



FUTURO DO TRABALHO, TRABALHO DO FUTURO

***EXPLORANDO A
INTEGRAÇÃO DE IOT
COM FLUTTER PARA
SMART GRID***

Apostila do aluno



MINISTÉRIO DA
CIÊNCIA, TECNOLOGIA
E INovação

GOVERNO FEDERAL
BRASIL
UNião e RECONSTRUÇÃO

Bruno Rodrigues – Analista de Capacitação Técnica

Priscila Santos – Pesquisadora de Energia

Rita Barbosa – Analista de Capacitação Técnica

Autores da apostila

Larissa Jessica Alves – Analista de Suporte Pedagógico Sr.

Revisão da apostila

Fit Instituto de Tecnologia

Sorocaba, julho de 2024

Autor(a)



Bruno Rafael Santos Rodrigues é Pós-graduado em Redes de Computadores pela UNICAMP e em Práticas do Project Management Institute (PMI) pelo Senac Sorocaba. Possui graduação em Engenharia de Computação pela Faculdade de Engenharia de Sorocaba (2009). Atualmente, é Analista de Capacitação Técnica no FIT - Flextronics Instituto de Tecnologia, onde ministra treinamentos na área de IoT, desenvolvimento Mobile e Computação em Nuvem. Tem experiência como professor no Senac e na Faculdade de Engenharia de Sorocaba (Facens). Também é instrutor da Cisco na plataforma NetAcademy.

Mais informações em: <http://lattes.cnpq.br/8416008762320926>



Rita de Cassia A. Barbosa, Pós Graduada em Gestão de Negócios em TI pela UNIP São Paulo. Graduada em Tecnologia em Processamento de Dados pela Faculdade de Informática Tibiriçá de São Paulo. Atua no FIT como Analista de Capacitação e atuou como docente de Ensino Superior na CEUNSP – Salto (Universidade ruzeiro do Sul).

Mais informações em: <https://www.linkedin.com/in/rita-barbosa-074641/>



Priscila Santos é Mestra em Energia pela UNICAMP (2017) e Doutoranda pela UNICAMP na área de Eletrônica. É graduada pela Universidade Federal do Paraná (2014). Tem experiência na área de projetos de pesquisa e desenvolvimento nas áreas de energia, eletrônica e agroenergia. Já formou mais de 10 mil alunos em treinamentos na área de energia solar

fotovoltaica.

Mais informações em: <https://br.linkedin.com/in/priscilalvesdosantos>

APRESENTAÇÃO

A presente apostila é um instrumento teórico que complementa o curso de capacitação de IoT com *Flutter* para *Smart Grid*, com ênfase no desenvolvimento de uma aplicação mobile em *Flutter*.

Na curso analisaremos um projeto de IoT desenvolvido em um simulador online (Wokwi), que inclui o dispositivo Esp32 com o sensor DHT22. Os dados gerados serão transmitidos para um Broker via protocolo MQTT. Após, será desenvolvido um aplicativo móvel que importará os dados deste Broker MQTT e os transformará em gráficos para facilitar a leitura e interpretação.

Este material é baseado em artigos científicos, periódicos, revistas e livros acadêmicos. É extremamente recomendável que o aluno, durante ou após a leitura de cada seção, realize os exercícios propostos e accesse os materiais indicados nas referências bibliográficas para aprofundar o conhecimento e complementar o que foi lido aqui.

Desejo a você, prezado aluno, um excelente curso!!!

Boa Leitura!!

Sumário

1 INÍCIO DA JORNADA – SMART GRID	6
1.1 SMART GRID, SMART ENERGY & SMART CITYS.....	7
2 Como funciona uma Rede Inteligente?.....	9
2.1 Controle de Redes	11
2.2 Desafios, Evolução, Tendências de mercado interno e oportunidades de trabalho.....	13
2.3 Integração de fontes de energia renovável.....	15
2.4 IoT e Smart Grid	16
2.5 Monitoramento Inteligente de Smart Grids com IoT: Aplicativo em <i>Flutter</i> Utilizando MQTT para Soluções Inovadoras	17
2.6 Conectando o Mundo Físico: Explorando IoT com ESP32 e Sensor DHT22	19
2.7 Miniprojeto do Sensor de temperatura e unidade (DHT22)	20
3 Desenvolvimento de Aplicativos IoT com <i>Flutter</i>: Transformando dados em ações	21
3.1 História.....	21
3.2 Explorando o Framework <i>Flutter</i>.....	22
3.3 Instalação do <i>Flutter</i>	23
3.4 Instalação do Android Studio – Windows.....	25
3.5 Instalação das extensões do <i>Flutter</i> e Dart	29
3.6 Emulador.....	31
3.7 Instalação e Configuração do VS Code.....	40
3.8 Configuração do <i>Flutter</i>- windows	43
3.9 Desenvolvimento online usando a ferramenta Project IDX47	
3.10 Criando seu primeiro projeto	48
4 <i>Flutter</i> domindo o mundo Energético: Widgets e UI com Aprendizagem Baseada em Projeto.....	54

4.1	Histórico Dart	64
5	<i>Flutter: conectando e manipulando dados em tempo real</i>	75
5.1	Construindo gráficos no mundo Energético	75
5.2	Aplicativo básico com lista	78
5.3	Aplicativo MQTT básico.....	83
5.4	Aplicativo para criar um gráfico de linha	92
5.5	Flutter Gráfico e MQTT: Visualização Dinâmica de Dados em Ação	100
6	<i>Atividade final do curso com Aprendizagem Baseada em Projeto</i>	
	108	
	Conclusão	109
	Referências	110

1 INÍCIO DA JORNADA – SMART GRID

O *Smart Grid*, traduzindo para o português significa Redes Inteligentes. No contexto técnico, as Redes Inteligentes são canais de comunicação bidirecional eficientes entre o fornecedor e o consumidor, otimizando o uso de recursos e melhorando a confiabilidade do sistema, apoiando a integração. Uma das características das Redes Inteligentes está na capacidade de responder rapidamente as mudanças na demanda e oferta, além disso são fundamentais para a criação de um futuro sustentável e resiliente.

O surgimento sobre o assunto de *Smart Grid* se deve a dois pesquisadores e professores da *University of Minnesota Center for Electric Energy, University of Minnesota* foram: S. Massoud Amin e Bruce F. Wollenberg. Eles Começaram a estudar como a infraestrutura de rede elétrica dos Estados Unidos foi construída e distribuída. Em seus estudos identificaram que uma grande parte da rede foi desenvolvida entre as décadas de 50 e 60. Após a identificação da idade de construção das redes de energia elétrica, os pesquisadores perceberam mudanças no perfil de consumo. O estudo revelou que a estrutura da rede elétrica começou a enfrentar desafios significativos devido ao aumento da demanda de energia e à necessidade de modernização, desafios para quais a rede elétrica não foi projetada e projetada para lidar, não se pensava o quanto a tecnologia poderia evoluir e modificar as conexões e operações na infraestrutura elétrica da distribuição.

O artigo *Toward a Smart Grid: Power Delivery for the 21st Century*" publicado em 2005 foi pioneiro sobre o assunto de *Smart Grid*. Os pesquisadores Amin e Wollenberg abordaram os desafios enfrentados sobre a temática da rede elétrica norte-americana, destacando a importância de uma mudança na infraestrutura de energia, tornado-a mais segura e confiável. A abordagem sobre o assunto se deve ao crescimento desregulado e a conexão interdependente na energia elétrica, com sistemas com padrões distintos, trazendo consigo problemas no fornecimento, desregulando o transporte de energia e confiabilidade da infraestrutura da rede elétrica, dificultando a segurança energética. Os autores discutem como as sobrecargas e falhas na rede podem afetar não apenas o fornecimento de energia, mas também setores vitais que

dependem da energia como: transporte, finanças, saúde e comunicações. As falhas em cascata podem ocorrer de forma instantânea e com consequência em regiões distantes do ponto focal, trazendo problemas de infraestrutura e desregulamentação. Através de estudos foi estabelecido evidências sobre a falta de inteligência na rede de energia elétrica. A implementação desta inteligência, visa estabelecer uma segurança maior sobre a confiabilidade do sistema elétrico. O estudo reforça a necessidade de implementação de uma rede de energia inteligente tornando-a mais segura, confiável, estável e autoregulável (B. D. Russell and C. L. Benner, 2010).

1.1 SMART GRID, SMART ENERGY & SMART CITYS

Ao pesquisar o assunto sobre redes inteligentes você poderá se deparar com esses termos *Smart Grid*, *Smart Energy* e *Smart Citys*. Os termos *Smart Energy* e *Smart Citys* são derivados do *Smart Grid*, porém com suas especificações e áreas:

- *Smart Grid*: é considerada um sistema inteligente de uma infraestrutura, que através de um sistema bidirecional coleta dados, facilita a decisão sobre a resposta de uma demanda de um determinado serviço, podendo ser aplicado a energia, gás, água ou outros recursos fundamentais para uma determinada região ou cidade.
- *Smart Energy*: é maneira como a energia é distribuída e gerenciada. No setor elétrico foi titulada, pela IEA - International Energy Agency, como REI – Redes Elétricas Inteligentes, refletindo a capacidade da rede elétrica de se reestabelecer com base em mudanças de comportamento no consumo de energia ou na geração destas (IEA, 2022). Sendo uma divisão da *Smart Grid*.
- *Smart City*: é o gerenciamento inteligente de toda uma infraestrutura de uma cidade, seja na distribuição de água, energia, acionamento de iluminação pública, rede de transporte e trânsito. Os dados em tempo real em uma cidade e ao redor vem auxiliando sobre a tomada de decisão mais acertiva, principalmente sobre as mudanças

climáticas, ajudando na tomada de decisão sobre efeitos climáticos adversos.

Curiosidade sobre as redes inteligentes:

Os estudos que antecederam sobre o conceito de *smart grid*, foram estabelecidos através de um incidente com uma **aeronave F15**, que perdeu 90% da asa direita, comprometendo sua simetria e estabilidade, que normalmente faria o avião cair e capotar. O piloto desse avião F15 pousou com sucesso usando os controles restantes e critérios de empuxo do motor. Após esse incidente a aeronave foi submetida a vários testes de dinâmica de vento e controle. A **sensibilidade do piloto** de sentir a estabilidade e **ajustar os padrões** para trazer o avião para solo, despertaram o interesse dos pesquisadores que durante os **anos de 1985 - 98** desenvolveram projetos pertinentes sobre otimização e controle, uso de Inteligência artificial, para um controle de voo inteligente, esse evento forneceu um ferramentas para estudos em diversas áreas... como a de energia elétrica

2 Como funciona uma Rede Inteligente?

Qual é a principal diferença entre a nossa atual rede e uma Rede Inteligente (*Smart Grid*)?

O sistema elétrico forma uma cadeia de abastecimento único, ou seja, toda geração de energia elétrica é consumida de forma instantânea. A comparação entre a energia elétrica e o abastecimento de água é bastante pertinente para ilustrar a importância da gestão de recursos e o uso das Redes Inteligentes. Imagine que nesse contexto a empresa de fornecimento de água da sua cidade fornece de forma contínua água e em sua residência não tivesse uma torneira, o que aconteceria com esse recurso? Toda a água fornecida deverá ser usada em tempo real, independente do horário, já imaginou o desperdício desse recurso?

Se imaginarmos que a energia elétrica é como a água fornecida continuamente por uma empresa, a ausência de uma "torneira" em casa significaria que toda a energia deveria ser consumida instantaneamente, independentemente da necessidade ou do momento. Chamamos isso no setor elétrico de 'apenas em sistema de entrega de tempo', tudo que é gerado e entregue em tempo real com ou sem desperdício. Para operar o sistema elétrico com segurança desta forma, requer soluções sofisticadas e inteligentes, com sistemas de controle além da supervisão qualificada e intervenções de engenheiros de controle.

A principal diferença entre a nossa rede atual e uma rede inteligente é a forma como a geração e a demanda serão mantidos em equilíbrio, a analogia da torneira descrita anteriormente. Antes da implementação das Redes Inteligentes, as concessionárias, geradoras e transmissoras de energia, atuavam no sistema elétrico de forma reativa, ou seja, a partir de uma perturbação alertada por telefonema, ou aviso de equipamento com um problema em uma subestação ou na geração. A partir da implementação dos sistemas inteligentes foi possível através dos dados coletados em tempo real, monitorando o que acontece em cada ponto da rede elétrica, as perturbações, falhas e inconsistências que podem ocorrer. (Russell & Benner, 2010).

Em nosso sistema elétrico brasileiro, temos um grande desafio, pois ainda temos uma dependência dos recursos hídricos e por consequência nas

questões do clima, pois sem chuvas nas cabeceiras dos rios, os reservatórios das usinas hidrelétricas diminuem. Em consequência deste efeito, as reações são geradas no abastecimento de energia, impactando para os consumidores o acréscimo na conta energia através das bandeiras tarifárias amarela e vermelha um encargo por acionar outras fontes de geração de combustíveis fosseis, para manter a demanda de energia existente. Com a aplicação de redes inteligentes na geração atreladas a coleta de dados e a previsão sobre mudanças climáticas, o gerenciamento do acionamento de outras fontes pode ser estudado com antecedência, para diminuir os custos dos encargos das bandeiras tarifárias.

O sistema de *Smart Grid* em território nacional vem sendo implementado de forma gradativa, isso se deve aos custos de modernização e adaptação da rede. Uma aplicação de *Smart Grid* que temos em território nacional é o SIN (Sistema Interligado Nacional), que gerencia as geradoras de energia elétrica, as linhas de distribuição e as demandas das distribuidoras de energia por localidade.

O que precisa ser controlado no sistema elétrico? Os três principais dados que precisam ser controlados de forma primária são:

- **Frequência:** isso é feito combinando geração e demanda segundo a segundo para garantir a estabilidade do sistema e garantir que todos recebam eletricidade em frequência constante.
- **Tensão:** isso é feito usando muitos dispositivos de controle, principalmente geradores e transformadores, em todo o sistema elétrico para garantir que as tensões permaneçam estáveis e que os clientes receberem sua eletricidade dentro de limites especificados.
- **Corrente:** cada dispositivo e circuito na rede tem um valor superior limite à corrente que ele pode transportar sem danos ou falha.

A rede, portanto, deve ser projetada de modo que estes limites são respeitados em todos os momentos, mesmo quando eventos de falha ocorrer. Isto é feito fornecendo capacidade ociosa na rede juntamente com ações de controle e proteção.

2.1 Controle de Redes

A rede elétrica é controlada a partir da demanda de energia existente na ponta, ou seja entre os consumidores residenciais, rurais e industriais. A partir da demanda as distribuidoras de energia encaminham essa informação ao SIN que estabelece qual geradora de energia e rede de transmissão irá fornecer essa energia. A dificuldade no controle do fornecimento de energia elétrica não está na trajetória da alta tensão, pois já conta com sensores e sofisticados sistemas de controle. Mas está na previsibilidade da demanda em tensões mais baixas em nossas redes de distribuição. O gerenciamento da rede da distribuição de energia, requer a demanda necessária naquele momento, o gerenciamento de cargas prioritárias na cidade ou região, principalmente hospitais e indústrias.

O controle por meio das redes inteligentes acontece através do:

- **Fluxo Bidirecional:** ao contrário das redes tradicionais, as *Smart Grids* permitem o fluxo bidirecional de energia. Isto significa que podem fornecer eletricidade a residências e empresas e, também distribuir energia gerada por fontes como painéis solares ou outros geradores
- **Controle Dinâmico:** as *Smart Grids* são inteligentes porque podem controlar dinamicamente as entradas e saídas de energia. Eles se adaptam às mudanças nas condições, garantindo uma distribuição eficiente de energia. Por exemplo, durante o dia, o excesso de energia gerado pelos painéis solares pode ser armazenado ou realimentado na rede. À noite, quando a procura excede a oferta, a energia armazenada pode ser utilizada
- **Inteligência Artificial (IA):** a IA pode ajudar a *Smart Grid* a ser mais eficiente, confiável e segura, além de permitir novos serviços e funcionalidades para os consumidores. Uma das principais maneiras pelas quais a IA pode ser útil para a *Smart Grid* é na previsão de demanda de energia. A IA pode analisar dados históricos de consumo de energia e dados meteorológicos para prever a demanda futura de energia elétrica. Isso pode ajudar as empresas de energia a planejar melhor sua produção de energia, evitando picos de demanda e

reduzindo a necessidade de fontes de energia não renováveis durante esses períodos. Outra aplicação da IA na *Smart Grid* é a detecção e diagnóstico de falhas na rede elétrica. A IA pode ser usada para analisar dados de sensores em tempo real para detectar problemas na rede elétrica, como falhas em transformadores ou linhas de transmissão. Isso pode permitir que as empresas de energia identifiquem e corrijam problemas mais rapidamente, reduzindo o tempo de inatividade para os consumidores e aumentando a confiabilidade da rede elétrica.

- **Monitoramento e controle de rede inteligentes:** fornecerá o meio de menor custo para fornecer a capacidade de atender a esse padrão muito diferente e dispositivos inovadores, como pode-se esperar que a eletrônica de potência melhore a utilização de planta de rede existente.
- **Implantação de contadores inteligentes e a introdução da automação residencial** (por exemplo, controles interno de energia que decidirá quando os aparelhos domésticos funcionam com base no preço da eletricidade em tempo real), muitos consumidores serão encorajados a gerenciarativamente sua demanda de energia para ajudar a equilibrar o sistema elétrico, mudando a curva de carga de consumo de energia.

Portanto, a principal razão pela qual precisaremos de uma rede mais inteligente é para ajudar as empresas da rede a atender às necessidades dos consumidores de forma mais rápida e econômica, acomodando uma crescente previsível pela procura de eletricidade e absorvendo geração embutida nos níveis mais baixos do sistema, os consumidores finais.

2.2 Desafios, Evolução, Tendências de mercado interno e oportunidades de trabalho.

Existem alguns desafios a serem enfrentados com a implementação de tecnologias considerando as inovações que ocorrem no setor de energia elétrica, Entre elas os desafios são:

- **Armazenamento de energia:** um desafio é a incompatibilidade entre os tempos de geração e consumo de energia. Por exemplo: os painéis solares geram mais energia durante o dia, mas o pico de consumo ocorre à noite. As Smart Grids resolvem isso usando tecnologias como: baterias, capacitores e super capacitores para armazenamento de energia.
- **Microgeração:** com o aumento da microgeração (produção de energia em pequena escala), as Smart Grids acomodam fontes de energia descentralizadas, como painéis solares em telhados e turbinas eólicas.
- **Medição Inteligente:** os medidores inteligentes desempenham um papel crucial na monitorização da utilização de energia, permitindo aos consumidores monitorizar o seu consumo e tomar decisões sobre o uso e preço da energia elétrica.
- **Ao nível da distribuição:** os desafios estão relacionados ao potencial de milhões de clientes conectarem novas cargas, como a de veículos elétricos, sistemas de armazenamento e conectar geradores para que eles consumam e produzam sua eletricidade. Isto exigirá uma abordagem completamente diferente para a forma como o sistema é controlado e esta mudança está no cerne da Rede Inteligente
- **Ao nível da transmissão:** os desafios incluem a conexão de geração Eólica offshore, hidrogênio verde, geração híbrida de grande porte e novas interligações com outros países.

Apesar dos diversos desafios para a implementação das Redes Inteligentes, os benefícios são inúmeros, como a melhoria da eficiência

energética, permitindo uma comunicação bidirecional entre as concessionárias de energia e os consumidores. Isso possibilita um monitoramento mais preciso do consumo e a identificação de padrões, o que pode levar a uma redução no uso excessivo de energia. Além disso, as *Smart Grids* contribuem para a redução de perdas na transmissão de energia, otimizando as operações e minimizando o desperdício. Outro ponto positivo é a integração facilitada de fontes de energia renovável, como Solar e Eólica, que podem ser gerenciadas de maneira eficiente para garantir um fornecimento estável.

A automação da distribuição de energia é outro aspecto relevante, permitindo a redistribuição instantânea de energia conforme a demanda e minimizando interrupções. Por fim, a implementação de *Smart Grids* pode resultar em economia para os consumidores, através de uma gestão mais eficaz e do uso de medidores inteligentes que comunicam informações em tempo real.

Os principais benefícios das redes *Smart Grids* são:

- **Eficiência:** as *Smart Grids* otimizam a distribuição de energia, reduzindo perdas e melhorando a eficiência geral.
- **Confiabilidade:** ao detectar falhas e redirecionar a energia, as *Smart Grids* melhoram a confiabilidade da rede.
- **Sustentabilidade:** promovem a integração das energias renováveis e reduzem o impacto ambiental.
- **Economia de custos:** o gerenciamento eficiente de energia leva à economia de custos de manutenção tanto para os serviços públicos quanto para os consumidores.

No Brasil já possui a implementação de projetos de *Smart Grid* especialmente em áreas urbanas, para melhorar a gestão de energia e reduzir perdas. Com a integração de veículos elétricos, o uso de Inteligência Artificial e a crescente demanda por energia verde. As empresas têm a oportunidade de modernizar suas redes elétricas e fornecer serviços mais eficientes e confiáveis para seus consumidores.

É importante ressaltar que a implementação de *Smart Grids* no Brasil pode trazer diversos benefícios para a economia do país, além de contribuir para a redução das emissões de gases de efeito estufa. Às empresas que se

prepararam para essa transição estarão mais bem posicionadas para aproveitar esses benefícios e fornecer serviços de energia elétrica mais modernos, eficientes e sustentáveis para seus consumidores.

Portanto, a Rede existe para fornecer conexões entre geradores e consumidores de eletricidade. Os benefícios da implementação de um sistema *Smart Grid* fornece serviços que no futuro serão imprescindíveis para os consumidores, dentre eles estão: segurança da rede inteligente, sustentabilidade e proteção do sistema contra ataques cibernéticos.

2.3 Integração de fontes de energia renovável

A crescente demanda por energia verde é uma das tendências mais significativas na indústria de energia atualmente.

Com a preocupação crescente das mudanças climáticas e a necessidade de reduzir as emissões de gases de efeito estufa, muitos consumidores e empresas estão procurando fontes de energia renovável, como a energia Solar e Eólica. A *Smart Grid* pode ser um elemento chave no gerenciamento dessa demanda crescente por energia verde. Uma das principais vantagens é que ela pode integrar fontes de energia renovável na rede elétrica, tornando mais fácil e eficiente o acesso à energia verde para os consumidores.

Espera-se que as soluções inteligentes ofereçam formas econômicas para:

- Facilitar e melhorar a conexão e operação de baixa e geradores de carbono zero.
- Facilitar o envolvimento da resposta à procura na operação do sistema elétrico.
- Permitir o crescimento da procura de eletricidade,
- Minimizar ao mesmo tempo o fornecimento de nova capacidade de rede.
- Reduzir o impacto do carbono da própria rede.
- Minimizar perdas.

À medida que a implantação crescer em todo o mundo nos próximos anos, estarão disponíveis dados que permitirão avaliação dos benefícios de soluções específicas de redes inteligentes.

2.4 IoT e Smart Grid

A Internet das Coisas (IoT) é conhecida por conectar equipamentos e sensores à internet. Esses dispositivos são conectados através de uma rede de sensores, que coleta diversos valores parametrizados e os processa para ações subsequentes. Para estabelecer a comunicação sequencial entre os dispositivos, o sistema é estruturado em três campos principais:

- **Campo de visualização:** consiste em redes de sensores, leitores de dados, entre outros, e é responsável pela coleta e alimentação dos dados.
- **Campo de rede:** inclui redes de telecomunicações, como redes móveis, redes ópticas, Wi-Fi, etc. Esta seção é responsável pela transmissão dos dados.
- **Campo de aplicação:** é a principal área do IoT, onde os dados são analisados e processados para que ações apropriadas sejam tomadas, garantindo que os dispositivos executem suas funções conforme a recepção dos dados.

Através dos avanços tecnológicos na IoT têm desempenhado um papel crucial na resolução de um dos maiores desafios do setor de energia: o aumento da demanda e consumo de energia elétrica. Para atender a essa crescente demanda, é necessário não apenas aumentar a produção com os recursos disponíveis, mas também adotar soluções complementares, como a conservação de energia. A aplicação do IoT em toda a cadeia de geração, transmissão e distribuição de energia é uma dessas soluções, permitindo otimizar a demanda de forma eficiente. Nesse contexto de evolução do IoT, a infraestrutura de rede inteligente se destaca como a principal conectividade entre o ponto de geração e a unidade consumidora (cliente final). Essa conectividade abrange residências, indústrias, veículos elétricos, consumo público,

eletrodomésticos inteligentes, entre outros, integrando todo o sistema de energia.

A transformação da rede elétrica em uma rede inteligente visa garantir o equilíbrio entre a geração e o consumo de energia, preservando assim os recursos naturais. Podemos fazer uma analogia com uma torneira: mantê-la aberta ou fechada conforme a necessidade de uso dos recursos.

Com a aplicação do IoT dentro das redes inteligentes podemos:

- Realizar a leitura automática do medidor de energia, estabelecendo a demanda de energia necessária por microrregião ou região.
- Rastrear em tempo real de Linha de Transmissão e Distribuição para manutenção ou falhas, a fim de restabelecer o sistema sem prejudicar os consumidores de energia.
- Utilizar o Sistema de Gestão de Carregamento de Veículos Elétricos e armazenamento, através do gerenciamento de microredes, estabelecendo o fluxo de energia no sistema de distribuição.

2.5 Monitoramento Inteligente de *Smart Grids* com IoT: Aplicativo em *Flutter* Utilizando MQTT para Soluções Inovadoras

O conceito de *Smart Grid* (Rede Inteligente) está transformando a forma como gerenciamos e utilizamos a energia elétrica, integrando tecnologias avançadas de comunicação e informação. Uma das principais inovações nessa área é a aplicação da Internet das Coisas (IoT), que permite a coleta e análise de dados em tempo real, melhorando a eficiência e a confiabilidade das redes elétricas.

A Internet das Coisas (IoT) tem revolucionado diversos setores, e um dos mais impactados é o setor de energia elétrica, especialmente com o desenvolvimento de smart grids. As redes inteligentes (smart grids) são redes elétricas que utilizam tecnologias de comunicação e informação para otimizar a produção, distribuição e consumo de energia elétrica. Uma das principais vantagens das smart grids é a capacidade de monitorar e gerenciar em tempo real o consumo de energia, permitindo uma gestão mais eficiente e sustentável.

Para aproveitar todo o potencial das *Smart Grids*, foi desenvolvido um aplicativo em *Flutter* que permite o monitoramento em tempo real dessas redes usando o protocolo MQTT (Message Queuing Telemetry Transport).

O MQTT é um protocolo de comunicação leve e eficiente, ideal para a transmissão de dados em redes IoT, que oferece alta confiabilidade e baixa latência, essenciais para o monitoramento contínuo e em tempo real das Smart Grids. Um exemplo prático dessa integração será apresentado durante este curso. Utilizaremos um ESP32, um microcontrolador de alta performance com conectividade *Wi-Fi* e *Bluetooth*, junto com um sensor DHT22, que mede temperatura e umidade. Este sistema será capaz de monitorar as condições ambientais em diferentes pontos da rede elétrica, fornecendo dados essenciais para a otimização do consumo e a manutenção preventiva.

Os dados coletados pelo ESP32 com o sensor DHT22 são enviados a um broker MQTT, um protocolo leve de comunicação utilizado para a transmissão de mensagens em IoT. O MQTT facilita a publicação e a subscrição de mensagens, permitindo que diferentes dispositivos e aplicações accessem os dados de forma eficiente e em tempo real.

Com base nesses dados enviados ao broker MQTT, nosso aplicativo em *Flutter* proporcionará uma interface amigável e intuitiva para os usuários, permitindo que eles acompanhem em tempo real o desempenho da rede elétrica, identifiquem falhas ou anomalias rapidamente e tomem decisões informadas para otimizar o uso de energia. Além disso, o aplicativo integrará diversas funcionalidades inteligentes, como alertas automáticos em caso de desvios significativos no consumo de energia, análises preditivas para antecipar problemas e sugestões de ações corretivas baseadas em dados históricos.

A inovação e a inteligência incorporadas no nosso aplicativo *Flutter*, aliadas ao protocolo MQTT, não apenas simplificarão o monitoramento das smart grids, mas também promoverão uma gestão mais sustentável e eficiente da energia elétrica. Isso resultará em redução de custos, diminuição do impacto ambiental e melhoria da confiabilidade do fornecimento de energia.

Para concluir nossa jornada, no último dia do curso, realizaremos uma discussão e reflexão sobre a aplicabilidade da solução desenvolvida em *Flutter*. Esta solução gerou gráficos a partir de um projeto de IoT voltado para redes inteligentes. A solução inovadora que estamos desenvolvendo tem o potencial

de transformar a forma como as redes elétricas são gerenciadas, tornando-as mais resilientes, eficientes e preparadas para os desafios futuros. Com o nosso aplicativo, empresas e consumidores poderão ter um controle muito mais preciso sobre o uso da energia, contribuindo para um futuro mais sustentável e inteligente.

2.6 Conectando o Mundo Físico: Explorando IoT com ESP32 e Sensor DHT22

A Internet das Coisas (IoT) é um conceito que descreve a interconexão de dispositivos físicos através da internet, permitindo que eles coletem, compartilhem e analisem dados. Esses dispositivos podem variar desde objetos cotidianos, como eletrodomésticos e carros, até sensores industriais e sistemas de controle ambiental. A IoT possibilita a automação e o monitoramento em tempo real, proporcionando maior eficiência, segurança e conveniência em diversas áreas, como saúde, transporte e energia.

Na busca contínua por soluções mais eficientes e sustentáveis no setor energético, a medição precisa de variáveis ambientais, como temperatura e umidade, desempenha um papel fundamental. O Sensor DHT22, também conhecido como AM2302, se destaca como uma ferramenta valiosa para a coleta de dados ambientais confiáveis. Ele fornece tanto temperatura quanto umidade do ar instantaneamente, utilizando um sensor capacitivo de umidade e um termistor para medir o ar circundante, ambos conectados a um controlador de 8 bits que produz um sinal digital serial no pino de dados (Data).

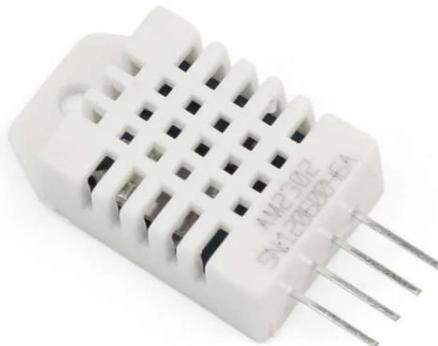


Figura 1 - Sensor de Temperatura e Umidade DHT22. Fonte: Vida de Silicio.

2.7 Miniprojeto do Sensor de temperatura e umidade (DHT22)

Este miniprojeto tem seu funcionamento ativado com 3 a 6 VDC (Volts em Corrente Contínua), têm sua escala de temperatura entre 40 a 80°C e de 0-100% para a umidade relativa do ar (UR), apesar dessas características sua imprecisão é de $\pm 0,2^{\circ}\text{C}$ na faixa de -20 a 60°C para a temperatura do ar e para outras faixas de $\pm 0,5^{\circ}\text{C}$. Para a umidade relativa do ar possui incerteza de $\pm 2\%$.

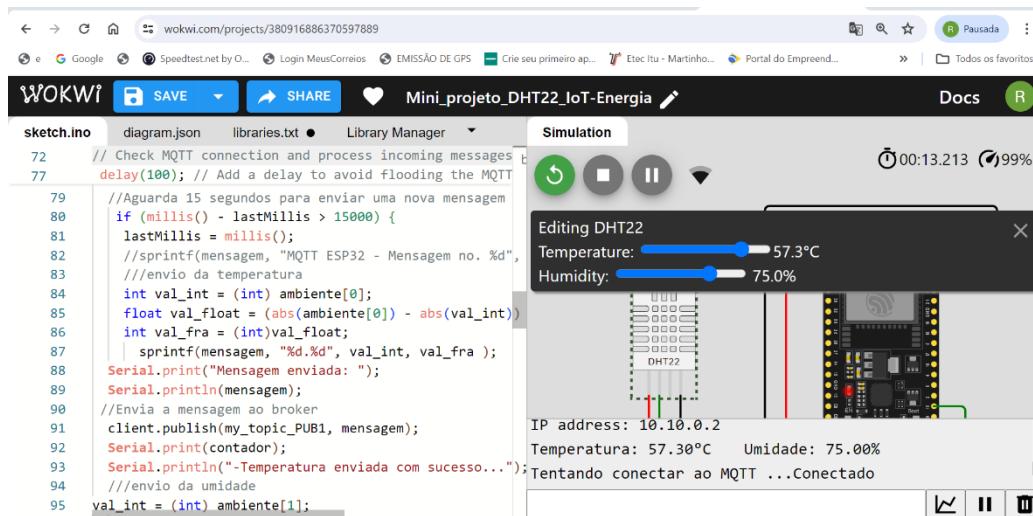


Figura 2 - Miniprojeto DHT22. Fonte: Autoria própria.

Você poderá executar o projeto no simulador Wokwi no seguinte link:

<https://wokwi.com/projects/380916886370597889>

3 Desenvolvimento de Aplicativos IoT com *Flutter*: Transformando dados em ações

O desenvolvimento de aplicativos para a Internet das Coisas (IoT) está revolucionando a maneira como interagimos com o mundo ao nosso redor. Essa tecnologia permite que dispositivos físicos se conectem à internet, coletando e compartilhando dados que podem ser transformados em ações práticas e automatizadas. Nesse cenário, o *Flutter*, um *framework* de código aberto criado pelo *Google*, surge como uma ferramenta poderosa para a criação de interfaces de usuário nativas para *iOS* e *Android* a partir de um único código-fonte. Sua capacidade de criar aplicativos rápidos, responsivos e visualmente atraentes faz dele uma escolha ideal para projetos de IoT.

Utilizando o *Flutter* para o desenvolvimento de aplicativos IoT, é possível integrar diversos sensores e dispositivos inteligentes em uma interface unificada, facilitando a interpretação dos dados coletados. A partir dessas informações, os aplicativos podem desencadear ações automáticas, como ajustar a temperatura de um ambiente, acender luzes, por exemplo. Essa transformação de dados em ações, não apenas melhora a eficiência e a conveniência, mas também abre novas possibilidades para inovação em domínios como a área de Energia. Assim, o *Flutter* não apenas simplifica o desenvolvimento de aplicativos IoT, mas também potencializa a criação de soluções inteligentes e interativas, alinhadas com as necessidades do mundo moderno.

3.1 História

A primeira versão do *Flutter* teve o codinome "Sky", era executada no sistema operacional *Android*. Em 2015, foi apresentado na cúpula de desenvolvedores *Dart* (linguagem de programação para criar aplicativos para Web) com a intenção declarada de ser capaz de renderizar consistentemente 120 quadros por segundo. Em 4 de dezembro de 2018, o *Flutter* 1.0 foi lançado no evento *Flutter Live*, substituindo a primeira versão "estável" do *Framework*. O *Flutter Release Preview 2*, foi anunciado durante a *keynote do Google Developer Days* em Xangai.

Em 06 de maio de 2020 foi lançado o *Dart SDK* na versão 2.8 e o *Flutter* na versão 1.17.0, onde foi adicionado suporte a *API Metal* (proporciona melhor desempenho no IOS). Melhorou muito o desempenho em dispositivos IOS em 50%, novos widgets do Material e novas ferramentas de rastreamento de rede. Atualmente em 2024, o *Flutter* está na versão 3.22.1 e o *Dart* na versão 3.4.1, datados em 22/05/2024.

Assim, o *Flutter* permite o desenvolvimento do código em “*Cross-Platform*”. Baseia-se no uso de um *framework* que possibilita ao programador desenvolver um código que seja executado em várias plataformas. No caso do nosso curso, será Android e iOS.

No que se refere a arquitetura do *Flutter*, na criação de um aplicativo, seu código é compilado para a linguagem base do dispositivo, ou seja, as aplicações são realmente nativas e por isso conseguem acessar recursos do dispositivo sem a “ajuda” de terceiros e com maior desempenho.

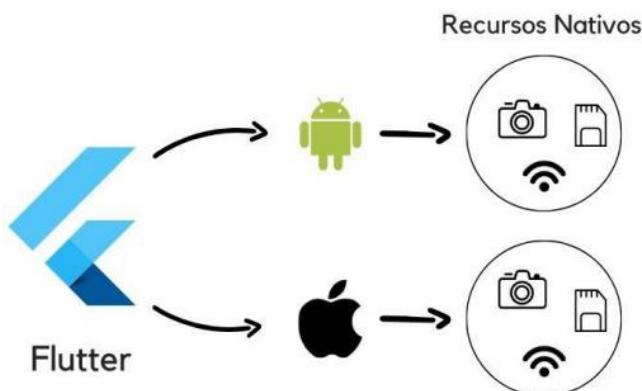


Figura 3 - Ilustração do acesso direto aos recursos nativos no Flutter. Fonte: TreinaWeb.

3.2 Explorando o Framework *Flutter*

Flutter é uma ferramenta de código aberto criada pelo *Google*. O *Flutter* utiliza a linguagem *Dart* para criar aplicativos para *Web* e vários sistemas operacionais como *Android*, *iOS*, *Windows*, *macOS*, *Linux* e *Fuchsia*.

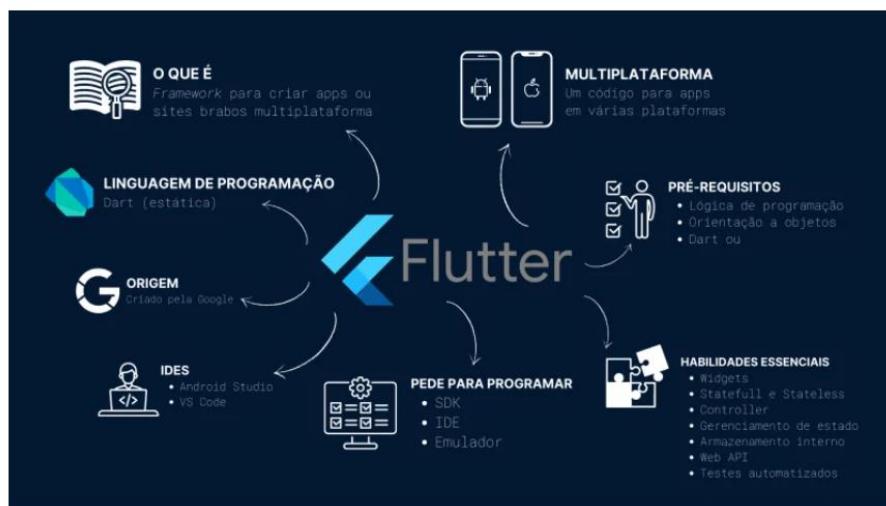


Figura 4 - O que é o Flutter. Fonte: Alura.

3.3 Instalação do Flutter

O Flutter SDK pode ser utilizado no *Windows*, *MacOS* e *Linux*. A seguir um passo a passo para instalação:

Para iniciarmos a instalação, acesse o link: <https://flutter.dev/> e clique no canto superior direito do site em “Get Started”:



Figura 5 - Acesso Flutter SDK para download. Fonte: Flutter.

Escolha o Sistema Operacional Windows:

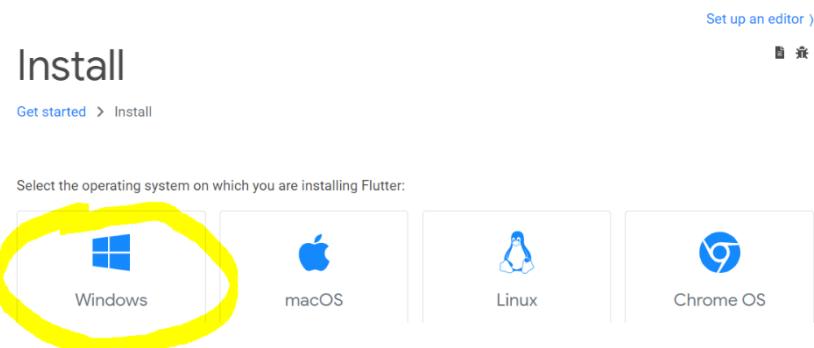


Figura 6 - Sistema operacional. Fonte: Flutter.

Importante atentar-se para os requisitos do sistema:

Para instalar e executar o Flutter, seu ambiente de desenvolvimento deve cumprir os seguintes requisitos mínimos:

- Sistema Operacional: Windows 7 SP1 ou mais atual (64-bit).
- Espaço em disco: 400 MB (não inclui o espaço em disco necessário para IDE/ferramentas).
- Ferramentas: O Flutter depende das seguintes ferramentas estarem disponíveis no seu ambiente.
 - [Windows PowerShell 5.0](#) ou mais recente (já vem pré-instalado com o Windows 10)
 - [Git para Windows](#) 2.x, com a opção Usar Git do Prompt de Comandos do Windows

Se o Git para Windows já estiver instalado, certifique-se que você pode executar comandos do `git` pelo prompt ou PowerShell.

Figura 7 - Requisitos de sistema. Fonte: Flutter.

Obtenha o SDK do Flutter

1. Baixe o seguinte pacote de instalação para obter a última release de versão estável do SDK do Flutter:

[flutter_windows_1.17.1-stable.zip](#)

Para outros canais de release, e versões mais antigas, veja o [arquivo de SDK](#).

2. Extraia o arquivo zip e coloque o `flutter` na localização desejada para a instalação do SDK do Flutter (por exemplo, `C:\src\flutter`; não instale o Flutter em um diretório como `C:\Arquivos de Programas` que requer privilégios elevados).

Figura 8 - Download Flutter SDK. Fonte: site oficial Flutter.

Verifique na sua pasta de Download se o arquivo foi salvo no formato Zip, transfira- o para outra pasta, recomendamos criar uma pasta na raiz com o nome `src` e inicie a instalação do Flutter neste diretório. O caminho final da instalação deverá ser `c:\src\flutter`.



É importante salientar que, conforme recomendação do site, procure não instalar o *Flutter* no diretório `C:\Program Files\`. Outro ponto importante, para a instalação dos softwares necessários para

acompanhamento deste curso, o aluno deverá ter acesso de administrados nas máquinas os softwares serão instalados.

3.4 Instalação do Android Studio – Windows

Segundo Developers o *Android Studio* é um IDE (ambiente de desenvolvimento integrado) para desenvolver na plataforma Android. Foi anunciado em 16 de Maio de 2013 na conferência Google I/O. *Android Studio* é disponibilizado gratuitamente sob a Licença Apache 2.0”.

Para a instalação do *Android Studio* entre no site:
<https://developer.android.com/studio?hl=pt-br>.

O site irá identificar a sua versão do *Windows*, basta apenas clicar no botão verde intitulado ”Download *Android Studio*” e siga os passos de instalação, a versão que iremos instalar é a *Android Studio Jellyfish | 2023.3.1*, conforme abaixo:

Esta é a tela de apresentação do *Android Studio*, clique em “Next”:

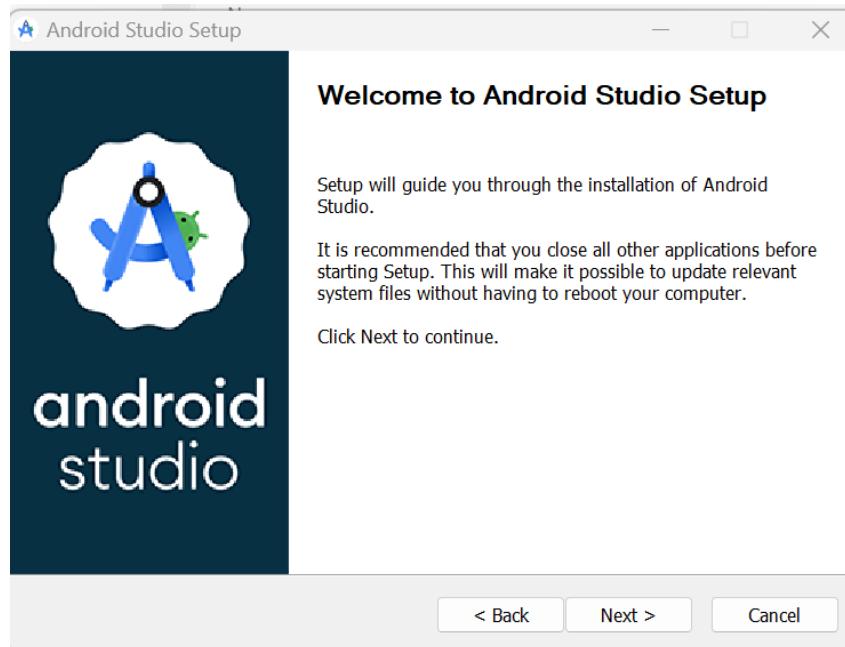


Figura 9 - Instalação do *Android Studio*. Fonte: autoria própria.

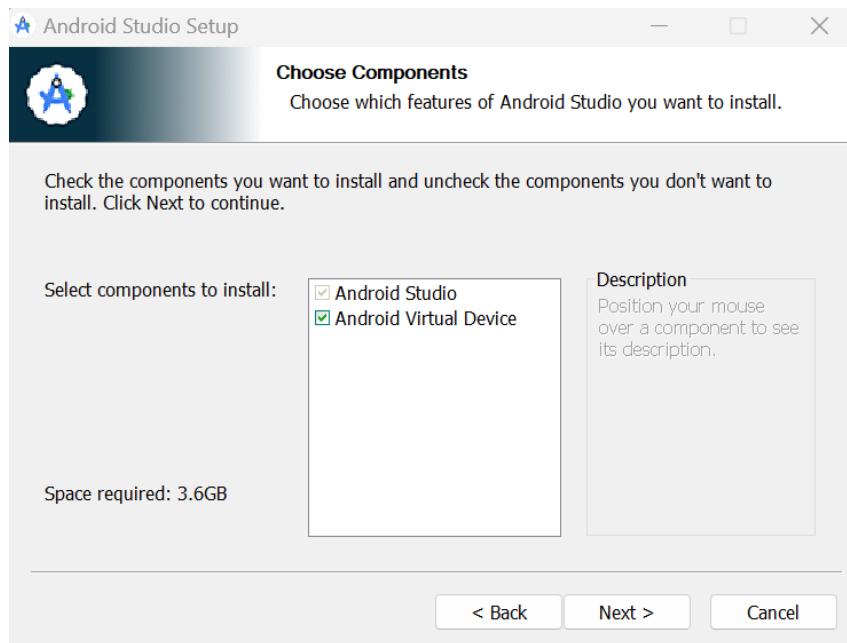


Figura 10 - Instalação do Android Studio. Fonte: autoria própria.

Na sequência, o Android Studio apresentará o caminho de instalação do produto, que deverá em C:\Program Files\Android\Android Studio. Atenção - Não modifique o caminho de instalação, pois, isso poderá comprometer o funcionamento do IDE.

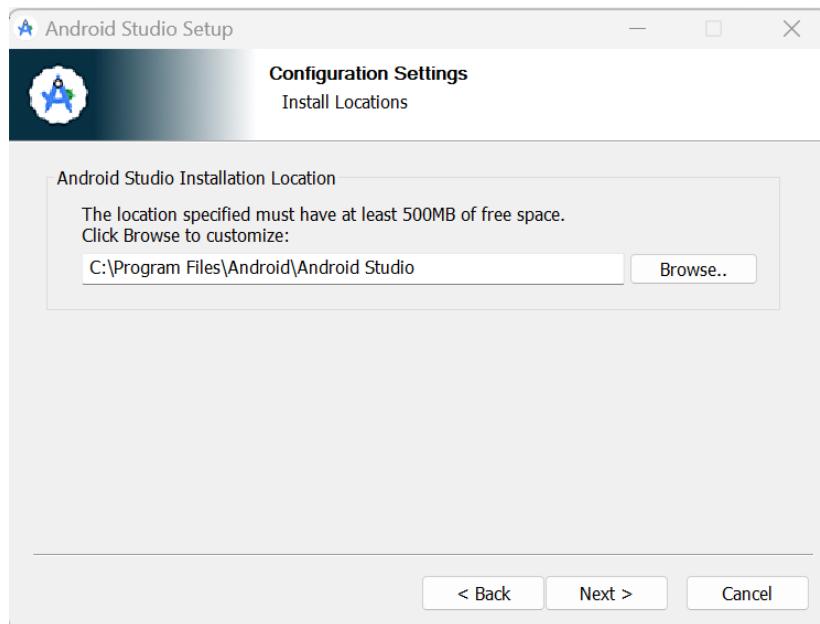


Figura 11 - Caminho de instalação do Android Studio. Fonte: autoria própria.

Selecione “Install”:

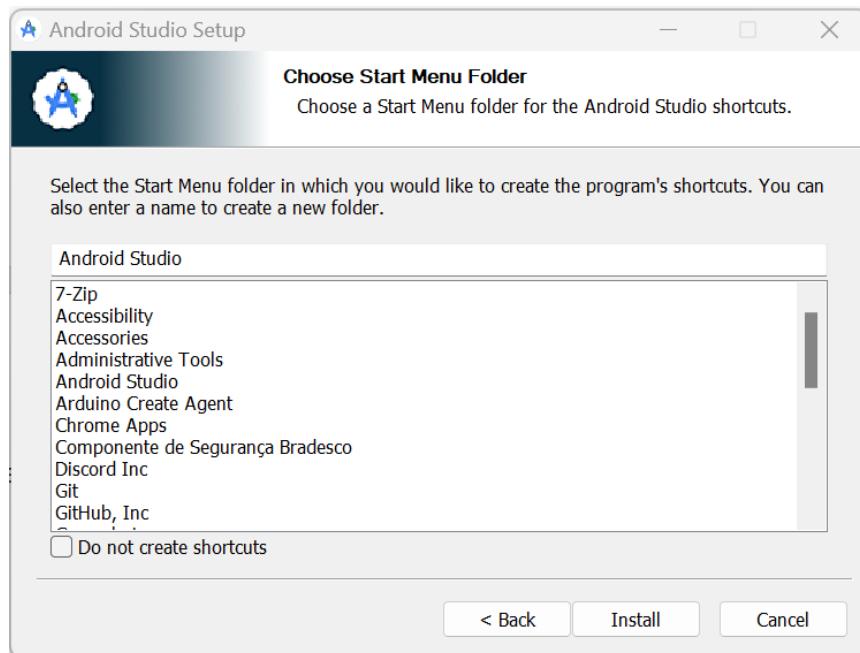


Figura 12 - Instalação do Android Studio. Fonte: autoria própria.

Clique em “Next”:

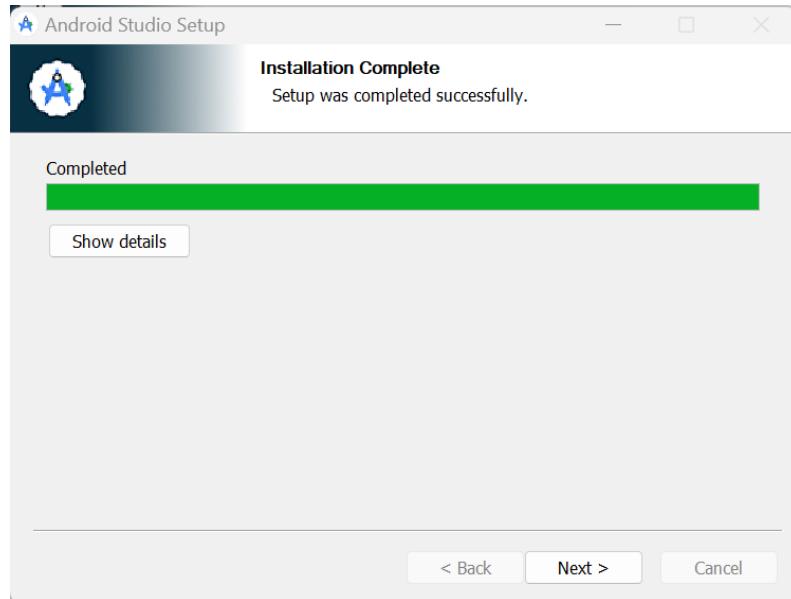


Figura 13 - Status da instalação do Android Studio. Fonte: autoria própria.

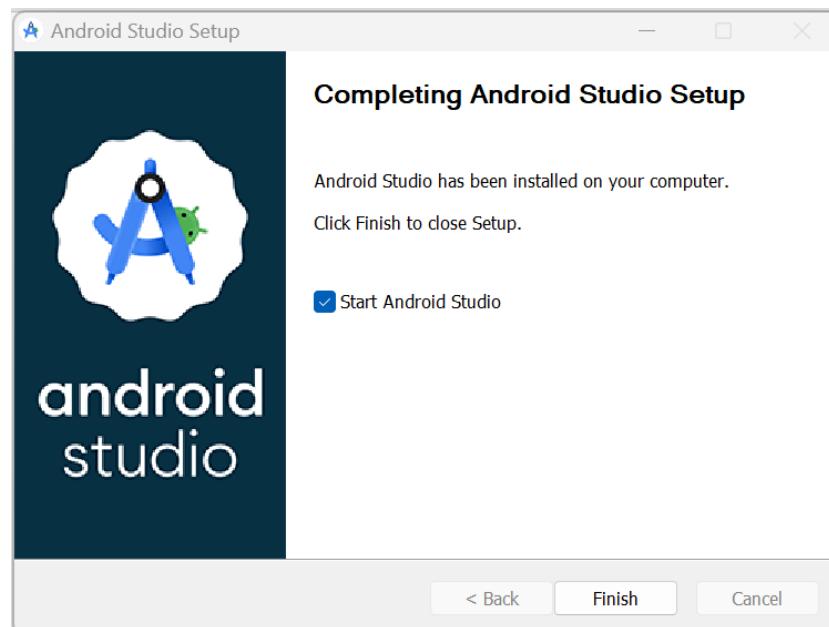


Figura 14 - Finalização da Instalação. Fonte: Developer.

Ao finalizar a instalação, o *Android Studio* será inicializado e aparecerá uma tela de apresentação dos novos recursos, bem como alertas de atualização de Plugins, clique em “Update” para realizar o download e instalação.

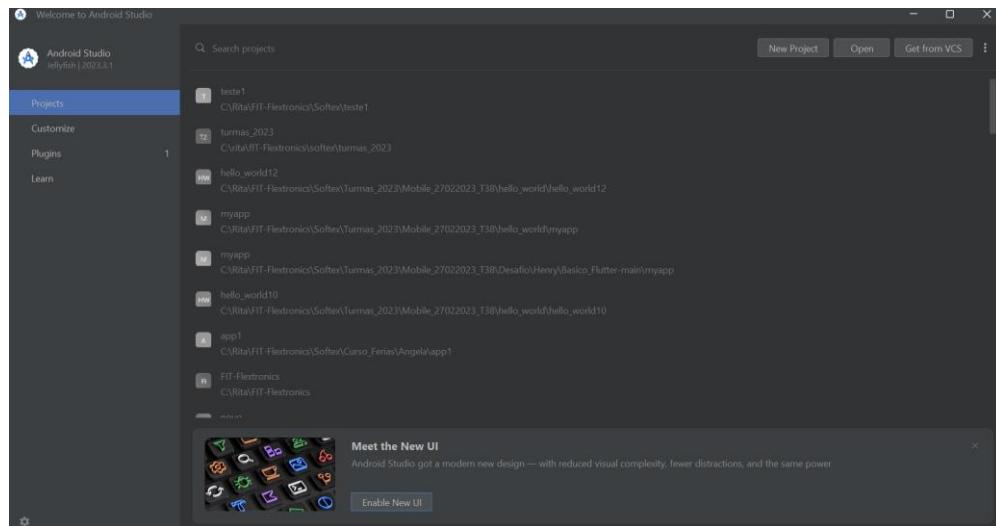


Figura 15 - Página inicial Android Studio. Fonte: autoria própria.

Ao finalizar cada *update*, clique em “Finish”. Na última atualização de plugins, reinicie se *Android Studio* para que as atualizações sejam concretizadas.

3.5 Instalação das extensões do Flutter e Dart

No menu do lado esquerdo, clique em **File**, **Settings**, **Plugins**, e verifique se os plugins do *Flutter* e *Dart* estão instalados, clique na aba “installed” para verificar. Vá até a caixa de pesquisa e digite *Flutter* e veja se está com o status “instaled”. Faço o mesmo com o *Dart*.

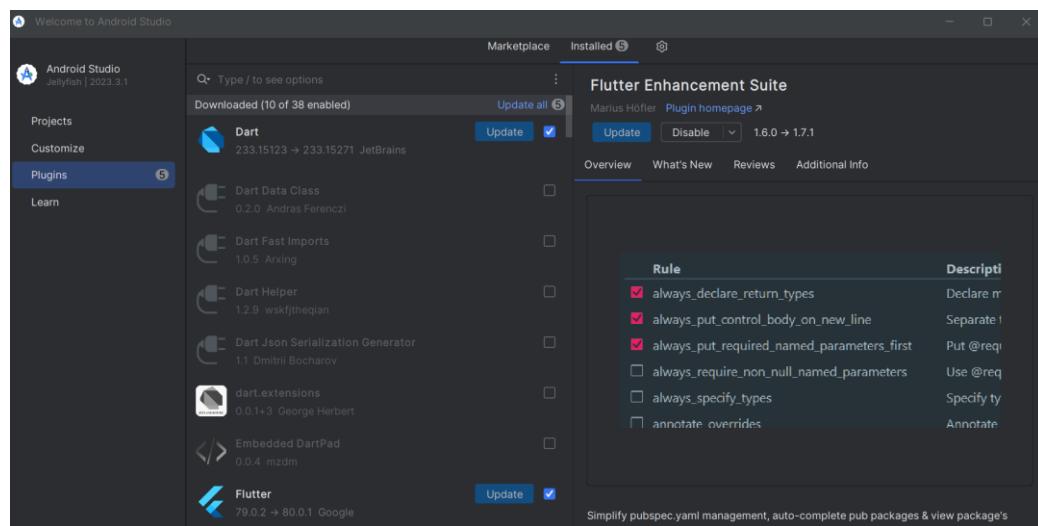


Figura 16 - Tela de instalação de Plugins. Fonte: autoria própria.

Caso não estejam instalados, pesquise por “Dart”, clique em “Install” e aguarde a instalação até o final:

Na sequência pesquise por “Flutter”, clique em “Install” e aguarde a instalação até o final:

Após a instalação, o Android Studio poderá aparecerá um botão ao lado do plugin instalado, ‘restart IDE’, clique para reiniciar seu Android Studio.

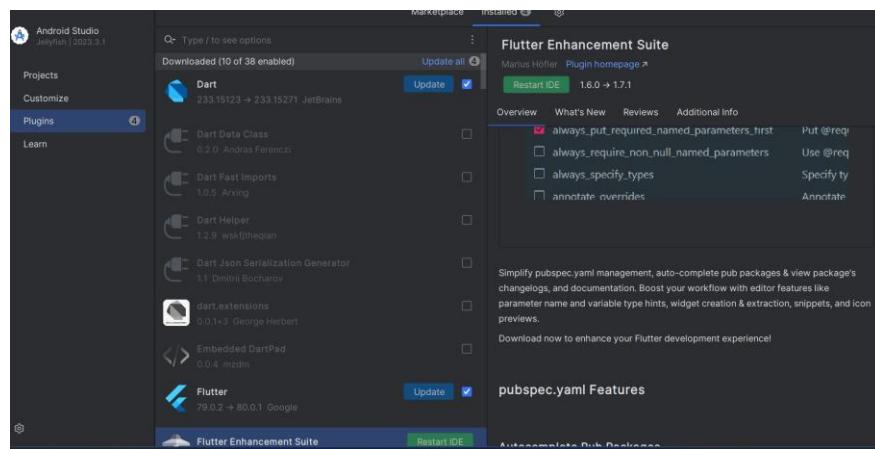


Figura 17 - Tela de instalação de Plugins. Fonte: autoria própria.

• SDK Manager

No canto superior direito, clique nos três pontos, e selecione SDK Manager.

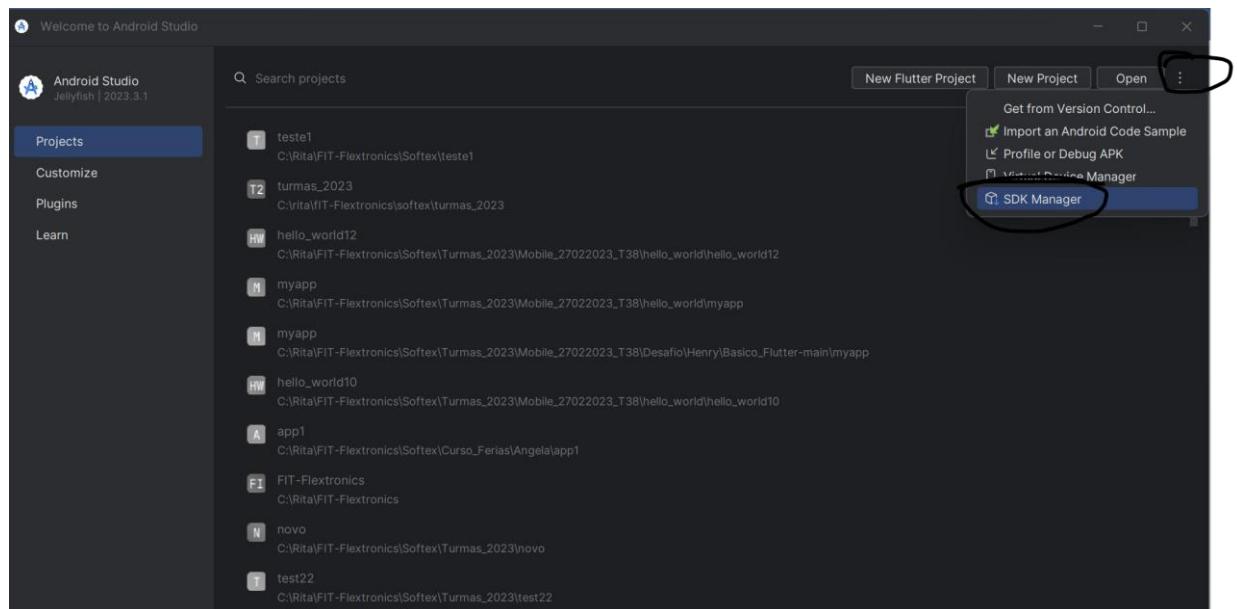


Figura 18 – SDK Manager. Fonte: autoria própria.

Na opção “SDK Tools” > Selecione a opção “Android SDK Command line Tools” e click em ok. Outras opções poderão pedir um update:

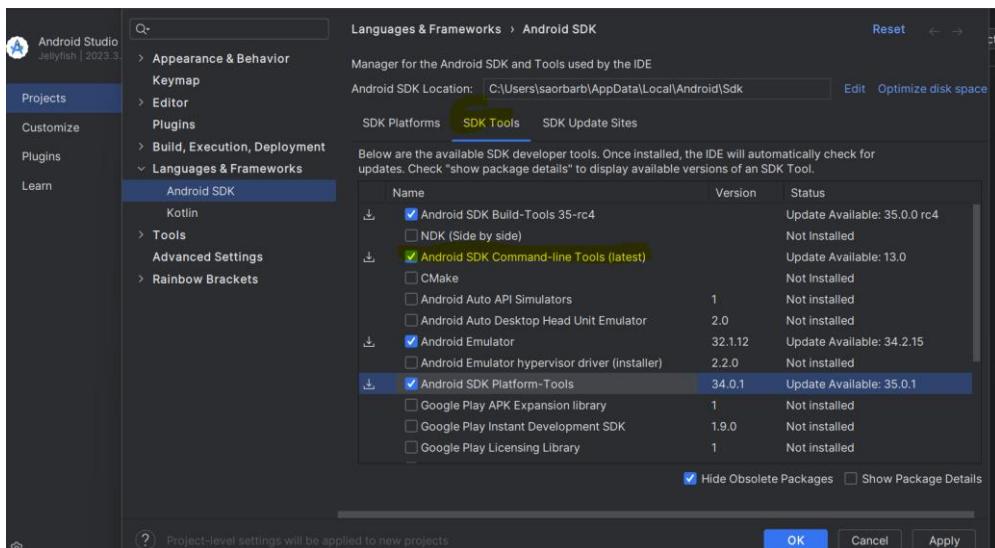


Figura 19 - SDK Tools. Fonte: autoria própria.

Clique em *Finish* para finalização dos *updates*.

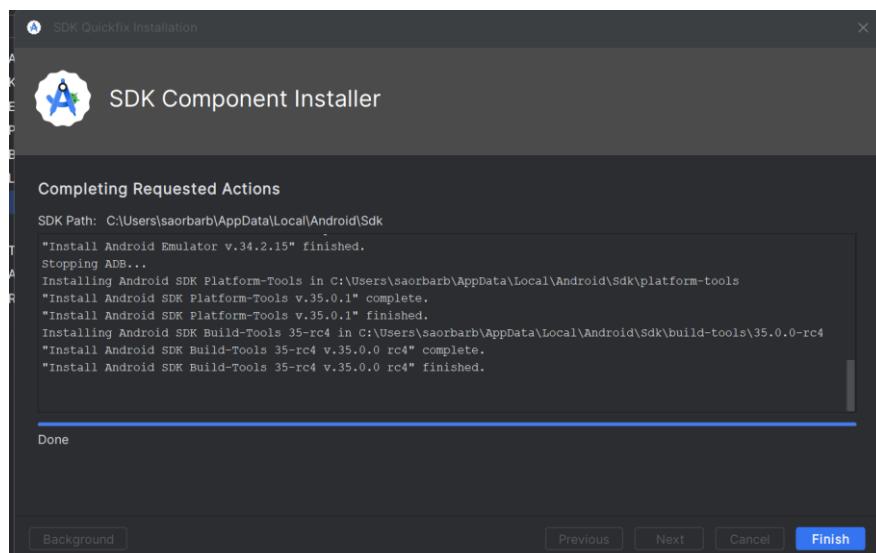


Figura 20 - SDK Tools. Fonte: autoria própria.

3.6 Emulador

Para criar o emulador no *Android Studio*, crie primeiramente um novo projeto. No menu principal, clique em **File, New, New Flutter Project**, a seguinte tela aparecerá conforme abaixo, veja o caminho apontando para a criação do projeto, deverá ser o mesmo onde o Flutter SDK foi instalado.

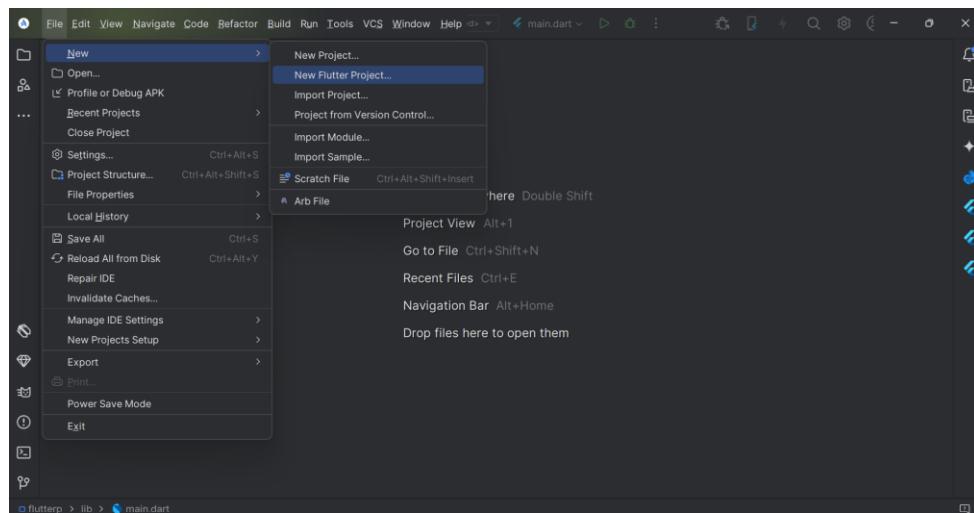


Figura 21 - Criação de Projeto no Flutter. Fonte: autoria própria.

Na sequência, dê um nome ao seu projeto e selecione o caminho onde está instalado o Flutter, conforme o exemplo abaixo:

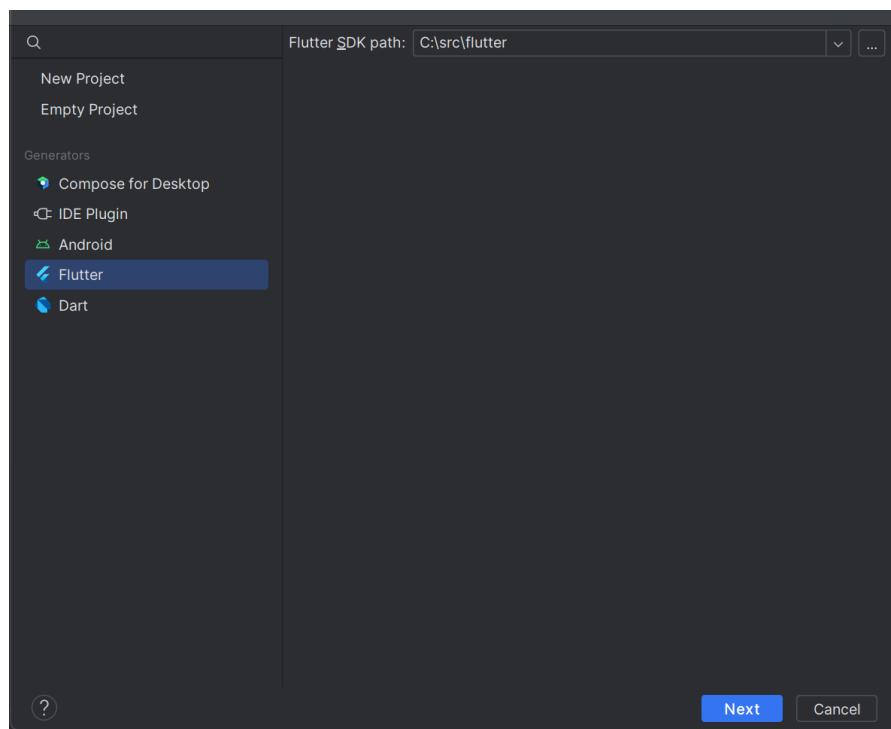


Figura 22 - Criação de Projeto no Flutter. Fonte: autoria própria.

Na sequência, dê um nome ao seu projeto e selecione o caminho onde está instalado o Flutter, conforme o exemplo abaixo:

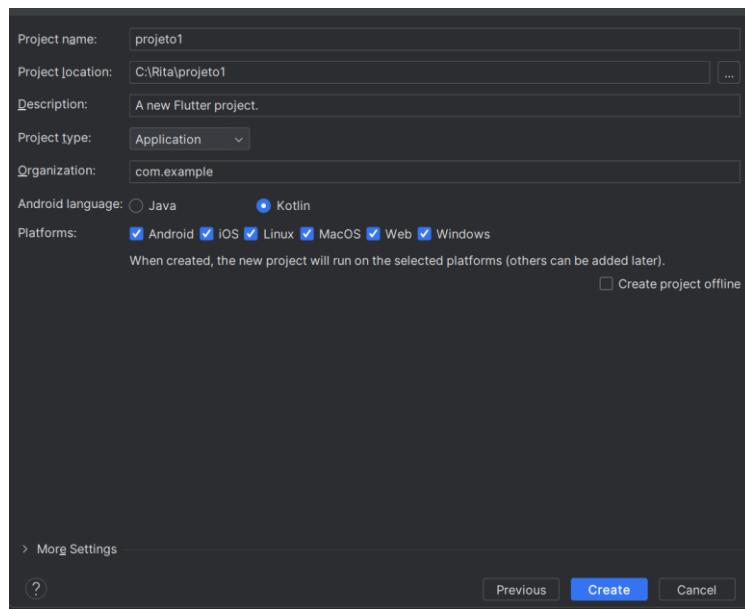


Figura 23 - Criação de Projeto no Flutter. Fonte: autoria própria.

Em seguida abra o seu projeto, clique em “The Window”, conforme abaixo:

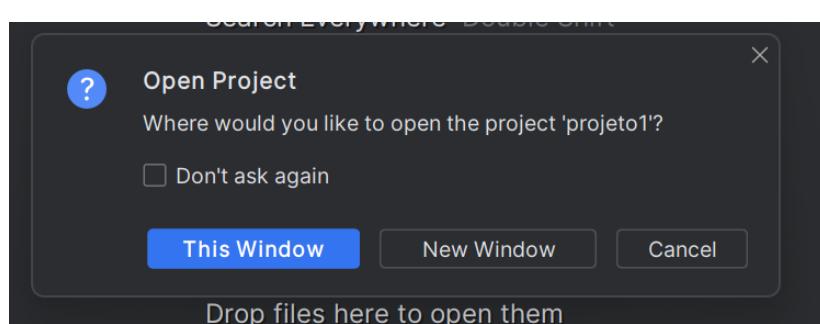


Figura 24 - Iniciando Projeto no Flutter. Fonte: autoria própria.

Clique na aba, Tools, Device Manager do menu principal do Android Studio:

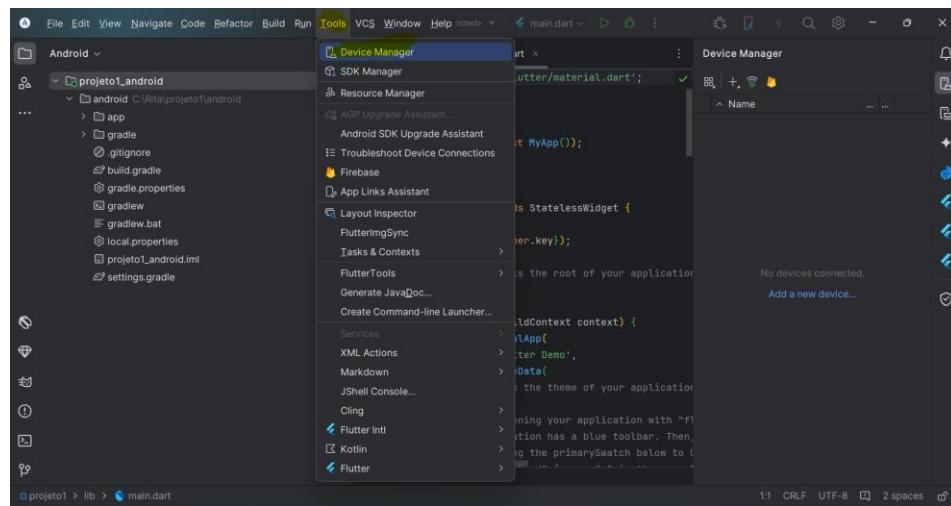


Figura 25 - Device Manager. Fonte: autoria própria.

Na sequência, em Device Manager, click no ícone “+” e selecione Create Virtual Device, conforme abaixo:

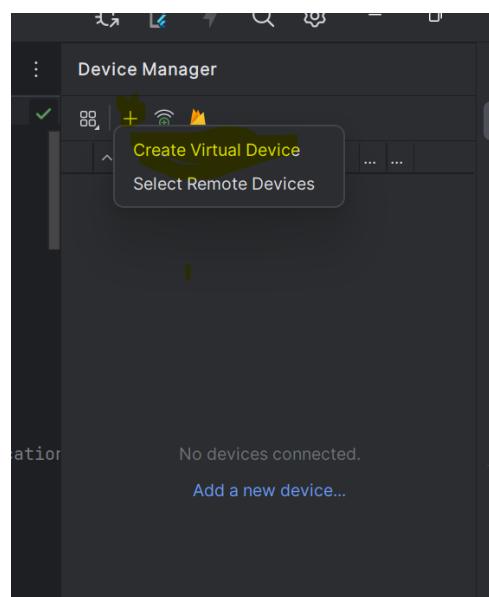


Figura 26 - Criando Device. Fonte: autoria própria.

Agora, será possível escolher o tipo de dispositivo. A lista de categorias à esquerda apresenta todos os tipos de dispositivos disponíveis para emular. Selecione Phone e na lista de dispositivos, marque o Nexus 6P e clique em “Next”.

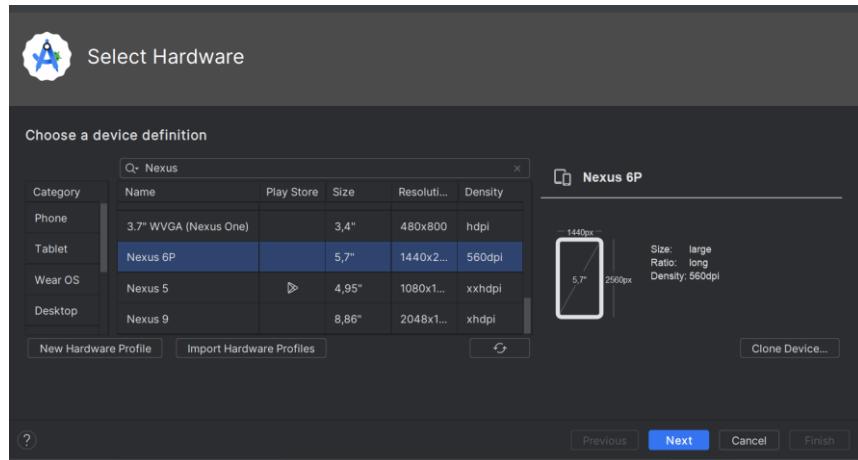


Figura 27 - Selecionando Dispositivo Nexus. Fonte: autoria própria.

O próximo passo, será a escolha da versão do Android no dispositivo virtual, que será executado. Selecione, por exemplo o R API 30 e certifique-se o valor x86 na coluna ABI. Faça o download (se necessário) e na sequência clique em “Next”.

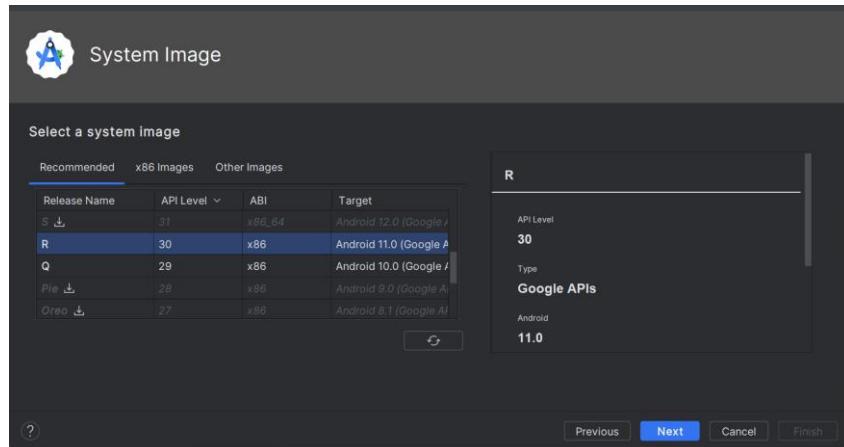


Figura 28 - Instalação R Android 11. Fonte: autoria própria.

A última tela permite confirmar suas escolhas e oferece opções para configurar algumas outras propriedades, como nome do dispositivo, orientação de inicialização. Por enquanto, use os padrões e clique em Finish:

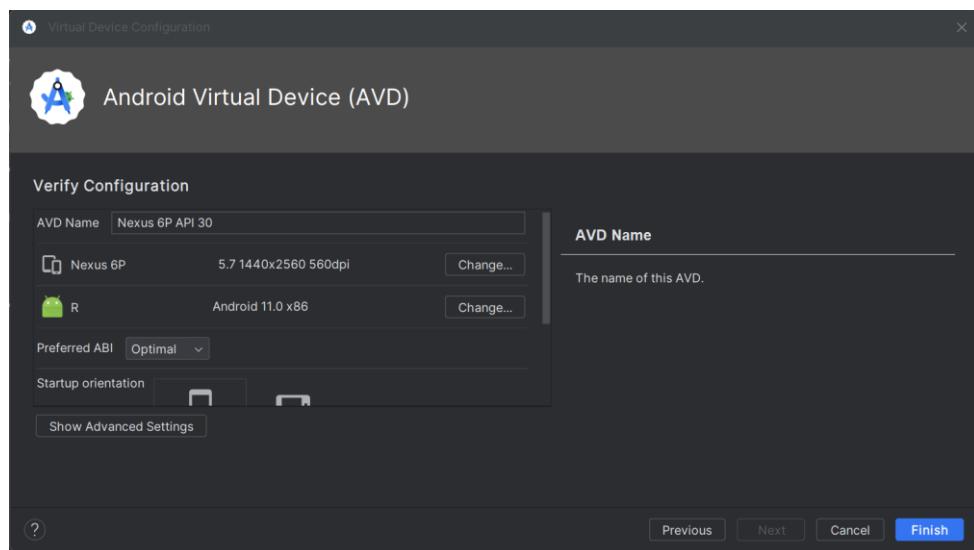


Figura 29 - Configuração final do Virtual Device Manager. Fonte: autoria própria.

O Emulador aparecerá na tela do Virtual Device Manager. Click em start para iniciar o emulador, conforme abaixo:

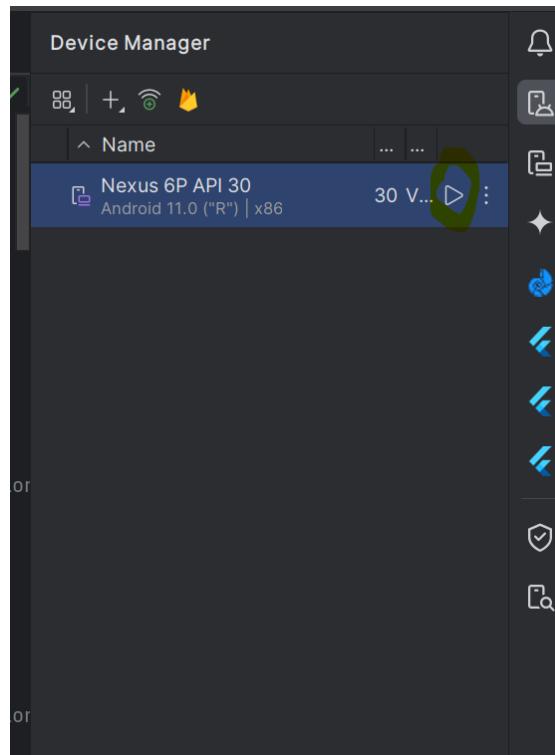


Figura 30 - Emulador configurado-Virtual Device Manager. Fonte: autoria própria.

Aguarde alguns minutos, e teremos o emulador ativo, conforme abaixo:

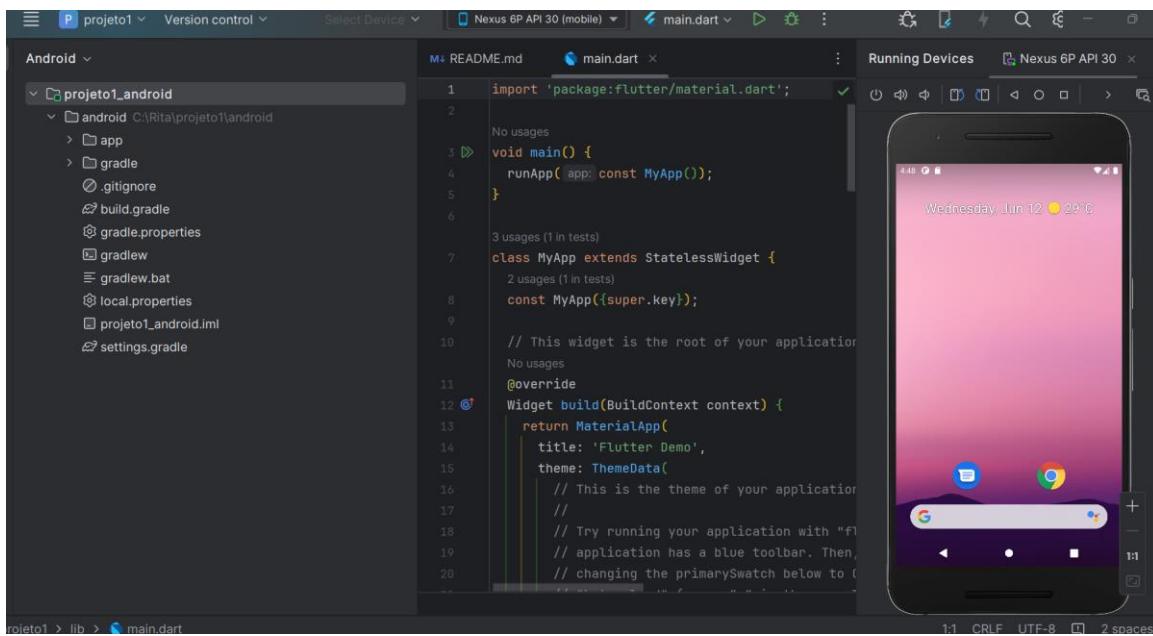


Figura 31 - Emulador em execução. Fonte: autoria própria.

As licenças do *Android* se fazem necessárias no processo para sanar alguns erros como:

License status is unknown

Android license starus unknown

Android sdkmanager tool not found

The system cannot find the path specified

A seguir, os passos e configurações necessárias:

Clique no menu principal em Tools, SDK Manager:

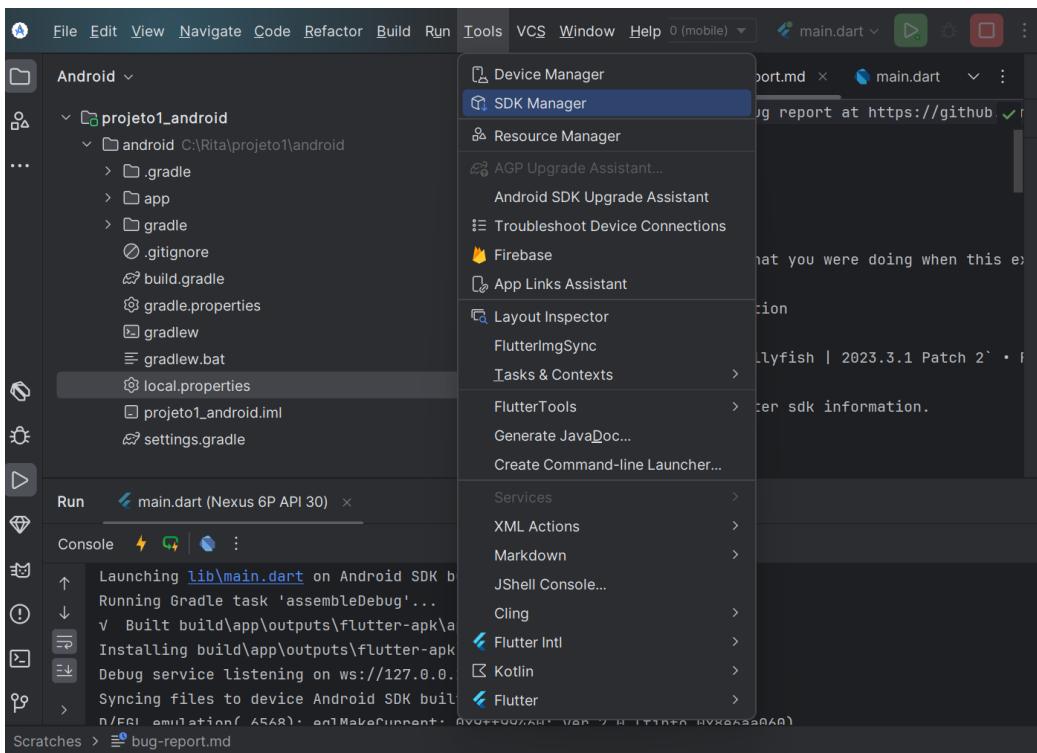


Figura 32 - SDK Manager. Fonte: autoria própria.

Em SDK Manager, acesse a guia SDK tools e desmarque a opção hide obsolete package:

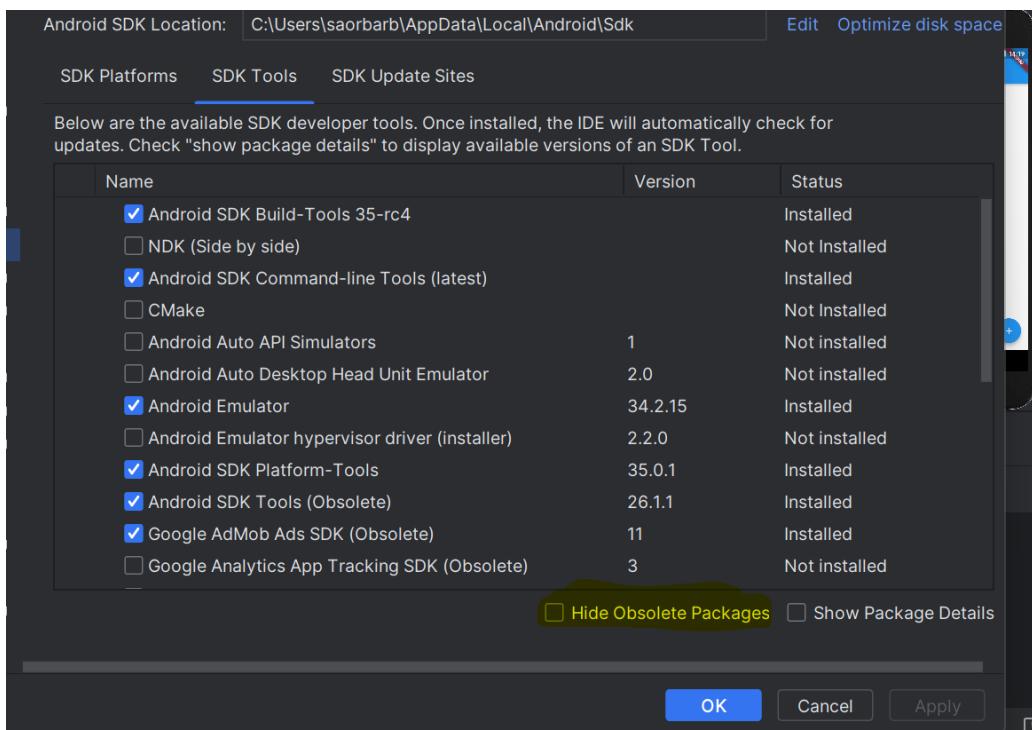


Figura 33 - SDK Tools - hide obsolete package. Fonte: autoria própria.

Escolha o pacote Android SDK tools obsolete e clicar em Apply:

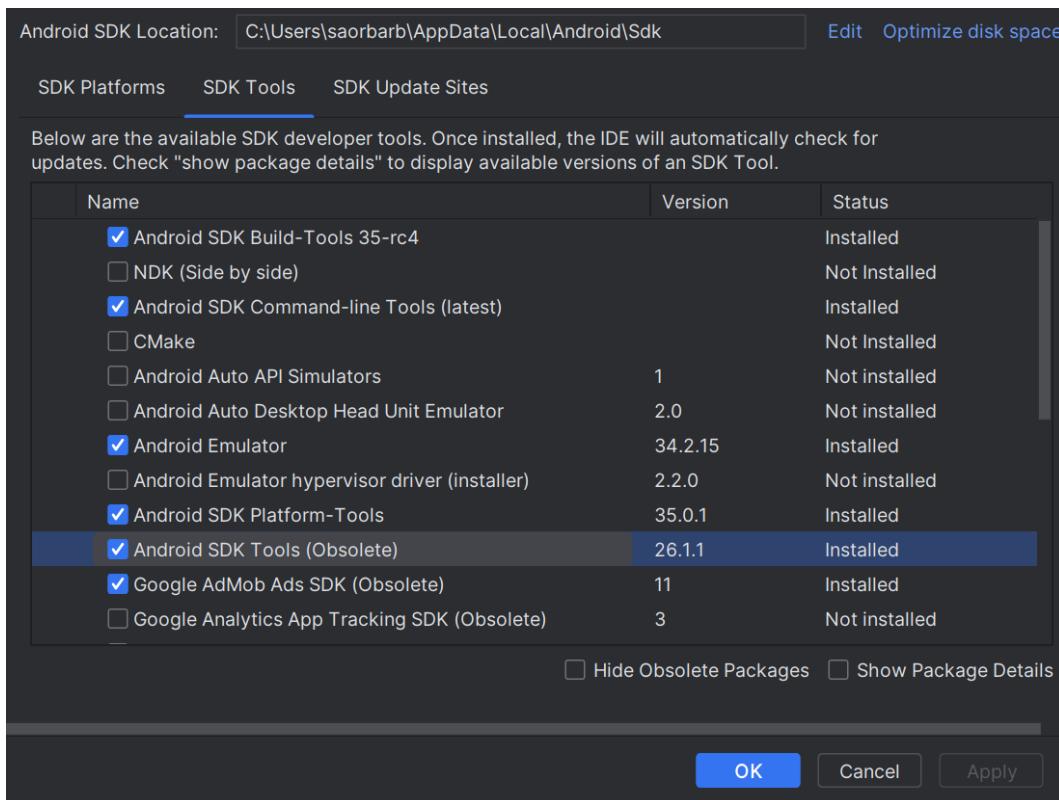


Figura 34 - SDK Tools - hide obsolete package. Fonte: autoria própria.

Confirmar o Download do pacote, clicar em Android SDK Tools com Ok:

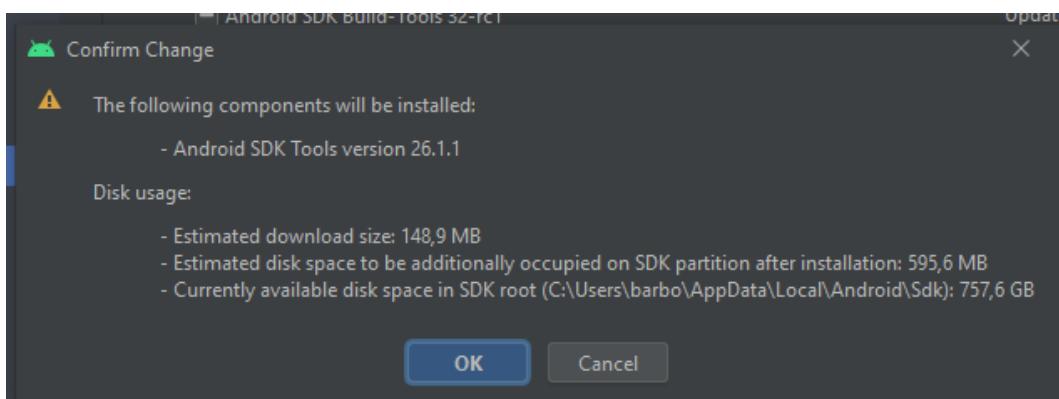


Figura 35 - SDK Tools – Download. Fonte: autoria própria.

Para finalizar a instalação do pacote, basta clicar em Finish:

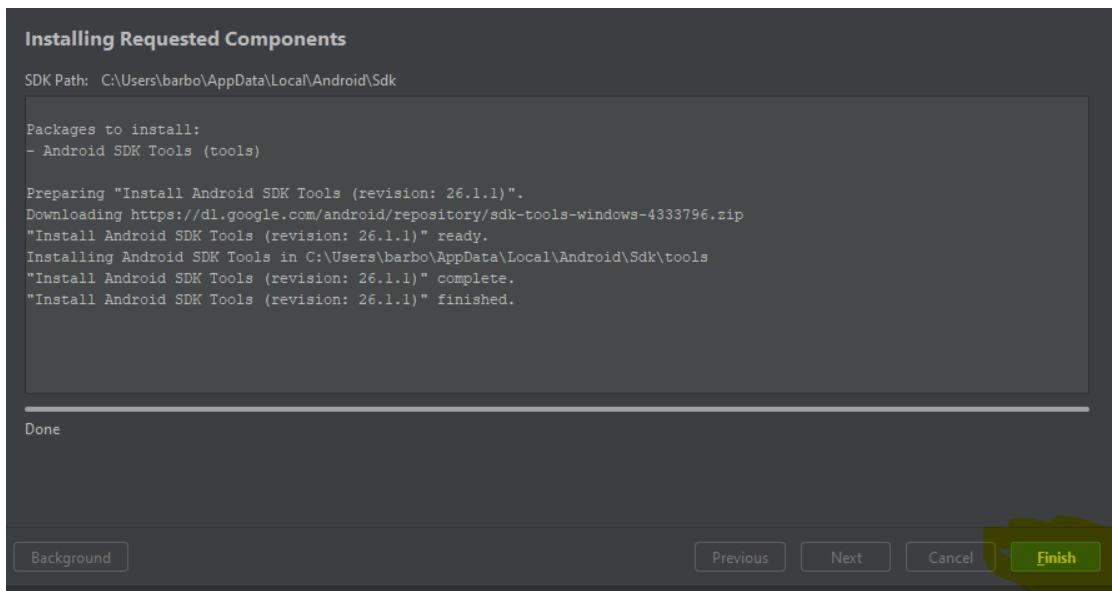


Figura 36 - Fim da instalação. Fonte: autoria própria.

3.7 Instalação e Configuração do VS Code

Agora que já temos as licenças instaladas, vamos Instalar e Configurar *Visual Studio Code*.

Esse passo é opcional, visto que se você quiser utilizar o *Android Studio* sem problemas. Porém, o *Android Studio* exige mais recursos de hardware do computador e se você possui um computador com pouca capacidade de processamento e memória, o uso do *VS Code* é uma alternativa interessante.

Faça o *download* a partir do endereço: <https://code.visualstudio.com/download>

Para nosso aprendizado, escolha o *Sistema Operacional Windows*, conforme abaixo:

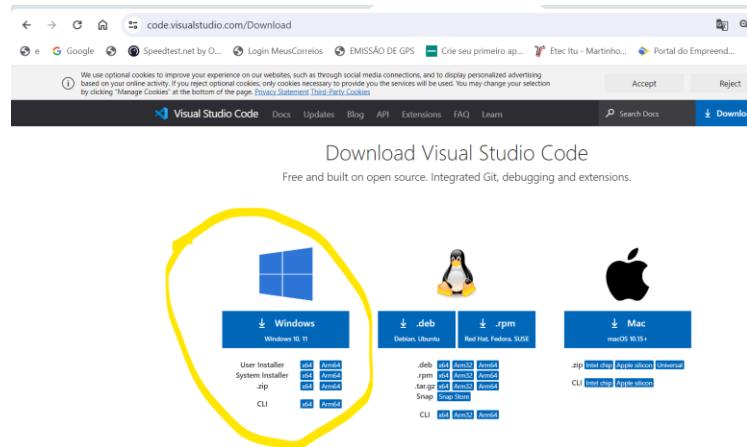


Figura 37 - Site para download VS Code. Fonte: Visual Studio.

Execute o arquivo `VSCodeUserSetup-x64-1.90.1.exe` para iniciar a instalação do Microsoft Visual Studio Code, click em “accept the agreement e click em next, conforme abaixo:

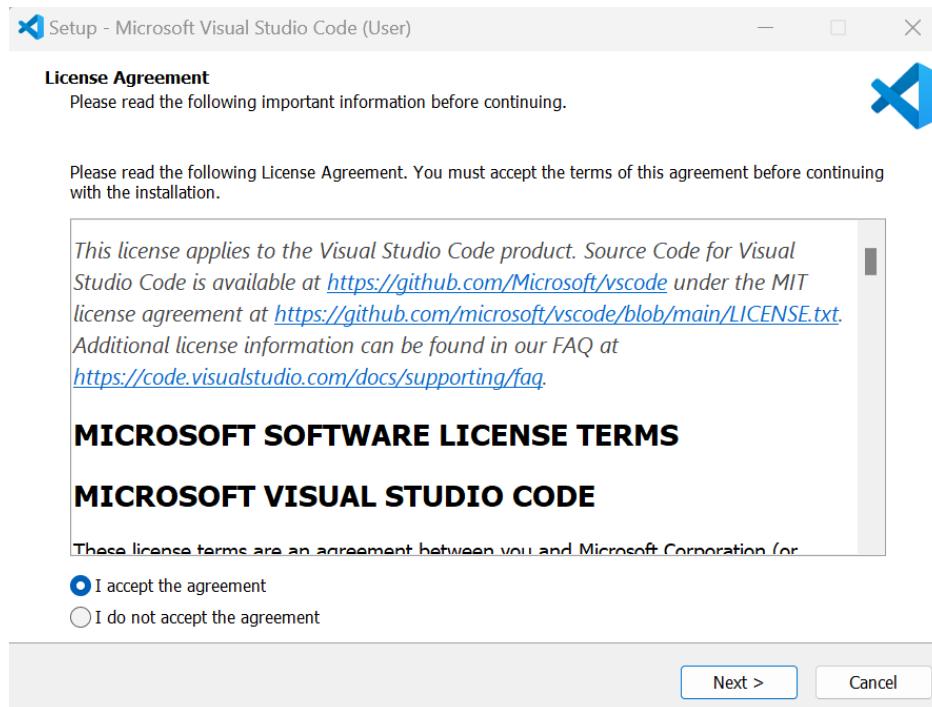


Figura 38 - Instalação do VS Code. Fonte: autoria própria

Confirme o diretório de instalação, e vá selecionando next para aceitar a instalação:

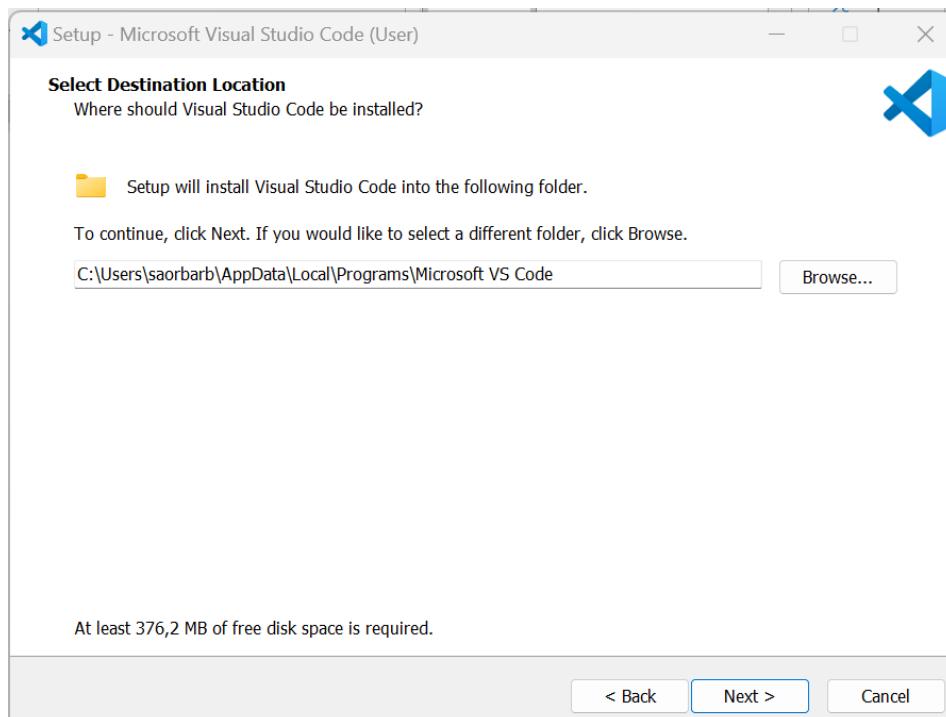


Figura 39 - Instalação do VS Code. Fonte: autoria própria.

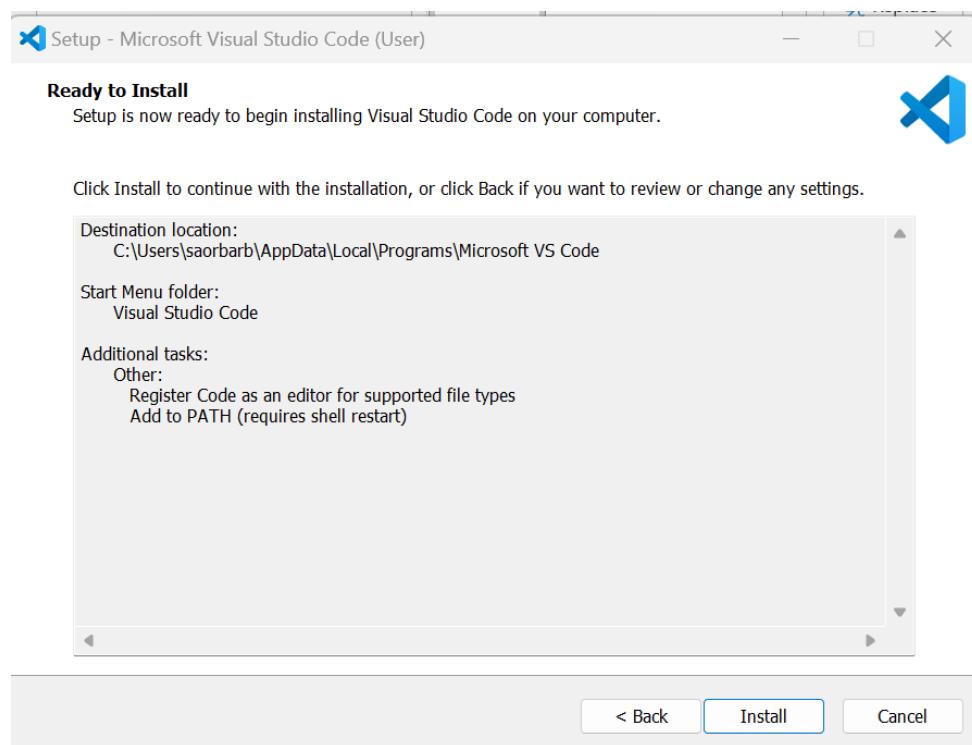


Figura 40 - Instalação do VS Code. Fonte: autoria própria.

Click em finished para terminar a instalação:

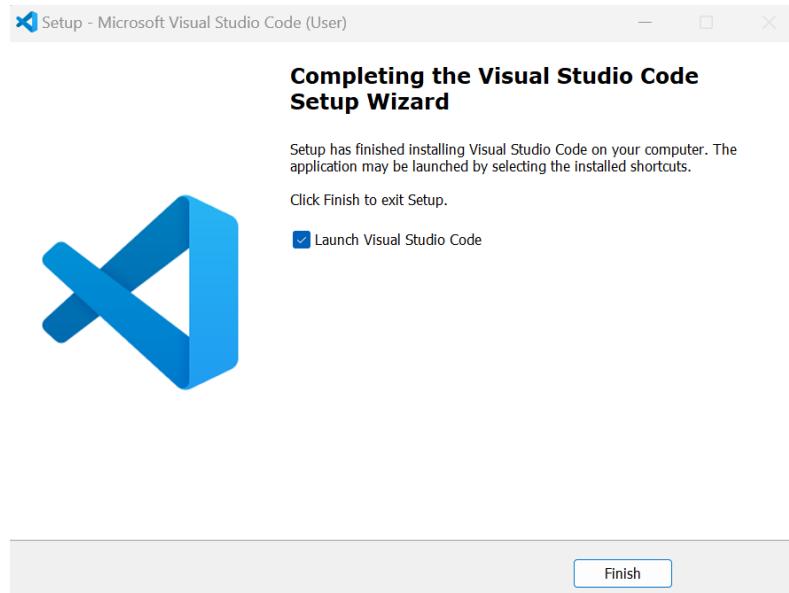


Figura 41 - Instalação do VS Code. Fonte: autoria própria.

Após a instalação, vamos instalar as extensões do Flutter e Dart, Clique em configurações, e digite as extensões, uma de cada vez:

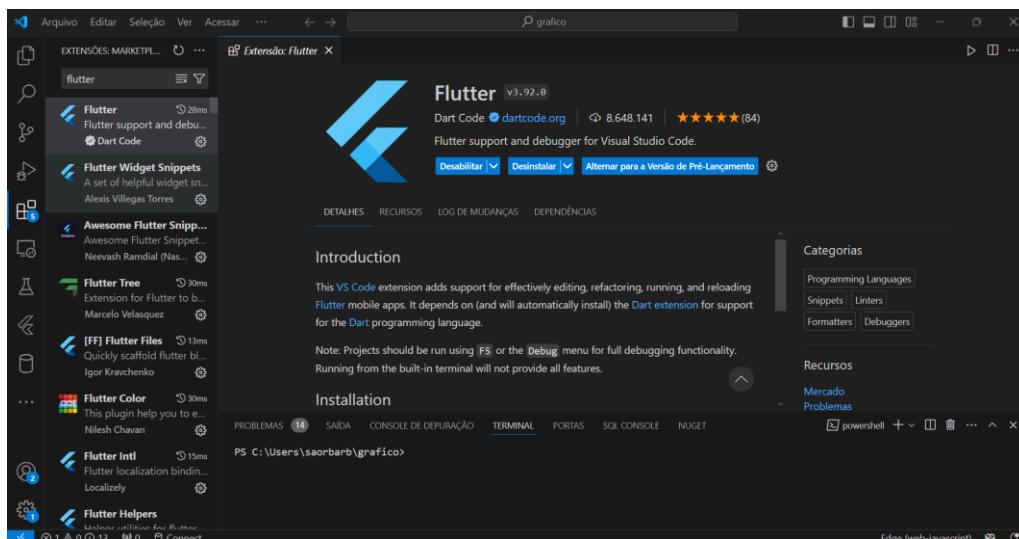


Figura 42 - Instalação de Extensões VS Code. Fonte: autoria própria.

3.8 Configuração do *Flutter*- windows

No início realizamos a instalação do Flutter, e agora, com as licenças do Android SDK instaladas, podemos realizar as devidas configurações no *Flutter*.

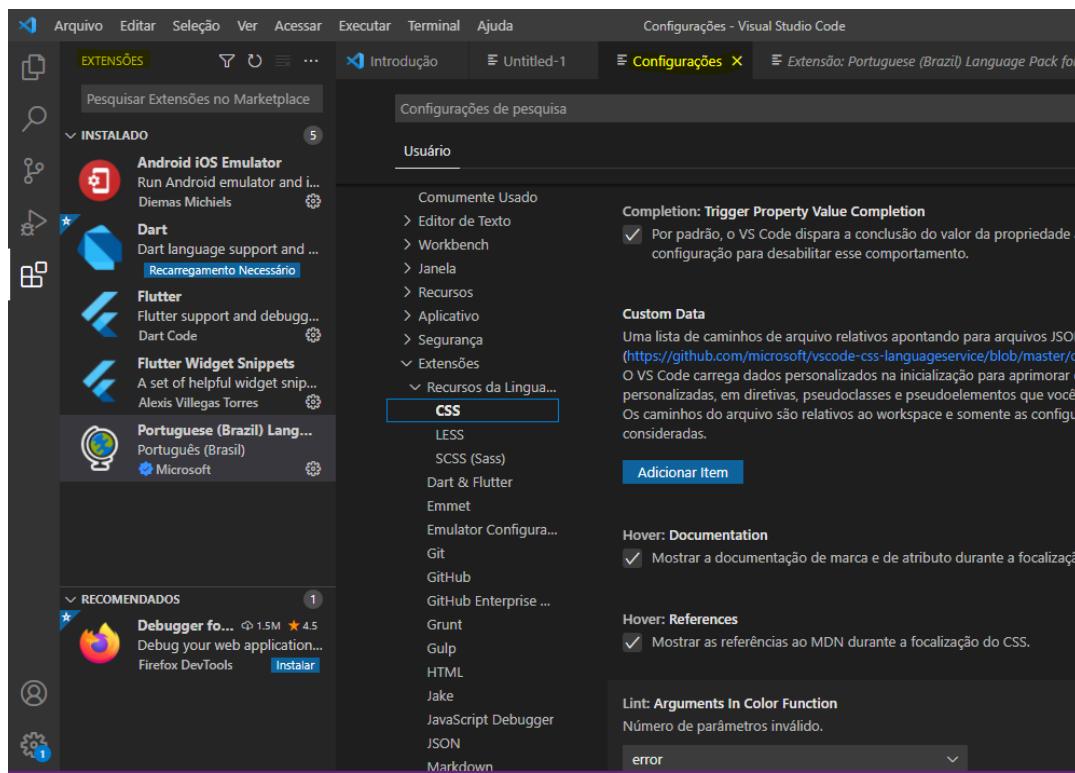


Figura 43 – Configuração do Flutter. Fonte: autoria própria.

Os procedimentos abaixo também são válidos para MacOS, onde o aluno deverá executar os comandos no terminal do MacOS.

No seu computador, vá até o diretório onde foi instalado o Flutter, e execute o arquivo flutter_console.bat.

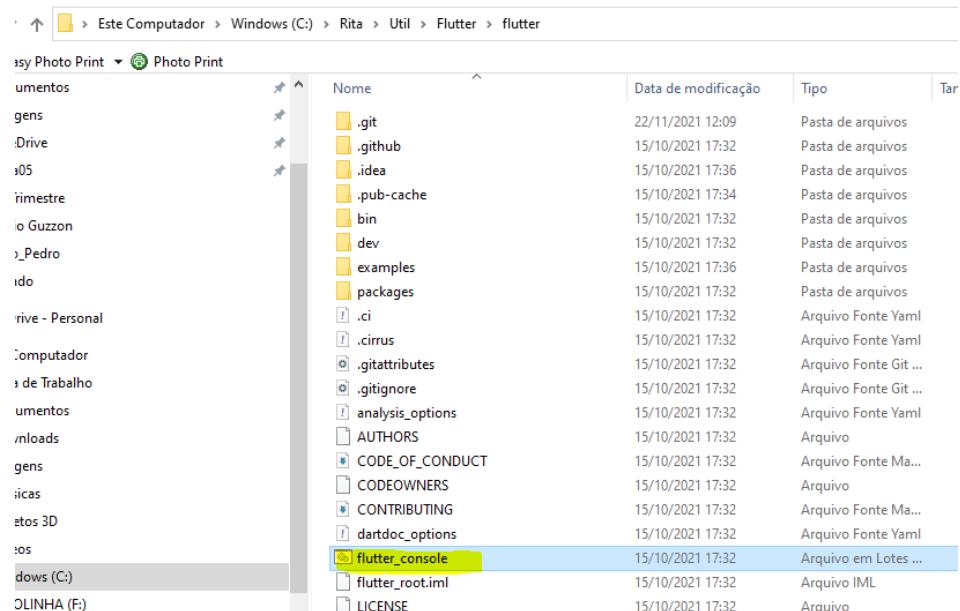


Figura 44 - Arquivo flutter_console.bat. Fonte: autoria própria.

```

#####
##      ##  ## ##### ## ##### ## #####
##      ##  ## ## ## ## ## ## ## ## ##
##      ##  ## ## ## ## ## ## ## ## ##
##### ##  ## ## ## ## ## ## ## ## ##
##      ##  ## ## ## ## ## ## ## ## ##
##      ##  ## ## ## ## ## ## ## ## ##
##      ##  ## ## ## ## ## ## ## ## ##
WELCOME to the Flutter Console.
=====
Use the console below this message to interact with the "flutter" command.
Run "flutter doctor" to check if your system is ready to run Flutter apps.
Run "flutter create <app_name>" to create a new Flutter project.

Run "flutter help" to see all available commands.

Want to use an IDE to interact with Flutter? https://flutter.dev/ide-setup/
Want to run the "flutter" command from any Command Prompt or PowerShell window?
Add Flutter to your PATH: https://flutter.dev/setup-windows/#update-your-path
=====

C:\Users\barbo>

```

Figura 45 - Arquivo flutter_console.bat executado. Fonte: autoria própria.

O próximo passo será averiguar se existem erros nas instalações do Android e demais programas, para isso, digite o comando: flutter doctor

Caso a execução do comando apresente algum erro, esse será relacionado a licença do Android, como na figura abaixo:

```

C:\Users\barbo>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[!] Flutter (Channel stable, 2.5.3, on Microsoft Windows [versão 10.0.19043.1288], locale pt-BR)
[!] Android toolchain - develop for Android devices (Android SDK version 31.0.0)
  ! Some Android licenses not accepted. To resolve this, run: flutter doctor --android-licenses
[!] Chrome - develop for the web
[!] Android Studio (version 2020.3)
[!] VS Code (version 1.62.2)
[!] Connected device (2 available)

```

Figura 46 - Erro comando Flutter Doctor. Fonte: autoria própria.

Para resolver esse problema, execute o comando: flutter doctor --android-licenses.

Após execução do comando acima (flutter doctor --android-licenses), será apresentado telas em espera para dar continuidade a instalação. Pressione Y a cada tela de espera, esta será a confirmação do aceite de configuração das instalações da licença.

Outro erro que poderá acontecer, é em relação ao Java, para quem está utilizando o Android Studio, conforme abaixo:

```
C:\Users\HP>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.3.10, on Microsoft Windows [Version 10.0.22621.1105], locale en-US)
[✓] Android toolchain - develop for Android devices (Android SDK version 33.0.1)
[✓] Chrome - develop for the web
[✓] Visual Studio - Develop for Windows (Visual Studio Community 2022 17.4.4)
[!] Android Studio (version 2022.1)
    X Unable to find bundled Java version.
[✓] IntelliJ IDEA Community Edition (version 2022.3)
[✓] VS Code (version 1.74.3)
[✓] Connected device (3 available)
```

Figura 47 - Erro Java para o Android Studio. Fonte: autoria própria.

Para resolver este problema do Java, copie os arquivos em C:\Program Files\Android\Android Studio\jbr e cole em C:\Program Files\Android\Android Studio\jre. Caso o diretório jre não exista, crie-o antes da cópia dos arquivos do diretório jbr.

Execute novamente o comando flutter doctor, e o problema estará resolvido.

```
C:\Users\barbo>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.5.3, on Microsoft Windows [versão 10.0.19043.1288], locale pt-BR)
[✓] Android toolchain - develop for Android devices (Android SDK version 31.0.0)
[✓] Chrome - develop for the web
[✓] Android Studio (version 2020.3)
[✓] VS Code (version 1.62.2)
[✓] Connected device (2 available)

• No issues found!

C:\Users\barbo>
```

Figura 48 - Erro Flutter Doctor solucionado. Fonte: autoria própria.

Antes de criarmos nosso projeto *Hello World*, vamos mostrar opcionalmente o carregamento do emulador do *Flutter* por linha de comando, dentro da console do *Flutter*, conforme abaixo:

Digite o comando **flutter emulator**:

```
C:\Users\barbo>flutter emulator
1 available emulator:

Nexus_6_API_22 • Nexus 6 API 22 • Google • android

To run an emulator, run 'flutter emulators --launch <emulator id>'.
To create a new emulator, run 'flutter emulators --create [--name xyz]'.

You can find more information on managing emulators at the links below:
  https://developer.android.com/studio/run/managing-avds
  https://developer.android.com/studio/command-line/avdmanager

C:\Users\barbo>
```

Figura 49 - Flutter mulatos. Fonte: autoria própria.

O próximo passo será digitar o comando `flutter emulator --launch <emulator id>`. Na prática, ficará `flutter emulator --launch Nexus_6_API_22`:

```
C:\Users\barbo>flutter emulators --launch Nexus_6_API_22
C:\Users\barbo>
```

Figura 50 - Flutter emulators launch. Fonte: autoria própria.



A instalação dos IDE's propostos neste apostila requerem um computador com capacidade para instalação e execução, conforme indicado no site de cada fabricante. Caso o aluno não possua equipamento para a instalação destes IDE's, temos 02 sugestões de IDE'S online, sendo: **Flutter Lab:** <https://flutlab.io/> e **Project IDX:** <https://idx.dev>. Nestes sites há documentação para criação dos projetos, mas, o instrutor em aula poderá também auxiliar o aluno na sua utilização.

3.9 Desenvolvimento online usando a ferramenta Project IDX

O Project IDX é uma plataforma inovadora que revoluciona o desenvolvimento de software ao oferecer um ambiente totalmente online. Seus principais benefícios incluem a eliminação da necessidade de instalações locais, permitindo que desenvolvedores iniciem seus projetos de qualquer lugar, utilizando apenas um navegador web. Isso não só economiza tempo e recursos. Com o Project IDX, as barreiras tecnológicas são reduzidas, proporcionando uma experiência de desenvolvimento ágil, flexível e acessível a todos os níveis de habilidade.

Para acessar digite: <https://idx.dev/>

3.10 Criando seu primeiro projeto

Com o emulador configurado, agora vamos criar no 1º Projeto Hello World. Ainda na console do Flutter, digite flutter create hello_world, e veja a criação do projeto, conforme abaixo:

```
C:\Users\barbo>flutter create hello_world
Creating project hello_world...
hello_world\lib\main.dart (created)
hello_world\pubspec.yaml (created)
hello_world\README.md (created)
hello_world\test\widget_test.dart (created)
hello_world\.gitignore (created)
hello_world\.idea\libraries\ Dart_SDK.xml (created)
hello_world\.idea\libraries\KotlinJavaRuntime.xml (created)
hello_world\.idea\modules.xml (created)
hello_world\.idea\runConfigurations\main_dart.xml (created)
hello_world\.idea\workspace.xml (created)
hello_world\.metadata (created)
hello_world\analysis_options.yaml (created)
hello_world\android\app\build.gradle (created)
hello_world\android\app\src\main\kotlin\com\example\hello_world\MainActivity.kt (created)
hello_world\android\build.gradle (created)
hello_world\android\hello_world_android.iml (created)
hello_world\android\.gitignore (created)
hello_world\android\app\src\debug\AndroidManifest.xml (created)
hello_world\android\app\src\main\AndroidManifest.xml (created)
hello_world\android\app\src\main\res\drawable\launch_background.xml (created)
hello_world\android\app\src\main\res\drawable-v21\launch_background.xml (created)
hello_world\android\app\src\main\res\mipmap-hdpi\ic_launcher.png (created)
hello_world\android\app\src\main\res\mipmap-mdpi\ic_launcher.png (created)
hello_world\android\app\src\main\res\mipmap-xhdpi\ic_launcher.png (created)
```

Figura 51 - Criação do Projeto Hello World. Fonte: autoria própria.

```
hello_world\ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-60x60@3x.png (created)
hello_world\ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-76x76@1x.png (created)
hello_world\ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-76x76@2x.png (created)
hello_world\ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-83.5x83.5@2x.png (created)
hello_world\ios\Runner\Assets.xcassets\LaunchImage.imageset\Contents.json (created)
hello_world\ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage.png (created)
hello_world\ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage@2x.png (created)
hello_world\ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage@3x.png (created)
hello_world\ios\Runner\Assets.xcassets\LaunchScreen.storyboard (created)
hello_world\ios\Runner\Base.lproj\Main.storyboard (created)
hello_world\ios\Runner\Info.plist (created)
hello_world\ios\Runner\xcodeproj\project.xcworkspace\contents.xcworkspacedata (created)
hello_world\ios\Runner\xcodeproj\project.xcworkspace\xcshareddata\IDEWorkspaceChecks.plist (created)
hello_world\ios\Runner\xcodeproj\project.xcworkspace\xcshareddata\WorkspaceSettings.xcsettings (created)
hello_world\ios\Runner\xcworkspace\contents.xcworkspacedata (created)
hello_world\ios\Runner\xcworkspace\xcshareddata\IDEWorkspaceChecks.plist (created)
hello_world\ios\Runner\xcworkspace\xcshareddata\WorkspaceSettings.xcsettings (created)
hello_world\hello_world.iml (created)
hello_world\web\favicon.png (created)
hello_world\web\icons\Icon-192.png (created)
hello_world\web\icons\Icon-512.png (created)
hello_world\web\icons\Icon-maskable-192.png (created)
hello_world\web\icons\Icon-maskable-512.png (created)
hello_world\web\index.html (created)
hello_world\web\manifest.json (created)
running "flutter pub get" in hello_world... 1.813ms
rote 81 files.

All done!
In order to run your application, type:

$ cd hello_world
$ flutter run

our application code is in hello_world\lib\main.dart.
```

Figura 52 - Criação do Projeto Hello World – finalização. Fonte: autoria própria.

Projeto criado!! Agora iremos alterar e executar o código, no arquivo main.dart para finalmente executarmos nosso programa Hello World, apague todo o código existente e digite o seguinte código:

```
import 'package:flutter/widgets.dart';
void main() {
  runApp(
    Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  )
}
```

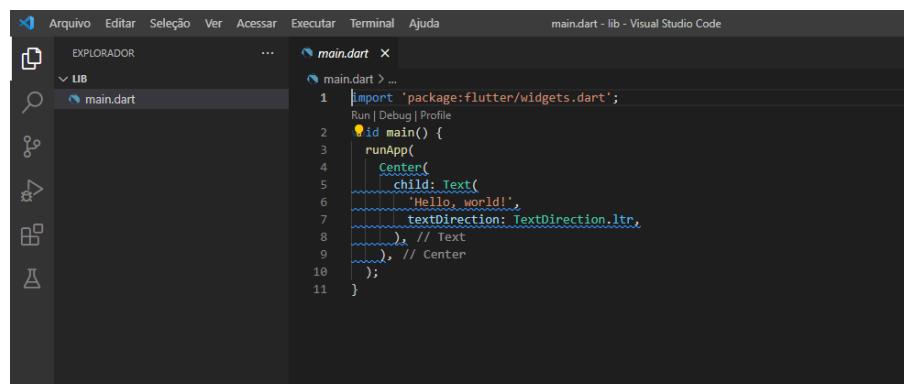


Figura 53 – Arquivo main.dart. Fonte: autoria própria.

Agora, vamos testá-lo, para isto, siga os seguintes passos: na console do Flutter, vá para o diretório onde foi criado o projeto Hello World. No nosso caso, cd\hello_world. Digite o comando flutter clean (limpa diretório). Na sequência, digite flutter pub get(atribui todas as dependências e libraries do flutter). E finalmente, vamos chamar o VS Code. digite o comando code ., a tela inicial do VS Code aparecerá da seguinte forma:

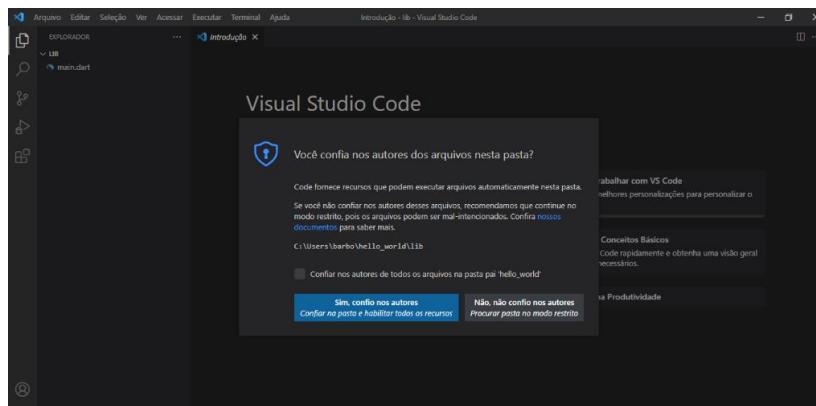


Figura 54 - Executando VS Code via console do Flutter. Fonte: autoria própria.

Clique para aceitar e confiar nos autores no VS Code, em seguida, Para editar o aplicativo, clique em: lib > main.dart.

Para iniciar o código, aperte as teclas **CRTL + F5**. No topo superior do Visual Studio Code abrirá uma pequena aba. Se você já possui emuladores configurados, eles devem aparecer nessa aba. Se não, basta selecionar a opção “Create New”:

```

File Edit Selection View Go Debug Terminal Help
main.dart - meu_primeiro_app - Visual Studio Code [Administrator]
EXPLORER
MEU PRIMEIRO APP
  Idea
  android
  ios
  lib
    main.dart
  test
    .gitignore
    .metadata
    .packages
    meu_primeiro_app.iml
    pubspec.lock
    pubspec.yaml
  README.md
  DEPENDENCIES
  1 void main() => runApp(MyApp());
  2
  3 class MyApp extends StatelessWidget {
  4   // This widget is the root of your application.
  5   @override
  6   Widget build(BuildContext context) {
  7     return MaterialApp(
  8       title: 'Flutter Demo',
  9       theme: ThemeData(
 10         // This is the theme of your application.
 11         // Try running your application with "flutter run". You'll see
 12         // application has a blue toolbar. Then, without quitting the
 13         // application has a blue toolbar. Then, without quitting the
 14         // application has a blue toolbar. Then, without quitting the
 15         // application has a blue toolbar. Then, without quitting the
 16         // application has a blue toolbar. Then, without quitting the
 17         // application has a blue toolbar. Then, without quitting the
 18         // application has a blue toolbar. Then, without quitting the
 19         // application has a blue toolbar. Then, without quitting the
 20         // application has a blue toolbar. Then, without quitting the
 21         primarySwatch: Colors.blue,
 22       ), // ThemeData
 23       home: MyHomePage(title: 'Flutter Demo Home Page'),
 24     ); // MaterialApp
 25   }
 26 }
 27
 28 class MyHomePage extends StatefulWidget {
 29   MyHomePage({Key key, this.title}) : super(key: key);
 30
 31   // This widget is the home page of your application. It is stateful,

```

Figura 55 - Execução Arquivo main.dart. Fonte: autoria própria.

O processo deve demorar alguns minutos, e você pode acompanhá-lo através da pequena aba aberta no canto inferior direito:

```

1 import 'package:flutter/material.dart';
2
3 void main() => runApp(MyApp());
4
5 class MyApp extends StatelessWidget {
6   // This widget is the root of your application.
7   @override
8   Widget build(BuildContext context) {
9     return MaterialApp(
10       title: 'Flutter Demo',
11       theme: ThemeData(
12         // This is the theme of your application.
13         //
14         // Try running your application with "flutter run". You'll see
15         // application has a blue toolbar. Then, without quitting the .
16         // changing the primarySwatch below to Colors.green and then is
17         // "hot reload" (press "r" in the console where you ran "flutte
18         // or simply save your changes to "hot reload" in a Flutter IDE)
19         // Notice that the counter didn't reset back to zero; the appl
20         // is not restarted.
21         primarySwatch: Colors.blue,
22       ), // ThemeData
23       home: MyHomePage(title: 'Flutter Demo Home Page'),
24     ); // MaterialApp
25   }
26 }
27
28 class MyHomePage extends StatefulWidget {
29   MyHomePage({Key key, this.title}) : super(key: key);
30
31   // This widget is the home page of

```

Creating emulator...

Figura 56 - Execução emulador. Fonte: autoria própria.

Ao final do processo, o emulador abrirá automaticamente. Caso seu aplicativo não seja executado, execute novamente o comando **CRTL + F5**. Entretanto, como o emulador já estará aberto, não será necessário escolher nenhuma opção. O Flutter ficará responsável por iniciar a aplicação no emulador e emitir todos os eventos e erros através da aba “DEBUG CONSOLE” no rodapé do Visual Studio Code:

```

D/EGL_emulation( 6510): eguiMakeCurrent: 0xe8c817ae0: ver 2 0 (tinfo 0x552c2ea0)
I/Choreographer( 6510): Skipped 70 frames! The application may be doing too much work on its main thread.
D/EGL_emulation( 6510): eguiMakeCurrent: 0xe73055a0: ver 2 0 (tinfo 0x73037780)
I/OpenGLES( 6510): Davy! duration:136ms; Flags=1, IntendedSync=43960937799, Vsync=440767604419, OldestInputEvent=9223372036854775807, NewestInputEvent=0, HandleInputStart=440783562948, AnimationStart=440783631358, PendingFrameCount=141, DrawStart=440790577350, SyncQueued=440790676210, SyncStart=4408309652710, IssueDrawCommandsStart=440831113400, SwapBuffers=440837462560, FrameCompleted=440968819550, DequeueBufferDuration=38834000, QueueBufferDuration=1222000,
D/egl( 6510): HostConnection::get(): New Host Connection established 0xe731d000, tid 6609
D/EGL_emulation( 6510): eguiMakeCurrent: 0xe8c817ae0: ver 2 0 (tinfo 0x73036a0)

```

Figura 57 - Debug Console do Flutter. Fonte: autoria própria.

Nesse momento, no emulador seu aplicativo já será executado:

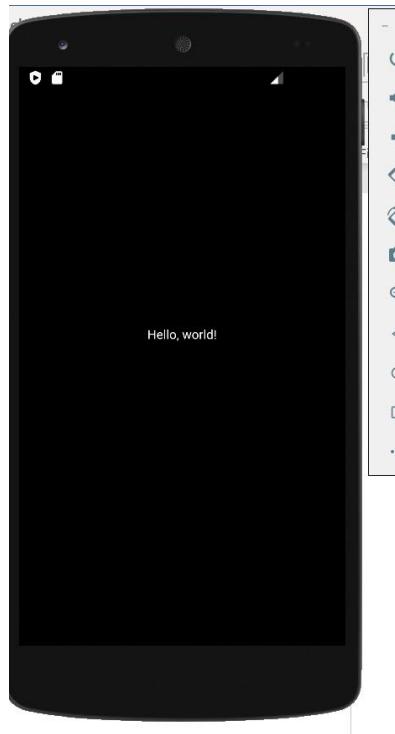


Figura 58 - Aplicativo criado automaticamente. Fonte: autoria própria.

Abra o **Android Studio**, no menu principal, click em **File, Open Project** e localize o arquivo **hello_world** e click em **ok**. O Android Studio abrirá o arquivo **main.dart**.

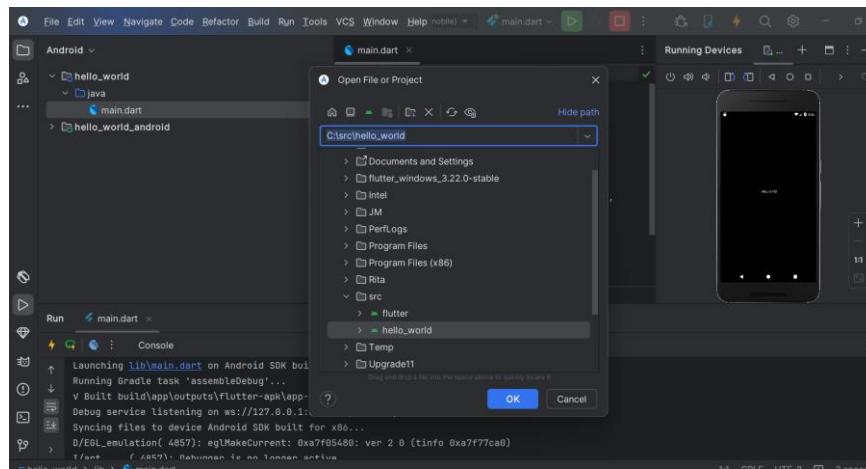


Figura 59 - Android Studio, abrindo projeto Hello World. Fonte: autoria própria.

Nosso próximo passo será ativar o emulador dentro do Android Studio da seguinte maneira: no canto direito da janela principal, clique no ícone no formato de um celular, na sequência, inicie o emulador conforme abaixo:

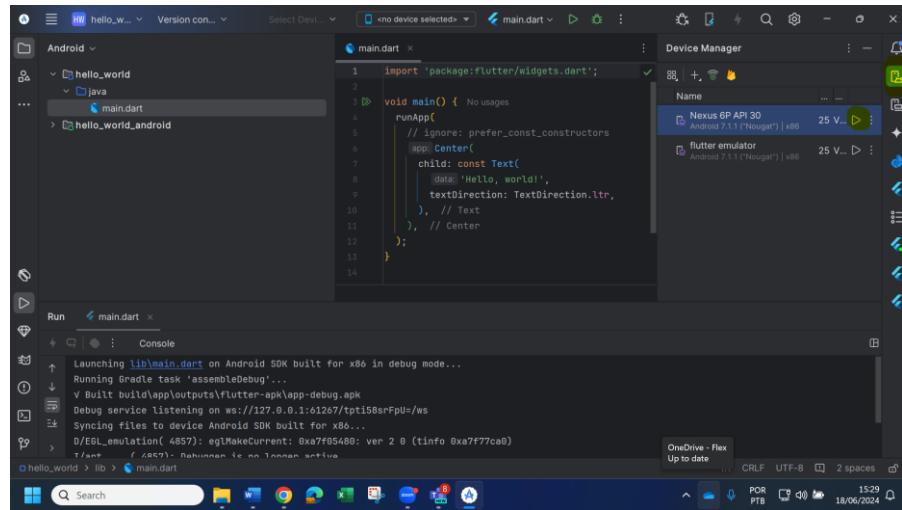


Figura 60 - Android Studio, iniciando emulador. Fonte: autoria própria.

4 *Flutter* domindo o mundo Energético: *Widgets* e *UI* com Aprendizagem Baseada em Projeto

Em uma abordagem prática baseada em projetos, mostraremos como utilizar os recursos do *Flutter* para construir interfaces intuitivas e eficientes. Esta metodologia prática permitirá que os desenvolvedores adquiram habilidades valiosas e aplicáveis, capacitando-os a criar soluções eficazes para os desafios do mercado atual. Prepare-se para uma exploração detalhada que combina teoria e prática, demonstrando o potencial do *Flutter* para moldar o futuro das interfaces de usuário no setor energético.

Widgets são atalhos que facilitam o acesso a aplicativos e ferramentas no celular. Todo aplicativo *Flutter* é um *Widget* formado de outras centenas de *Widgets*. A ideia central é que você construa sua interface com *Widgets*. Os *Widgets* do *Flutter*, foram inspirados no *React*.

Para melhor ilustrar o conceito de *Widget*, você poderá assimilar ao “LEGO”, onde cada pequeno *Widget* representa uma peça e ao final, várias peças compõem um brinquedo.

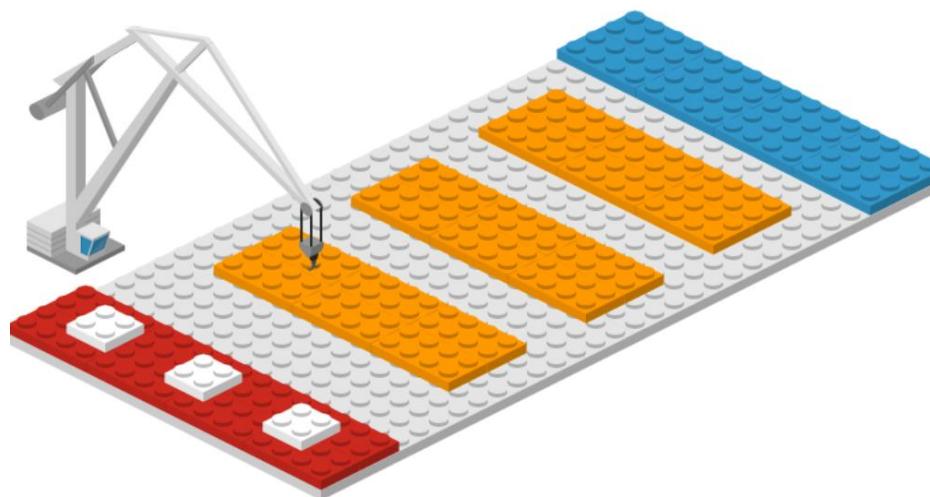


Figura 61 - Widgets. Fonte: Flutter para iniciantes.

Existe apenas 02 tipos de Widgets: **Stateless** e **Stateful**:

Stateless - *Widget* sem o controle de estado. Este tipo de *Widget* não possibilita alterações dinâmicas, entenda-o como algo completamente estático. São utilizados para a criação de estruturas não mutáveis nos aplicativos (telas, menus, imagens etc.), ou seja, tudo que não envolva entradas de dados dos usuários, acessos a APIs e coisas que mudem ao longo do processo.

Stateful – Esse *Widgets* são praticamente o oposto dos Stateless. Possuem estado e por conta disso, se tornam mutáveis. São elementos-chave para o desenvolvimento móvel da forma interativa.

Ilustraremos abaixo esses 02 tipos para melhor entendimento:

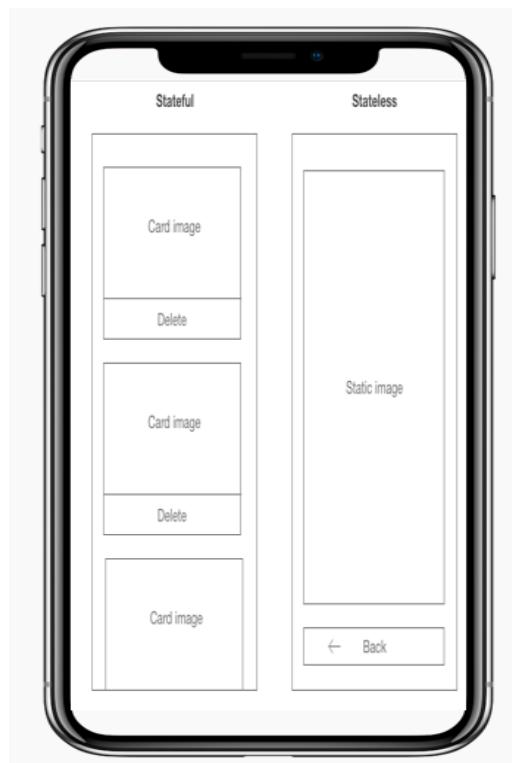


Figura 62 - Tipos Widgets - Stateless e Stateful. Fonte: Medium.

No lado esquerdo, há uma tela baseada em um widget stateful, ou seja, a tela será atualizada quando um cartão for excluído. Essa ação afetará seu estado, fazendo com que exista uma nova renderização. No lado direito, uma tela apenas exibe uma imagem e que permite uma mudança de rota, para que o usuário retorne à tela anterior, sem alteração de estado alguma.

O núcleo do mecanismo de layout do *Flutter* são os *Widgets*. Como já mencionamos, no *Flutter* quase tudo é um *Widget* até mesmo modelos de layout são *widgets.complexos*.

Os *Widgets* são organizados em uma árvore de *Widgets* numa hierarquia: pai e filho. Toda a árvore de widgets é o que forma o layout visualizado na tela do celular. Abaixo temos um diagrama da árvore de widgets para esta interface do usuário:

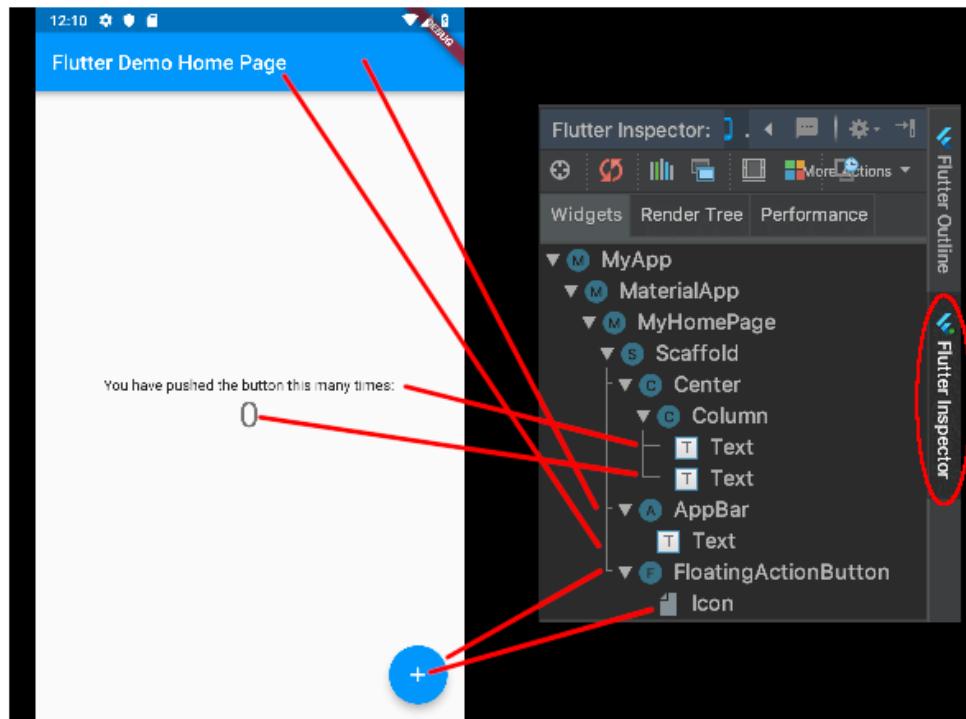


Figura 63 - Diagrama da Árvore Widgets. Fonte: Macoratti.

Abaixo, listaremos os principais *Widgets*:

Image:

Exibe imagens, sendo que os formatos suportados são JPEG, PNG, GIF, GIF animado, WebP, WebP Animado, BMP e WBMP, e, podemos exibir imagens locais e imagens remotas, obtidas a partir de uma url.

O *Flutter* usa o `pubspec.yaml`, arquivo localizado na raiz do seu projeto, para identificar os ativos exigidos por um aplicativo.

```

flutter:
  assets:
    - assets/my_icon.png
    - assets/background.png

```

Figura 64 - Image. Fonte: Flutter Dev.

Para carregar uma imagem, use a `AssetImage` classe no método de um widget `build()`.

Por exemplo, seu aplicativo pode carregar a imagem de plano de fundo das declarações de ativos do exemplo anterior:

```

return const Image(image: AssetImage('assets/background.png'));

```

Figura 65 - Image. Fonte: Flutter Dev.

ListView

O Widget `ListView` é uma lista rolável de `Widgets` organizados linearmente, sendo um dos widgets de rolagem mais usados.

```

1 | ListView(
2 |   children: <Widget>[
3 |     ListTile(
4 |       leading: Icon(Icons.map),
5 |       title: Text('Mapa'),
6 |     ),
7 |     ListTile(
8 |       leading: Icon(Icons.photo_album),
9 |       title: Text('Album'),
10 |      ),
11 |     ListTile(
12 |       leading: Icon(Icons.phone),
13 |       title: Text('Fone'),
14 |     ),
15 |   ],
16 | );

```

Figura 66 - ListView. Fonte: Site Imasters.

ListView.builder

O **ListView.builder** permite criar listas longas para rolagem. Em vez de construir todos os itens da lista de uma vez, ele constrói apenas os itens que estão visíveis na tela, economizando memória e melhorando o desempenho do aplicativo.

Sua funcionalidade consiste em utilizar uma função chamada **itemBuilder** que realiza a chamada para cada item visível na lista e, também o **itemCount** que permite definir a quantidade de itens da lista.

Abaixo um exemplo do ListView.builder:

```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Exemplo ListView.builder')),
        body: MyListView(),
      ),
    );
  }
}

class MyListView extends StatelessWidget {
  // Lista de itens
  final List<String> items = List<String>.generate(100, (i) => "Item $i");

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: items.length, // Número total de itens
      itemBuilder: (context, index) {
        // Função que constrói cada item
        return ListTile(
          title: Text(items[index]),
        );
      },
    );
  }
}
```

Figura 67 – ListView.Builder Fonte: Site Imasters.

Criando um array (matriz) com List

Para criar uma matriz (ou array bidimensional) no Flutter, utiliza-se o conceito de lista (list). Abaixo, um exemplo de app que apresenta a criação de uma matriz/array (List), e exibe seus elementos em um widget Column::

```
dart
Copiar código

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Matriz Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Matriz Flutter'),
        ),
        body: Center(
          child: MatrizWidget(),
        ),
      ),
    );
  }
}

class MatrizWidget extends StatelessWidget {
  // Criação de uma matriz 3x3
  final List<List<int>> matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
  ];
}
```

```

@Override
Widget build(BuildContext context) {
    return Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: matriz.map((row) {
            return Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: row.map((element) {
                    return Padding(
                        padding: const EdgeInsets.all(8.0),
                        child: Text(element.toString(), style: TextStyle(fontSize: 20)),
                    );
                }).toList(),
            );
        }).toList(),
    );
}

```



Figura 68 – Criação Array(List). Fonte: autoria própria

Button

Os *widgets* de botão no *Flutter* são flexíveis e podem ser facilmente personalizados para atender às necessidades de design e interação do seu aplicativo. Dependendo da importância da ação e do design desejado, você pode escolher entre os diferentes tipos de botões disponíveis e personalizá-los conforme necessário.

ElevatedButton

O *ElevatedButton* (anteriormente conhecido como *RaisedButton*) é um botão que eleva quando pressionado, dando uma sensação de profundidade. Ele é comumente usado para ações primárias no aplicativo.

```

dart
Copiar código

ElevatedButton(
    onPressed: () {
        // Ação ao pressionar o botão
    },
    child: Text('Elevated Button'),
);

```

Figura 69 – ElevateButton. Fonte: Própria

TextButton

O TextButton (anteriormente conhecido como FlatButton) é um botão sem elevação, geralmente usado para ações menos proeminentes.

```
dart

TextButton(
  onPressed: () {
    // Ação ao pressionar o botão
  },
  child: Text('Text Button'),
);
```

Figura 70 – TextButton. Fonte: autoria própria.

OutlinedButton

O OutlinedButton é um botão com uma borda contornando-o. Ele é útil para ações que precisam de destaque, mas não tanto quanto um ElevatedButton.

```
dart

OutlinedButton(
  onPressed: () {
    // Ação ao pressionar o botão
  },
  child: Text('Outlined Button'),
);
```



Figura 71. – OutlineButton. Fonte: autoria própria.

IconButton

O IconButton é um botão que exibe um ícone em vez de texto. É comum em barras de ferramentas e como botões de ação em listas.

```
dart
IconButton(
  icon: Icon(Icons.thumb_up),
  onPressed: () {
    // Ação ao pressionar o botão
  },
);
```

Figura 72 – IconButton. Fonte: autoria própria.

FloatingActionButton

O FloatingActionButton é um botão circular flutuante geralmente usado para ações principais em um aplicativo, como adicionar um novo item.

```
dart
FloatingActionButton(
  onPressed: () {
    // Ação ao pressionar o botão
  },
  child: Icon(Icons.add),
);
```

Figura 73 – FloatingActionButton. Fonte: autoria própria.

TextField

O widget TextField exibe uma sequência de texto com estilo único. A sequência pode quebrar em várias linhas ou ser exibida na mesma linha, dependendo das restrições de layout.

O argumento de estilo é opcional. Quando omitido, o texto usará o estilo do DefaultTextStyle delimitador mais próximo. Se a propriedade TextStyle.inherit do estilo fornecido for verdadeira (o padrão), o estilo fornecido será mesclado com o DefaultTextStyle delimitador mais próximo. Esse comportamento de mesclagem é útil, por exemplo, para deixar o texto em negrito ao usar a família e o tamanho de fonte padrão.

O exemplo abaixo, mostra como exibir texto usando o widget Text com o overflow definido como TextOverflow.ellipsis



Figura 74 - Widget Text (Text). Fonte: Site Flutter Dev.

Há outros argumentos de classes que poderão ser utilizadas, dependendo da necessidade a ser aplicada.

Row e Column:

Row

Widget que alinha componentes horizontalmente:

```
const Row(
  children: <Widget>[
    FlutterLogo(),
    Expanded(
      child: Text("Flutter's hot reload helps you quickly and easily experiment, build U
    ),
    Icon(Icons.sentiment_very_satisfied),
  ],
)
```

Figura 75 - Widget Linha (Row). Fonte: Site Flutter Dev.

Column

Widget que alinha componentes verticalmente:

```
const Column(
  children: <Widget>[
    Text('Deliver features faster'),
    Text('Craft beautiful UIs'),
    Expanded(
      child: FittedBox(
        child: FlutterLogo(),
      ),
    ),
  ],
)
```

Figura 76 - Widget Coluna (Column). Fonte: Site Flutter Dev.



Você pode ver mais detalhes sobre a documentação oficial de Widgets em: <https://docs.flutter.dev/ui/widgets>

4.1 Histórico Dart

Dart (originalmente denominada Dash) é uma linguagem de programação voltada à web, desenvolvida pela Google. O objetivo da linguagem Dart inicialmente foi substituir o JavaScript como linguagem principal embutida nos navegadores.

Em novembro de 2013 foi lançada a primeira versão estável Dart 1.0. e, em agosto de 2018 foi lançado o Dart 2.0, um reboot da linguagem, otimizado para o desenvolvimento client-side (executada no cliente) para Web e dispositivos móveis.

Abaixo algumas características da linguagem:

A sintaxe é C-like, para quem tem experiência com C#, PHP ou Javascript.

Segue o paradigma orientado a objetos.

Todos os objetos herdam da classe Object.

Fortemente tipada, significa que uma vez que a variável foi declarada com um tipo, ela será até o fim do mesmo tipo. Também normalmente possuem declaração explícita de tipo, onde o tipo da variável deve ser especificado logo na sua declaração.

Palavras reservadas são representadas por um underline (_) no início do nome de um atributo, método ou classe para torná-lo privado.

Dart pode ser compilada em *ahead-of-time* (AOT - processo de compilação que ocorre antes da execução do aplicativo e não durante) e *just-in-time* (JIT - é a compilação de um programa em tempo de execução).

Variáveis

Variável é um local para armazenar temporariamente alguma informação. Os principais tipos de variáveis são:

```
void main() {
    // Variável que armazena números inteiros
    int idade = 33;
    print("Idade: $idade");

    // Variável que armazena números decimais
    double raio = 10.25;
    print("Raio: $raio");

    // Variável que armazena caracteres e textos
    String nome = "Kleber";
    print("Olá $nome, seja bem vindo!");

    // Variável que armazena verdadeiro ou falso
    bool ligado = true;
    print("Ligado: $ligado");

    // Variável que guarda uma lista genérica
    List numerosGenericos = [10, "Kleber", true, 20];
    print(numerosGenericos);
```

```

// Variável que guarda uma lista de números inteiros
List<int> numerosInteiros = [10, 20, 30, 40];
print(numerosInteiros);

// Variável que guarda um dicionário com chave e valor em formato
texto
Map<String, String> nome_sobrenome = {"Kleber": "Andrade",
"Claudia": "Trevisan"};

// Variável sem tipo definido, seu tipo é igual ao tipo do primeiro
valor que recebe
var sobrenome = nome_sobrenome[nome];
print("O sobrenome do $nome é $sobrenome");

// Constantes (valores imutáveis)
const double pi = 3.1416;
print("O valor de PI é $pi");

// Variável dinâmica (neste momento é do tipo inteiro pois recebeu
o valor 10)
dynamic x = 10;
print(x);

// O tipo da variável pode ser alterada em tempo de execução
(agora é um texto)
x = "Curso";
print(x);
}

```

O resultado será:

```

Idade: 33
Raio: 10.25
Ola Kleber, seja bem vindo!
Ligado: true
[10, Kleber, true, 20]
[10, 20, 30, 40]
O sobrenome do Kleber é Andrade
O valor de PI é 3.1416
10
Curso

```

Figura 77 - Variáveis DART – resultado. Fonte: Medium.

Em *Flutter*, assim como em outras linguagens de programação baseadas em Dart, as variáveis são usadas para armazenar diferentes tipos de dados. Abaixo, apresentamos os principais tipos de variáveis que você encontrará em *Flutter*:

Int - armazena de qualquer número inteiro, seja ele negativo ou positivo;

Double - armazena números de pontos flutuantes;

String - são vetores de caracteres que podemos representar com aspas duplas ou aspas simples;

Bool – booleano, armazena valores true (verdadeiro) e false (falso);

Dynamic - atribuir valores de todos os outros tipos, permitindo a modificação desses valores em tempo de execução;

List – utilizado no Dart para criação de vetores do tipo lista, permitindo criá-las vazias ou atribuindo valores na criação;

Map - estrutura similar à uma List porém, trabalha com o conceito de chave e valor, ou seja, não se trata de uma lista simples.

Const – tem por objetivo, tornar um objeto uma constante, ou seja, quando se define um objeto como constante, esse não poderá ter seu valor de estado alterado após sua inicialização.

Os tipos int, double e num (pode ser inteiro ou ponto flutuante) fornecem diversos métodos e propriedades que podem ser utilizados para a transformação

e checagem de dados. Abaixo apresentamos os operadores de comparação e simples, utilizados no *Dart*.

Operador	Descrição
<code>>=</code>	Maior ou igual
<code>></code>	Maior
<code><=</code>	Menor ou igual
<code><</code>	Menor
<code> s</code>	Mesmo tipo
<code> s!</code>	Não é mesmo tipo
<code>==</code>	Igual
<code>!=</code>	Diferente
<code>&&</code>	E" lógico (AND)
<code> </code>	E" lógico (AND)
<code>+</code>	Adição
<code>-</code>	Subtração
<code>*</code>	Multiplicação
<code>/</code>	Divisão
<code>%</code>	Percentual (resto da divisão)

Tabela 1 - Tipos de operadores de comparação. Fonte: autoria própria.

Operadores Relacionais

Operadores relacionais são usados para comparações no Dart. O resultado de cada expressão é sempre um bool (boolenao), ou seja terá o valor que é true ou false.

Condicionais

As Condicionais são utilizadas para tomada de decisão no código, sendo:

```

void main(){
    double media = 4.9;

    // IF (condição verdadeira) / ELSE
    if (media < 6.0){
        print("Reprovado!");
    } else {
        print("Aprovado!");
    }

    /* Podemos também utilizar IF TERNÁRIO
     * CONDIÇÃO ? RETORNO VERDADEIRO : RETORNO FALSO
     */
    print(media < 6.0 ? "Reprovado!" : "Aprovado");

    /* Toda variável declarada e que não recebe valor, automaticamente
     é nula
     * VARIAVEL ?? RETORNO CASO SEJA NULO
     */
    String linguagem;
    print(linguagem ?? "Não Informado");

    linguagem = "Dart";
    print(linguagem ?? "Não Informado");

    /* SWITCH / CASE / DEFAULT
     * Utilizado geralmente quando temos constantes
     * Cada cláusula de case não vazio termina com uma instrução
     break, como regra.
     */
    switch(linguagem){
        case "Dart":
            print("É Dart!");
    }
}

```

```

        break;
    case "Java":
        print("É Java!");
        break;
    case "C#":
        print("É C#!");
        break;
    default:
        print("Não sabe no que programa");
    }
}

```

O resultado de execução deste código será:

```

Reprovado!
Reprovado!
Não Informado
Dart
É Dart!

```

Figura 78 - Condicionais DART – resultado. Fonte: Medium.

Repetições

As Repetições são utilizadas para processos de repetição no código, usando os comandos FOR, WHILE, DO/WHILE ou FOREACH sendo:

```

void main(){
    // Repetição de 0 a 5 (conhecemos o número inicial e final)
    // FOR (INICIO; CONDIÇÃO; INCREMENTO)
    for(int i = 0; i < 5; i++){
        print(i);
    }

    // Repetição de 0 a 5
    // INICIO; WHILE (CONDICAO){ INCREMENTO; }
    // Teste condicional no início
    int j = 0;      // Início
    while(j < 5){  // Condição

```

```

    print(j);
    j++;           // Incremento
}

// Repetição de 0 a 5
// INICIO; DO { INCREMENTO; } WHILE(CONDICAO);
// Teste condicional no final
int k = 0;      // Início
do {
    print(k);
    k++;           // Incremento
} while(k < 5); // Condição

// Conjunto de números (Lista)
List numeros = [0, 1, 2, 3, 4];

// FOREACH
// FOR (VARIAVEL DENTRO DO CONJUNTO)
for (int numero in numeros){
    print(numero);
}
}

```

Os resultados de execução desse código serão os valores 0, 1, 2, 3, 4 para cada tipo de repetição executada.

Funções

Funções são objetos e possuem um tipo- Função. Isso significa que as funções podem ser atribuídas a variáveis ou passadas como argumentos para outras funções.

```

// Função sem parâmetros
void escreverBemVindo(){

```

```

        print("Seja bem-vindo!");
    }

    // Quando a função só tem um comando interno, você pode usar desta
forma
    void escreverDesculpas() => print("Desculpa, encontramos um
erro.");

    // Função com passagem de parâmetros (podem ter quantos parâmetros
quiser)
    void calcularSoma(double a, double b){
        double resultado = a + b;
        print(resultado);
    }

    // Função que retorna uma variável do tipo double
    double calcularSubtracao(double a, double b){
        double resultado = a - b;
        return resultado;
    }

    // Exemplo reduzido de uma função que retorna valor
    double calcularAreaCirculo(double raio) => 3.14 * raio * raio;

    // Função com parâmetros opcionais (utiliza-se os parâmetros dentro
de chaves {})
    void exibirNomeCursoIdade(String nome, {int idade, String curso}) {
        if(idade != null && curso != null) {
            print("$nome tem $idade anos e faz o curso de $curso.");
        } else if(idade == null && curso != null) {
            print("$nome faz o curso de $curso.");
        } else if(idade != null && curso == null) {
            print("$nome tem $idade anos.");
        } else {
            print("Olá $nome");
        }
    }
}

```

```

// Passar funções como parâmetros
void calcular(double a, double b, Function funcao){
    funcao(a, b);
}

// Função principal
void main() {
    // Executando a função escreverBemVindo()
    escreverBemVindo();

    // Executando a função escreverDesculpas()
    escreverDesculpas();

    // Executando a função calcularSoma(a, b)
    calcularSoma(10, 20);

    // Executando a função calcularSubtracao(a, b)
    print(calcularSubtracao(10, 20));

    // Executando a função calcularAreaCirculo(raio)
    print(calcularAreaCirculo(10));

    // Execuntado a função exibirNomeCursoIdade(nome)
    exibirNomeCursoIdade("Kleber");

    // Execuntado a função exibirNomeCursoIdade(nome, idade)
    exibirNomeCursoIdade("Kleber", idade: 33);

    // Execuntado a função exibirNomeCursoIdade(nome, curso)
    exibirNomeCursoIdade("Kleber", curso: "Ciência da Computação");

    // Execuntado a função exibirNomeCursoIdade(nome, idade, curso)
    exibirNomeCursoIdade("Kleber", idade: 33, curso: "Ciência da
Computação");

    // Executando a função calcular(a, b, função), como função foi
passada a calcularSoma(a,b)
    calcular(30, 20, calcularSoma);
}

```

```
// Executando a função calcular(a, b, função), como função foi  
criado uma função anônima(a,b)  
  
calcular(30, 20, (a, b){  
    var resultado = a * b;  
    print(resultado);  
});  
}
```

O resultado de execução deste código será:

```
Seja bem-vindo!  
Desculpa, encontramos um erro.  
30  
-10  
314  
Ola Kleber  
Kleber tem 33 anos.  
Kleber faz o curso de Ciência da Computação.  
Kleber tem 33 anos e faz o curso de Ciência da Computação.  
50  
600
```

Figura 79 - Funções DART – resultado. Fonte: Medium.

5 *Flutter: conectando e manipulando dados em tempo real*

O *Flutter* dentre suas características, a sua capacidade de se conectar e manipular dados em tempo real, é uma funcionalidade essencial para uma ampla gama de aplicativos modernos. Alguns exemplos são: redes sociais, sistemas de chat, aplicativos de monitoramento em tempo real, entre outros.

Para conectar um aplicativo *Flutter* a serviços de dados em tempo real, você pode utilizar diversas tecnologias e ferramentas disponíveis. Para as práticas deste curso, utilizaremos a ferramenta `syncfusion_flutter_charts`.

5.1 Construindo gráficos no mundo Energético

O `syncfusion_flutter_charts` oferece uma ampla gama de tipos de gráficos, incluindo gráficos de linha, barra, coluna, área, pizza, radar, polar, *burndown* e muitos outros. Nesta seção, será possível desenvolver um código em *Flutter* que possibilite a visualização dinâmica dos dados coletados via MQTT.



Figura 80 – Exemplo de Gráficos – resultado. Fonte: `syncfusion_flutter_charts`.

Além da utilização do `syncfusion_flutter_charts`, se faz necessário criar uma matriz de *Data Point* conforme exemplo abaixo:

Adicionar a dependência ao pubspec.yaml:

```
yaml
dependencies:
  charts_flutter: ^0.12.0
```

Figura 81 – Exemplo de Data Point. Fonte: autoria própria.

Criar a classe DataPoint:

```
dart
class DataPoint {
  final int x;
  final int y;

  DataPoint(this.x, this.y);
}
```

Figura 82 – Exemplo de Data Point. Fonte: autoria própria.

Definir a matriz de DataPoint:

```
List<List<DataPoint>> dataMatrix = [
  [
    DataPoint(0, 3),
    DataPoint(1, 5),
    DataPoint(2, 2),
  ],
  [
    DataPoint(0, 4),
    DataPoint(1, 6),
    DataPoint(2, 3),
  ],
  [
    DataPoint(0, 2),
    DataPoint(1, 4),
    DataPoint(2, 5),
  ],
];
```

Figura 83 – Exemplo de Data Point. Fonte: autoria própria.

Utilizar a matriz de DataPoint em um gráfico:

```
List<charts.Series<DataPoint, int>> _createSeriesList() {
    return dataMatrix.map((dataPoints) {
        return charts.Series<DataPoint, int>(
            id: 'Data',
            colorFn: (_, _) => charts.MaterialPalette.blue.shadeDefault,
            domainFn: (DataPoint point, _) => point.x,
            measureFn: (DataPoint point, _) => point.y,
            data: dataPoints,
        );
    }).toList();
}

void main() {
    runApp(MaterialApp(
        home: LineChartSample(
            dataMatrix: [
                [
                    DataPoint(0, 3),
                    DataPoint(1, 5),
                    DataPoint(2, 2),
                ],
                [
                    DataPoint(0, 4),
                    DataPoint(1, 6),
                    DataPoint(2, 3),
                ],
                [

```



```
class LineChartSample extends StatelessWidget {
    final List<List<DataPoint>> dataMatrix;

    LineChartSample({required this.dataMatrix});

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Gráfico de Linhas com Matriz de DataPoint'),
            ),
            body: Center(
                child: Container(
                    height: 400.0,
                    padding: EdgeInsets.all(20.0),
                    child: charts.LineChart(
                        _createSeriesList(),
                        animate: true,
                    ),
                ),
            ),
        );
    }
}
```

```

        DataPoint(0, 2),
        DataPoint(1, 4),
        DataPoint(2, 5),
    ],
],
),
);
}
}

```

Figura 84 – Exemplo de Data Point. Fonte: autoria própria.

5.2 Aplicativo básico com lista

Para criar um aplicativo *Flutter* que permite ao usuário adicionar valores a uma lista e exibir esses valores. O aplicativo terá dois *TextField* para a entrada de dados e um *ElevatedButton* para adicionar os valores à lista.

Siga os passos abaixo para criar o aplicativo:

Configuração Inicial

Crie um novo projeto *Flutter*.

Edite o arquivo **main.dart**: Substitua o conteúdo de **lib/main.dart** pelo seguinte código:

```

import 'package:flutter/material.dart';

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

// Widget principal do aplicativo
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Lista Flutter', // Título do aplicativo
      theme: ThemeData(
        primarySwatch:
          Colors.blue, // Define a cor primária do tema do
aplicativo
    )
  }
}

```

```

        ),
        home: ListaPage(), // Define a página inicial do
aplicativo
    );
}
}

// StatefulWidget que mantém o estado do widget
class ListaPage extends StatefulWidget {
    @override
    _ListaPageState createState() => _ListaPageState();
}

// Estado do StatefulWidget ListaPage
class _ListaPageState extends State<ListaPage> {
    final List<String> _lista = [];// Lista que armazenará os
itens adicionados
    final TextEditingController _textController1 =
        TextEditingController(); // Controlador para o primeiro
TextField
    final TextEditingController _textController2 =
        TextEditingController(); // Controlador para o segundo
TextField

    // Função para adicionar itens à lista
    void _adicionarItem() {
        // Verifica se ambos os campos de texto não estão vazios
        if (_textController1.text.isNotEmpty &&
_textController2.text.isNotEmpty) {
            setState(() {
                // Adiciona a combinação dos textos dos campos de texto
à lista
                _lista.add('${_textController1.text}
${_textController2.text}');
                // Limpa os campos de texto
                _textController1.clear();
                _textController2.clear();
            });
        }
    }

    @override

```

```

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Lista Flutter'), // Título da AppBar
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0), // Padding em torno
      dos widgets
      child: Column(
        children: <Widget>[
          // Primeiro TextField para entrada de dados
          TextField(
            controller: _textController1,
            decoration:
              InputDecoration(labelText: 'Item 1'), // Rótulo do TextField
          ),
          // Segundo TextField para entrada de dados
          TextField(
            controller: _textController2,
            decoration:
              InputDecoration(labelText: 'Item 2'), // Rótulo do TextField
          ),
          SizedBox(height: 20), // Espaçamento entre os
          TextFields e o botão
          // Botão para adicionar itens à lista
          ElevatedButton(
            onPressed:
              _adicionarItem, // Chama a função
            _adicionarItem ao pressionar o botão
            child: Text('Adicionar à Lista'), // Texto do
            botão
          ),
          // Widget para exibir a lista de itens
          Expanded(
            child: ListView.builder(
              itemCount: _lista.length, // Número de itens na
              lista
              itemBuilder: (context, index) {
                return ListTile(

```

```

        title: Text(_lista[index]), // Exibe o item
da lista
            );
        },
    ),
),
],
),
),
);
}
}
}

```

Lógica do Programa:

Estrutura Inicial:

MyApp: é o widget principal que configura o MaterialApp e define o tema do aplicativo. Ele especifica que a página inicial do aplicativo será **ListaPage**.

ListaPage: é um StatefulWidget que permite manter o estado da lista e dos controladores de texto entre as reconstruções do widget.

_ListaPageState:

- **_lista**: uma lista de strings que armazena os itens adicionados.
- **_textController1** e **_textController2**: controladores para os dois campos de texto, permitindo obter o texto digitado pelo usuário e controlar o texto nesses campos.
- **_adicionarItem()**: função chamada quando o botão é pressionado. Ela verifica se ambos os campos de texto não estão vazios, adiciona a combinação dos textos à lista **_lista** e, em seguida, limpa os campos de texto.

Interface do Usuário (UI):

Scaffold: estrutura básica do widget que fornece uma estrutura visual para o aplicativo. Inclui uma AppBar, um corpo com padding e uma coluna de widgets.

TextFields: dois campos de texto para a entrada dos itens.

ElevatedButton: Um botão que, ao ser pressionado, chama a função `_adicionarItem`.

ListView.builder: um widget que constrói a lista de itens dinamicamente. Ele usa o comprimento da lista `_lista` para determinar quantos itens exibir e cria um **ListTile** para cada item na lista.

Tela do aplicativo criado:



Figura 85 – Aplicativo MQTT. Fonte: autoria própria.

5.3 Aplicativo MQTT básico

A seguir, temos um exemplo básico de um aplicativo *Flutter* que permite a entrada do TOPIC e do Broker MQTT, além de um botão para conectar e mostrar as informações recebidas. Para implementar essa funcionalidade, vamos usar a biblioteca `mqtt_client`.

Siga os passos abaixo para criar o aplicativo:

Configuração Inicial

Crie um novo projeto *Flutter*.

Adicione as dependências `mqtt_client` ao seu arquivo `pubspec.yaml`.

Execute este comando:

```
flutter pub add mqtt_client
```

Execute `flutter pub get` para instalar as dependências.

Explicação Detalhada:

Vamos analisar todas as partes do código , entendendo as funções e classes.

Importações:

`package:flutter/material.dart`: Importa o pacote principal do *Flutter*.

`package:mqtt_client/mqtt_client.dart` e

`package:mqtt_client/mqtt_server_client.dart`: Importa as bibliotecas necessárias para trabalhar com o protocolo MQTT.

Função principal (main):

Inicializa o aplicativo *Flutter*.

Classe MyApp:

- Define o widget principal do aplicativo.
- Usa **MaterialApp** para configurar o tema e a rota inicial do aplicativo.

Classe MqttApp:

Define um **StatefulWidget** que mantém o estado do aplicativo.

Classe _MqttAppState:

- Define o estado para MqttApp.
- Possui controladores de texto para capturar a entrada do usuário (tópico e broker).
- Mantém variáveis para o status da conexão e a mensagem recebida.
- Possui um cliente MQTT (**_client**).

Método _connect:

- Inicializa e configura o cliente MQTT.
- Define mensagens de conexão e configura callbacks para eventos de conexão, desconexão e inscrição.
- Tenta conectar ao broker e inscreve-se no tópico fornecido.
- Adiciona um listener para atualizar a mensagem recebida no estado do widget.

Callbacks _onDisconnected, _onConnected, _onSubscribed:

Atualize o estado do widget com o status da conexão, inscrição e desconexão.

Método build:

- Constrói a interface do usuário.
- Inclui campos de entrada para o broker e o tópico, um botão para conectar, e exibe o status da conexão e a mensagem recebida.

Este código cria um aplicativo funcional que se conecta a um broker MQTT, se inscreve em um tópico e exibe mensagens recebidas, com uma interface de usuário simples para facilitar a interação.

Código completo

Edite o arquivo **main.dart**: Substitua o conteúdo de **lib/main.dart** pelo seguinte código:

```
// Importa os pacotes necessários para o aplicativo Flutter e a
biblioteca MQTT.
import 'package:flutter/material.dart';
import 'package:mqtt_client/mqtt_client.dart';
import 'package:mqtt_client/mqtt_server_client.dart';

// Função principal que inicia o aplicativo.
void main() {
  runApp(MyApp());
}

// Classe MyApp que representa o widget principal do aplicativo.
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MqttApp(), // Define o widget inicial como MqttApp.
    );
  }
}

// Classe MqttApp que representa o estado do widget.
class MqttApp extends StatefulWidget {
  @override
  _MqttAppState createState() => _MqttAppState();
}

// Estado do widget MqttApp.
class _MqttAppState extends State<MqttApp> {
  // 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 = "Desconectado";
  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, '');
    _client.logging(on: true); // Habilita o log para depuração.

    // 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()
        .withClientIdentifier('flutter_client') // Identificador
        do cliente.
        .startClean() // Inicia uma sessão limpa.
        .withWillQos(
            MqttQos.atLeastOnce); // Define a qualidade de
        serviço (QoS).
    _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 = 'Falha na conexão: $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 = 'Desconectado';
    });
}

// Callback para quando o cliente é conectado.
void _onConnected() {
    setState(() {
        _status = 'Conectado';
    });
}

// Callback para quando o cliente se inscreve em um tópico.
void _onSubscribed(String topic) {
    setState(() {
        _status = 'Inscrito no tópico: $topic';
    });
}

@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text('MQTT App'),
        ),
        body: Padding(

```

```
padding: const EdgeInsets.all(16.0),
child: Column(
  children: [
    // Campo de entrada para o broker MQTT.
    TextField(
      controller: _brokerController,
      decoration: InputDecoration(labelText: 'Broker
MQTT'),
    ),
    // Campo de entrada para o tópico MQTT.
    TextField(
      controller: _topicController,
      decoration: InputDecoration(labelText: 'Tópico'),
    ),
    SizedBox(height: 20),
    // Botão para conectar ao broker MQTT.
    ElevatedButton(
      onPressed: _connect,
      child: Text('Conectar'),
    ),
    SizedBox(height: 20),
    // Exibe o status atual da conexão.
    Text('Status: ${_status}'),
    SizedBox(height: 20),
    // Exibe a mensagem recebida do broker MQTT.
    Text('Mensagem Recebida: ${_receivedMessage}'),
  ],
),
),
);
});
```

Lógica Utilizada no Desenvolvimento

Captura de Entrada do Usuário:

TextEditingController é usado para capturar as informações fornecidas pelo usuário (broker e tópico).

Facilita a recuperação dos valores quando necessário.

Gestão do Estado:

O uso de um **StatefulWidget** permite que a interface do usuário seja atualizada automaticamente quando o estado muda (por exemplo, quando uma mensagem é recebida).

Conexão ao Broker MQTT:

- Inicializa um cliente MQTT com o endereço do broker fornecido.
- Configura callbacks para eventos de conexão, desconexão e inscrição.
- Tenta se conectar ao broker e, se falhar, atualiza o estado do widget para refletir a falha.

Manipulação de Mensagens:

- Inscreve-se no tópico fornecido pelo usuário.
- Configura um listener para atualizar o estado do widget quando uma nova mensagem é recebida.
- Converte a carga útil da mensagem recebida para uma string e a exibe na interface do usuário.

Interface do Usuário:

- Cria campos de entrada para o usuário fornecer o broker e o tópico.
- Um botão para iniciar a conexão ao broker.
- Text widgets para exibir o status da conexão e a mensagem recebida.

Tela do aplicativo criado:

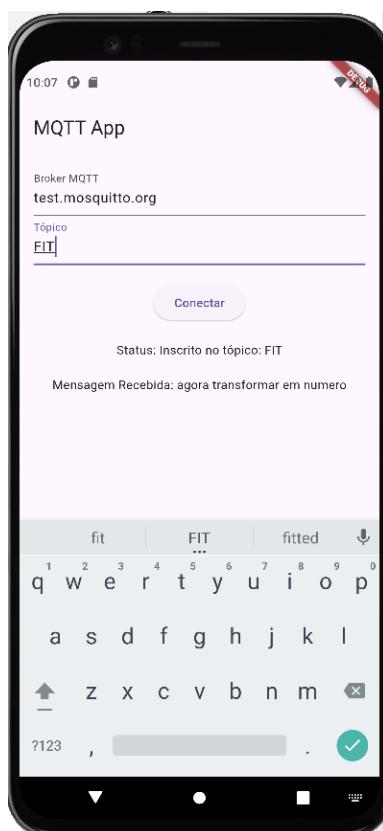


Figura 86 – Aplicativo MQTT. Fonte: autoria própria.

5.4 Aplicativo para criar um gráfico de linha

Para criar um programa em *Flutter* que permite inserir valores em dois *TextField*, um para o eixo X e outro para o eixo Y, e usar a biblioteca syncfusion_flutter_charts para exibir esses valores em um gráfico de linha.

Passos para Criar o Aplicativo

Configuração Inicial

Crie um novo projeto Flutter.

Adicione as dependências syncfusion_flutter_charts ao seu arquivo pubspec.yaml.

Execute este comando:

```
flutter pub add syncfusion_flutter_charts
```

Execute flutter pub get para instalar as dependências.

Código Completo

Edite o arquivo **main.dart**: Substitua o conteúdo de **lib/main.dart** pelo seguinte código:

```

import 'package:flutter/material.dart'; // Importa o pacote principal do Flutter para construir a interface do usuário.
import 'package:syncfusion_flutter_charts/charts.dart'; // Importa a biblioteca Syncfusion para gráficos.

void main() {
    runApp(MyApp()); // O ponto de entrada da aplicação. Executa o widget MyApp.
}

class MyApp extends StatelessWidget {
    // Define o widget principal da aplicação.
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Line Chart', // Define o título da aplicação.
            theme: ThemeData(
                primarySwatch: Colors
                    .blue, // Define o tema da aplicação com uma cor primária azul.
            ),
            home: LineChartPage(), // Define a página inicial da aplicação.
        );
    }
}

class LineChartPage extends StatefulWidget {
    // Define um widget stateful para a página do gráfico de linha.
    @override
    _LineChartPageState createState() =>
        _LineChartPageState(); // Cria o estado do widget.
}

class _LineChartPageState extends State<LineChartPage> {
    final TextEditingController _xController =
        TextEditingController(); // Controlador para o campo de texto do eixo X.
    final TextEditingController _yController =
        TextEditingController(); // Controlador para o campo de texto do eixo Y.
}

```

```

List<_DataPoint> _dataPoints =
    [];// Lista para armazenar os pontos de dados inseridos.

void _addDataPoint() {
    final double? x = double.tryParse(_xController
        .text);// Tenta converter o texto do campo X para um
número.
    final double? y = double.tryParse(_yController
        .text);// Tenta converter o texto do campo Y para um
número.

    if (x != null && y != null) {
        // Verifica se os valores são válidos.
        setState(() {
            // 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.
    }
}

@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text('Gráfico de linha'),// Título da barra de
aplicativos.
        ),
        body: Padding(
            padding: const EdgeInsets.all(
                16.0),// Define o preenchimento de 16 pixels ao
redor do corpo.
            child: Column(
                children: <Widget>[
                    Row(
                        children: <Widget>[
                            Expanded(
                                child: TextField(
                                    controller:

```

```

        _xController, // Controlador para o
campo de texto do eixo X.
        decoration: InputDecoration(
            labelText: 'Valor eixo X'), // Rótulo do
campo de texto.
        keyboardType:
            TextInputType.number, // Define o
teclado numérico.
    ),
),
SizedBox(
    width: 16), // Espaço de 16 pixels entre os
campos de texto.
Expanded(
    child: TextField(
        controller:
            _yController, // Controlador para o
campo de texto do eixo Y.
        decoration: InputDecoration(
            labelText: 'Valor eixo Y'), // Rótulo do
campo de texto.
        keyboardType:
            TextInputType.number, // Define o
teclado numérico.
    ),
),
],
),
SizedBox(
    height: 16), // Espaço de 16 pixels abaixo dos
campos de texto.
ElevatedButton(
    onPressed:
        _addDataPoint, // Chama a função _addDataPoint
quando o botão é pressionado.
    child: Text('Inserir'), // Texto do botão.
),
SizedBox(height: 16), // Espaço de 16 pixels abaixo
do botão.
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.
}

```

Explicação Detalhada:

Importações:

- **package:flutter/material.dart:** Importa os componentes essenciais do Flutter para a construção da interface do usuário.

- **package:syncfusion_flutter_charts/charts.dart:** Importa a biblioteca Syncfusion para criar gráficos.

Função main:

runApp(MyApp()): Inicia a aplicação executando o widget MyApp.

Classe MyApp:

Um widget stateless que configura o título, tema e a página inicial da aplicação (LineChartPage).

Classe LineChartPage:

Um widget stateful que representa a página principal do aplicativo, onde serão inseridos os dados e exibido o gráfico de linha.

Classe _LineChartPageState:

- Define o estado do widget **LineChartPage**.
- **TextEditingController _xController** e **TextEditingController _yController**: Controladores para os campos de texto dos valores X e Y.
- **List<_DataPoint> _dataPoints**: Lista que armazena os pontos de dados inseridos.

Método _addDataPoint:

- Converte os valores dos campos de texto para **double** e adiciona-os à lista **_dataPoints** se forem válidos.
- Limpa os campos de texto após a inserção.

Método build:

Cria a interface do usuário com uma **Scaffold** contendo um **AppBar** e um corpo com:

- Dois **TextField** para a entrada dos valores X e Y.
- Um botão **ElevatedButton** para inserir os valores.
- Um gráfico **SfCartesianChart** que exibe os pontos de dados inseridos como uma série de linha (**LineSeries**).

Classe _DataPoint:

Representa um ponto de dados com valores X e Y.

Comportamento do Código:

Entrada de Dados:

- O usuário pode inserir valores nos campos de texto para os eixos X e Y.
- Ao pressionar o botão "Inserir", os valores são adicionados à lista de pontos de dados e os campos de texto são limpos.

Exibição de Dados:

- O gráfico de linha é atualizado automaticamente para refletir os novos pontos de dados inseridos.
- O gráfico usa LineSeries para conectar os pontos com linhas, criando um gráfico de linha.

Componentes Gráficos:

- **SfCartesianChart** é o contêiner principal para o gráfico.
- **NumericAxis** define os eixos X e Y como numéricos.
- **LineSeries<_DataPoint, double>** mapeia os pontos de dados para o gráfico de linha.

Tela do aplicativo criado:

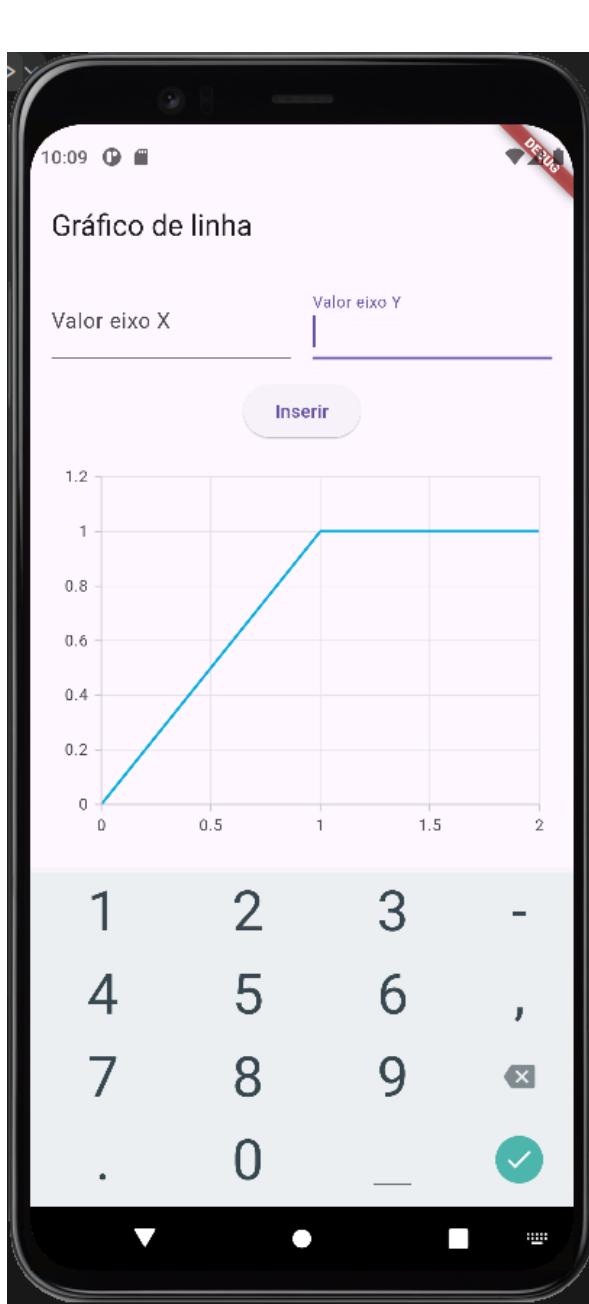


Figura 87 – Aplicativo Gráfico. Fonte: autoria própria.

5.5 Flutter Gráfico e MQTT: Visualização Dinâmica de Dados em Ação

Passos para Criar o Aplicativo

Configuração Inicial

Crie um novo projeto *Flutter*.

Adicione as dependências `mqtt_client` e `syncfusion_flutter_charts` ao seu arquivo `pubspec.yaml`.

Execute este comando:

```
flutter pub add syncfusion_flutter_charts  
flutter pub add mqtt_client
```

Execute flutter pub get para instalar as dependências.

Código Completo

No arquivo lib/main.dart, insira o seguinte código:

```

import 'package:flutter/material.dart';
import 'package:mqtt_client/mqtt_client.dart';
import 'package:mqtt_client/mqtt_server_client.dart';
import 'package:syncfusion_flutter_charts/charts.dart';
import 'package:intl/intl.dart'; // Importação do pacote intl

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

// Classe principal do aplicativo
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'MQTT Chart', // Título do aplicativo
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(), // Página inicial do aplicativo
    );
  }
}

// Classe que representa a página inicial do aplicativo
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

// Estado associado à página inicial
class _MyHomePageState extends State<MyHomePage> {
  final TextEditingController _brokerController =
    TextEditingController(); // Controlador para o TextField
do broker
  final TextEditingController _topicController =
    TextEditingController(); // Controlador para o TextField
do tópico
  MqttServerClient? _client; // Cliente MQTT
  List<ChartData> _chartData = []; // Lista para armazenar os
dados do gráfico
  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 Scaffold(
    appBar: AppBar(
      title: Text('MQTT Chart'), // Título da AppBar
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          TextField(
            controller:
              _brokerController, // Controlador do TextField
do broker
            decoration: InputDecoration(
              border: OutlineInputBorder(),
              labelText: 'Broker MQTT', // Rótulo do TextField
            ),
          ),
          SizedBox(height: 10), // Espaço entre os TextFields
          TextField(
            controller:
              _topicController, // Controlador do TextField
do tópico
            decoration: InputDecoration(
              border: OutlineInputBorder(),
              labelText: 'Topic', // Rótulo do TextField
            ),
          ),
          SizedBox(height: 20), // Espaço antes do botão
          ElevatedButton(
            onPressed: _connect, // Chama a função _connect ao
clickar
            child: Text('Conectar'), // Texto do botão
          ),
          SizedBox(height: 20), // Espaço antes do gráfico
          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: 'Hora'), // Título do
            eixo X
            ),
            primaryYAxis: NumericAxis(
                title: AxisTitle(text: 'Valor'), // 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
}
```

Tela do aplicativo criado:

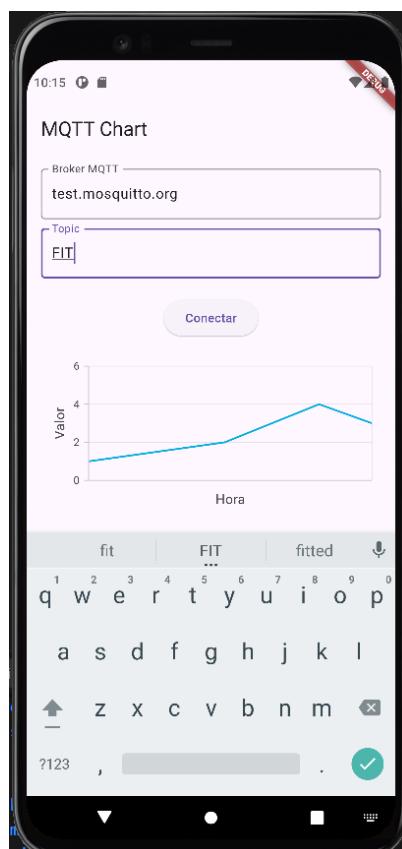


Figura 88 – Criação do App com Gráfico. Fonte: autoria própria.

6 Atividade final do curso com Aprendizagem Baseada em Projeto

Chegou a hora de colocar em prática tudo o que aprendemos! Vocês terão a oportunidade de desenvolver uma aplicação *mobile* em *Flutter* que importará os dados a partir de um *Broker MQTT* de um projeto IoT pronto, e os transformará em gráficos para facilitar a leitura e interpretação.

Os exercícios práticos realizados no *Flutter*. Durante o curso, deverão ser inseridos em um relatório de Atividade Final do Curso. A criação desse documento é fundamental para consolidar o aprendizado, facilitando a revisão dos conceitos abordados e permitindo a avaliação do progresso ao longo do curso. O modelo do relatório está disponível na nossa Plataforma de Educação do FIT.

Certifiquem-se de realizar a entrega do relatório final no formato PDF, de forma individual, mesmo que algumas atividades tenham sido realizadas em grupo. No documento, esperamos ver os dados obtidos durante as aulas.

Fiquem tranquilos, pois os instrutores estarão disponíveis para orientar sobre o preenchimento do documento (original em Word) e ajudar no acesso e entrega na Plataforma de Educação do FIT.

O documento deverá conter:

- Print da geração de dados para o Broker MQTT;
- Código do Programa em *Flutter*;
- Print da execução do aplicativo em *Flutter*, do qual deverá apresentar os gráficos.

Importante salientar que o relatório de Atividade Final do curso, deverá ser entregue na plataforma Educação do FIT. Qualquer outro meio de envio será descartado e considerado como documento “não entregue”.

Conclusão

A abordagem desta apostila permite ao leitor uma interação ativa no desenvolvimento de uma aplicação *mobile* em *Flutter*. Este aplicativo que irá gerar gráficos dinâmicos a partir de dados importados de um Broker MQTT, originários de um projeto IoT já pronto (Wokwi).

O aprendizado de como desenvolver um aplicativo mobile em *Flutter*, foi direcionado para estimular a autonomia dos alunos, promover o senso crítico e contribuir para uma aprendizagem mais efetiva. Assim, o conteúdo do curso *Flutter* para *Smart Grid* familiariza o leitor com o mundo tecnológico.

Obrigada por fazer parte do curso e ter realizado a leitura desta apostila. Espero que o interesse pelo *Flutter* para *Smart Grid* apresentado neste curso esteja aguçado, para que você pratique e conheça ainda mais as maravilhas do mundo de IoT integrado ao desenvolvimento *mobile*.

Aproveite as recomendações de leitura disponíveis nas referências para um maior aprofundamento sobre o tema. Esperamos que sua jornada de aprendizagem tenha sido divertida, dinâmica e interativa, como a Metodologia Ativa utilizada em aulas! Não deixe de responder nossa pesquisa de satisfação, para que possamos sempre melhorar a jornada de aprendizagem oferecida a nossos alunos!

Referências

Bibliografia básica:

FLUTTER. Documentação do Flutter. Disponível em: <https://flutter.dev/>. Acesso em: junho 2024.

FLUTTER PARA INICIANTES. Disponível em: <https://www.flutterparainiciantes.com.br>. Acesso em: maio 2024.

PROGRAMAÇÃO DART. Disponível em: <https://medium.com/flutter-comunidade-br/introdu%C3%A7%C3%A3o-a-linguagem-de-programa%C3%A7%C3%A3o-dart-b098e4e2a41e>. Acesso em: maio 2024.

SYNCFUSION_FLUTTER_CHARTS. Ferramenta para geração de gráficos no Flutter. Disponível em: https://pub.dev/packages/syncfusion_flutter_charts. Acesso em: junho 2024.

AMIN, S. Massoud; WOLLENBERG, B. F. Toward a smart grid: power delivery for the 21st century. IEEE Power and Energy Magazine, v. 3, n. 5, p. 34-41, Sept.-Oct. 2005. DOI: 10.1109/MPAE.2005.1507024.

CONTROLE DE REVISÃO DO DOCUMENTO / DOCUMENT REVISION CONTROL

Revisão	Descrição	Razão	Autor	Data
A	Elaboração inicial e Revisão pedagógica	Elaboração inicial	Bruno Rodrigues, Rita Barbosa, Priscila Santos e Larissa Alves	24/07/2024



FUTURO DO TRABALHO, TRABALHO DO FUTURO

Bom curso



MINISTÉRIO DA
CIÊNCIA, TECNOLOGIA
E INOVAÇÃO

GOVERNO FEDERAL
BRASIL
UNIÃO E RECONSTRUÇÃO