }  }  // Classe para representar os dados do gráfico  class \_ChartData {    \_ChartData(this.time, this.value);    final DateTime time; // Hora do dado    final double value; // Valor do dado  }  **Print Execução:** |
|  |

|  |
| --- |
| **Discussão/reflexão do Aplicativo para Visualização Dinâmica de Dados em Flutter, aplicado à Smart Grid** |
| * Descrição das discussões/reflexões (prática/mundo real) que surgiram no grupo:   **Discussões:**  As reflexões que surgiram no grupo:  **Reflexões:**  Após as reflexões em grupo, chegamos em alguns pontos importantes, sendo: |

Parabéns por concluir o curso e por se dedicar ao desenvolvimento de habilidades em IoT na área de Energia!