Capítulo 1: O Alicerce - HTML (A Estrutura) e CSS (A Aparência)

Objetivo deste Capítulo: Construir a interface visual completa do nosso "Painel de Controle MQTT".

1.1 - O que é HTML? O Esqueleto

Pense no HTML (HyperText Markup Language) como a planta baixa ou o esqueleto de um site. Ele não diz qual é a cor da parede, mas diz onde a parede está, onde fica a porta e onde fica a janela. Usamos "tags" (etiquetas) para descrever o conteúdo. Por exemplo:

```
<h1> diz: "Isto é um título principal."</h1>
<p> diz: "Isto é um parágrafo de texto."</p>
```

Todo projeto HTML precisa de um ponto de partida, um arquivo que chamaremos de “index.html”. Crie este arquivo em uma pasta para o nosso projeto.

index.html

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Meu Painel MQTT</title>
   <link rel="stylesheet" href="style.css">
</head>

<body>
</body>
</html>
```

1.2 - Construindo o HTML do Nosso Painel

Vamos agora adicionar os elementos do nosso painel dentro da tag <body>. Usaremos tags semânticas como <main> (que diz "este é o conteúdo principal") e (uma "caixa" genérica para organizar).

```
<body>
   <main class="container">
      <h1>Painel de Controle MQTT</h1>
      <div class="status-box">Status: <span id="status-text">Desconectado</span></div>
      <div class="form-control">
         <label for="topic">Tópico:</label>
         <input type="text" id="topic" value="meu/teste/senai">
        <label for="message">Mensagem:</label>
         <input type="text" id="message" placeholder="Digite sua mensagem...">
         <button id="btn-publicar">Publicar</button>
      </div>
      <h2>Mensagens Recebidas:</h2>
      <div id="messages-box" class="messages-box">
         <p class="mensagem-placeholder">Aguardando mensagens...</p>
      </div>
   </main>

   <script src="https://cdnjs.cloudflare.com/ajax/libs/paho-mqtt/1.0.1/mqttws31.min.js"></script>
   <script src="app.js"></script>
</body>
```

1.3 - O que é CSS? A Aparência

Pense no CSS (Cascading Style Sheets) como a equipe de decoração e pintura da casa. O HTML construiu a parede, o CSS decide a cor, a textura, o quadro que vai nela e se ela fica lado a lado com outra parede ou embaixo.

O CSS funciona com Seletores e Regras:

```
seletor { propriedade: valor; }
```

- Crie o arquivo style.css na mesma pasta do index.html.

- Os seletores “body, h1 e button” aplicam o estilo a todas as tags daqueles tipos.
- Classes começando com “ . ”, por exemplo “.container” aplicam o estilo a qualquer tag que tenha class="container". É a forma mais comum e reutilizável de aplicar estilos.
- Podemos criar IDs como “#status-text, #btn-publicar” que começam com #. ID Aplica o estilo ao único elemento que tenha o como em id="status-text". IDs devem ser únicos na página.

1.4 - Estilizando o Nosso Painel (o style.css)

Vamos adicionar o conteúdo ao nosso arquivo style.css. Vou comentar cada bloco para explicar o "porquê".

style.css

```
/* Um "Reset" básico. Isso remove margens padrão do navegador e define uma fonte mais agradável. */
body {
   font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Arial, sans-serif;
   background-color: #f0f2f5; /* Um cinza bem claro para o fundo */
   margin: 0;
   padding: 20px; /* Dá um respiro nas bordas da tela */
   /* Centralizando nosso painel na tela */
   display: flex;
   justify-content: center;
   align-items: flex-start; /* Alinha no topo */
   min-height: 100vh; /* Altura mínima de 100% da tela */
}
/* Estilizando o seletor de CLASSE ".container". Esta é a nossa caixa principal */
.container {
   background-color: #ffffff; /* Fundo branco */
   border-radius: 8px; /* Bordas arredondadas */
   box-shadow: 0 4px 12px rgba(0, 0, 0, 0.08); /* Sombra sutil */
   padding: 24px;
   width: 100%; /* Ocupa 100% do espaço disponível... */
   max-width: 600px; /* ...mas no máximo 600px de largura */
   box-sizing: border-box; /* Faz o padding não aumentar a largura total */
}
/* Estilizando MÚLTIPLAS tags de uma vez */
h1, h2 {
   color: #333;
   border-bottom: 2px solid #eee;
   padding-bottom: 8px;
   margin-top: 0; /* Remove margem do topo do h1 */
}
h2 {
   font-size: 1.2rem; /* Tamanho um pouco menor para o h2 */
   margin-top: 20px;
}
/*Estilizando o seletor de CLASSE ".form-control". Usamos Flexbox para organizar os itens do formulário.*/
.form-control {
   display: flex;
   flex-direction: column; /* Empilha os itens verticalmente */
   gap: 12px; /* Adiciona um espaço de 12px ENTRE cada item */
   margin-bottom: 20px;
}
/* Estilizando as tags <label> e <input> */
label {
   font-weight: bold;
   color: #555;
   margin-bottom: -8px; /* Puxa o label para mais perto do input */
}
input[type="text"] {
   padding: 10px;
   border: 1px solid #ccc;
   border-radius: 4px;
   font-size: 1rem;
}
/* Estilizando o seletor de ID "#btn-publicar" */
#btn-publicar {
   background-color: #007BFF; /* Azul primário */
   color: white;
   font-weight: bold;
   border: none;
   padding: 12px;
```

```
   border-radius: 4px;
   font-size: 1rem;
   cursor: pointer; /* Muda o cursor para "mãozinha" */
   transition: background-color 0.2s; /* Efeito suave ao passar o mouse */
}
/* "Pseudo-classe" :hover. Define o estilo QUANDO o mouse está em cima do botão.*/
#btn-publicar:hover {
   background-color: #0056b3; /* Um azul mais escuro */
}
/* Caixa de Status */
.status-box {
   margin-bottom: 20px;
   font-size: 1.1rem;
}
/* Estilizando o seletor de ID "#status-text" */
#status-text {
   font-weight: bold;
   color: #d9534f; /* Vermelho (cor de "Desconectado") */
   /* Vamos mudar essa cor via JS no Cap. 2 */
}
/* Caixa de Mensagens */
.messages-box {
   border: 1px solid #eee;
   background-color: #fafafa;
   min-height: 150px;
   max-height: 300px;
   overflow-y: auto; /* Adiciona barra de rolagem se o conteúdo transbordar */
   padding: 15px;
   border-radius: 4px;
}
.mensagem-placeholder {
   color: #999;
   font-style: italic;
}
```

Capítulo 2: O Cérebro - JavaScript Básico (DOM e Eventos)

O Objetivo agora é fazer nossa página estática se tornar interativa. Vamos "escutar" o clique no botão, ler os valores dos campos de texto e manipular o HTML para exibir informações dinamicamente.

2.1 - O que é JavaScript? O Cérebro

Se o HTML é o esqueleto e o CSS é a pele, o JavaScript (JS) é o sistema nervoso e os músculos. Ele é a linguagem de programação que roda diretamente no navegador e nos permite:

- Manipular o DOM: Alterar o HTML e o CSS da página depois que ela já carregou.
- Reagir a Eventos: Executar um código quando o usuário faz algo (ex: clica, digita, passa o mouse).
- Comunicar-se: Fazer requisições para servidores e, como veremos no próximo capítulo, conectar-se a brokers MQTT.

Onde o código JS fica? No nosso index.html, nós já incluímos a tag **<script src="app.js"></script>** no final do <body>. Agora, vamos criar esse arquivo app.js na mesma pasta. Todo o nosso código JS deste capítulo irá dentro dele.

2.2 - Passo 1: Selecionando os Elementos (O DOM)

Antes de manipular qualquer elemento do HTML, o JS precisa "encontrá-lo". Usamos métodos do objeto document para isso. O mais comum e direto é o getElementById, que busca um elemento pelo id único que demos a ele. No seu arquivo app.js, comece "guardando" os elementos que vamos usar em variáveis. Usamos const (constante) porque a referência a esse elemento específico (o botão, o input) não vai mudar.

app.js

```
// Guardamos referências aos elementos HTML que vamos manipular.
const topicInput = document.getElementById('topic');
```

```
const messageInput = document.getElementById('message');

const publishButton = document.getElementById('btn-publicar');

const statusText = document.getElementById('status-text');
const messagesBox = document.getElementById('messages-box');

const placeholder = document.querySelector('.mensagem-placeholder');

console.log('JavaScript carregado!');
console.log(publishButton); // Ótimo para testar se o elemento foi encontrado
```

Teste Rápido: Salve o app.js, abra seu index.html no navegador e abra o "Console" do desenvolvedor (clique com o botão direito > Inspecionar > aba Console). Você deve ver a mensagem "JavaScript carregado!" e o objeto do seu botão.

2.3 - Passo 2: Ouvindo Eventos

Agora, queremos que algo aconteça quando o usuário clicar no botão. Para isso, adicionamos um "ouvidor de evento" (addEventListener) ao nosso botão.

A sintaxe é: elemento.addEventListener('tipo_do_evento', funcao_a_executar);

Adicione no app.js (Continue adicionando ao final)

```
// Dizemos ao JS o que fazer quando um evento (como 'click') acontecer.
publishButton.addEventListener('click', handlePublish);
```

Isso é como dizer: "Ei, publishButton! Quando alguém 'clicar' em você, execute a função chamada handlePublish." Mas... que função é essa? Vamos criá-la.

2.4 - Passo 3: Criando Funções e Manipulando o DOM

Uma função é um bloco de código nomeado que podemos executar quando quisermos. É aqui que a "mágica" acontece. Vamos criar a função handlePublish.

app.js (Continue adicionando ao final)

```
function handlePublish() {
   console.log("Botão clicado!");
   const topico = topicInput.value;  //Usamos a propriedade '.value' para pegar o texto dentro de um input
   const mensagem = messageInput.value;
   if (!topico || !mensagem) {  // Não queremos publicar se os campos estiverem vazios
      alert("Por favor, preencha tanto o Tópico quanto a Mensagem.");
      return; //Para a execução da função aqui
   }
   console.log(`Simulando publicação: Tópico='${topico}', Mensagem='${mensagem}'`);
   // Deixa o app pronto para a próxima mensagem
   messageInput.value = "";
   messageInput.focus();
}
```

Teste Agora: Salve o app.js e atualize seu navegador. Preencha os campos "Tópico" e "Mensagem". Clique em "Publicar". Olhe o "Console": você verá sua mensagem logada! O campo "Mensagem" deve ter sido limpo. Tente clicar em "Publicar" com um campo vazio: você verá o alert.

2.5 - Passo 4: Funções Auxiliares (Boa Prática)

Nosso app precisa de duas coisas que faremos muito:

Mudar o texto e a cor do Status e adicionar uma nova mensagem na caixa de mensagens. É uma ótima prática criar funções pequenas e dedicadas para isso.

app.js (Adicione estas novas funções)

```
//Atualiza o texto e a cor do status da conexão.
```

```
//@param {string} texto - O novo texto para o status.
//@param {string} cor - A cor (ex: 'green', 'red', '#d9534f').
function setStatus(texto, cor) {
   statusText.textContent = texto;
   statusText.style.color = cor;
}
//Adiciona uma nova linha de mensagem na caixa de mensagens.
//@param {string} topico - O tópico da mensagem.
//@param {string} mensagem - O conteúdo (payload) da mensagem.
function addMessageToBox(topico, mensagem) {
   // Se o placeholder ("Aguardando mensagens...") ainda estiver lá, remova-o.
   if (placeholder) {
      placeholder.style.display = 'none'; // Esconde o placeholder
   }
   // Cria um novo elemento <p> para a mensagem
   const novaMensagemElemento = document.createElement('p');
   // Adiciona o conteúdo HTML de forma segura. (innerHTML é mais flexível aqui)
   novaMensagemElemento.innerHTML = `<strong>[${topico}]:</strong> ${mensagem}`;
   // Adiciona o novo <p> dentro da nossa messagesBox
   messagesBox.appendChild(novaMensagemElemento);
   // Faz a caixa "rolar" para baixo automaticamente para ver a última msg
   messagesBox.scrollTop = messagesBox.scrollHeight;
}
// --- Vamos testar nossas novas funções! ---
function handlePublish() {
   console.log("Botão clicado!");
   const topico = topicInput.value;
   const mensagem = messageInput.value;
   if (!topico || !mensagem) {
      alert("Por favor, preencha tanto o Tópico quanto a Mensagem.");
      return;
   }
   console.log(`Simulando publicação: Tópico='${topico}', Mensagem='${mensagem}'`);
   // *** AQUI A MUDANÇA ***
   // Simulação: Simule que publicamos e recebemos a msg de volta
   addMessageToBox(topico, mensagem);

   messageInput.value = "";
   messageInput.focus();
}
// 2. Defina um status inicial quando a página carregar
setStatus("Aguardando conexão...", "#ffa500"); // Laranja
```

Capítulo 3: O Sistema Nervoso - A Teoria do MQTT

Objetivo: Construir o modelo mental correto do que é o MQTT e como ele é fundamentalmente diferente do modelo de "Request/Response" (Requisição/Resposta) da web que você já conhece.

3.1 - O Problema: A Web Tradicional é um Jogo de "Pergunte e Responda"

Se quiséssemos buscar dados de um servidor, usaríamos um fetch(). Isso se chama Request/Response.

E se quiséssemos o oposto? E se quiséssemos que o servidor (ou outro usuário) pudesse nos enviar dados sem que nós tivéssemos perguntado primeiro? É para isso que serve o MQTT.

3.2 - A Solução MQTT: O Modelo "Publish/Subscribe" (Publicar/Assinar)

O MQTT (Message Queuing Telemetry Transport) é um protocolo de mensagens. Ele é o padrão da indústria para Internet das Coisas (IoT) porque é incrivelmente leve e eficiente. Ele usa um modelo de Assinatura e Publicação.

O Publisher escreve uma mensagem. Ele não sabe quem vai ler.
O Broker recebe essa mensagem matéria e a coloca em um local.
O Subscriber já disse ao broker: "Ei, me envie tudo sobre X assim que for publicado"
O Broker entrega a mensagem para todos os subscribers.

3.3 - Os 3 Componentes Essenciais do MQTT

O modelo MQTT (Pub/Sub) tem três partes. Você precisa entender o papel de cada uma:

- **O Broker**: É o servidor central. No nosso curso, usaremos o HiveMQ Cloud para ser o nosso broker. Ele é o ponto de encontro. Todo cliente se conecta a ele. Sua única função é receber mensagens e entregá-las a quem se interessar. Ele não armazena as mensagens (geralmente).
- **O Publisher**: É um cliente (nosso navegador, um sensor de temperatura, outro usuário) que envia uma mensagem. O Publisher envia e "esquece". Ele confia que o Broker fará a entrega. Para enviar, ele precisa de duas coisas:
    - O conteúdo da mensagem (ex: "Ligar luz").
    - O Tópico (o endereço) da mensagem (ex: casa/sala/luz).
- **O Subscriber**: É um cliente (nosso navegador, um app de celular) que recebe mensagens. Ele se conecta ao Broker e diz: "Por favor, me avise sempre que chegar uma mensagem no Tópico X". Ele fica "ouvindo" passivamente, e o Broker o notifica imediatamente quando uma nova mensagem é publicada naquele tópico.

O mais importante: No nosso app, seremos as duas coisas ao mesmo tempo. Quando publicamos uma mensagem, somos o Publisher. Quando recebemos mensagens de outros, somos o Subscriber.

3.4 - A Peça-Chave: Tópicos (Topics)

O Tópico é o "endereço" da mensagem no MQTT. É o que o Broker usa para saber para quem entregar. São apenas strings, mas são hierárquicas, como pastas no seu computador.

Exemplos:

- minha/casa/quarto/temperatura
- chat/sala_principal/bruno
- meu/teste/senai (o que usaremos no nosso projeto)

Quando publicamos, publicamos EM um tópico. Quando assinamos, assinamos UM tópico. Se o Cliente A publica em chat/sala1 e o Cliente B assina o tópico chat/sala2, eles nunca se comunicarão. Eles devem publicar e assinar o mesmo tópico para que a conversa aconteça.

3.5 - Como o JavaScript (Navegador) se encaixa nisso?

Aqui está um detalhe técnico crucial para a web. O protocolo MQTT "puro" roda sobre TCP/IP. Por motivos de segurança, o JavaScript em um navegador não pode abrir uma conexão TCP/IP bruta com um servidor.

A Solução: MQTT sobre WebSockets O navegador pode abrir uma conexão chamada WebSocket (WSS). Pense nisso como um "túnel" permanente e de mão dupla entre o navegador e um servidor.

O que faremos é: O JavaScript usa a biblioteca Paho (Capítulo 4). O Paho abre uma conexão WebSocket com o Broker HiveMQ (que deve ser configurado para aceitar WebSockets). O Paho então "fala" a língua MQTT por dentro desse túnel WebSocket. É por isso que as credenciais do nosso HiveMQ têm uma porta específica para "Websockets" (como 8884) e é essa que usaremos.

Capítulo 4: A Ponte - Paho MQTT Javascript

Objetivo deste Capítulo: Aprender a sintaxe da biblioteca Paho. Vamos ver quais são as funções e objetos que ela nos oferece para implementar o modelo Publish/Subscribe que aprendemos no Capítulo 3.

4.1 - O que é o Paho.js?

Pense no Paho.js como o "tradutor" ou o "manual de instruções". Ele é uma biblioteca JavaScript de código aberto da Eclipse Foundation que "fala" MQTT (empacotado dentro de WebSockets, como vimos).

Já a incluímos no nosso index.html com esta linha, que a carrega de um CDN (uma rede de distribuição de conteúdo): <script src="https://cdnjs.cloudflare.com/ajax/libs/paho-mqtt/1.0.1/mqttws31.min.js"></script>

4.2 - A Receita: Os 5 Passos para usar Paho

Usar o Paho sempre seguirá 5 passos.

Passo 1 - Configurar Credenciais e Cliente: Primeiro, precisamos das informações do nosso Broker (o HiveMQ) e de um ID de Cliente único.

```
// 1. Informações do Broker (Vindas do seu painel HiveMQ)
const BROKER_HOST = 'seu-cluster.s1.eu.hivemq.cloud';
const BROKER_PORT = 8884; // Lembre-se: Porta WEBSOCKET
const BROKER_USER = 'seu-usuario';
const BROKER_PASS = 'sua-senha';

// 2. Um ID de Cliente ÚNICO para esta conexão
// Se dois clientes com o MESMO ID se conectarem, o broker derruba o mais antigo!
const CLIENT_ID = "meuAppWeb_" + parseInt(Math.random() * 1000);

// 3. Criar a instância do Cliente
// Paho.MQTT.Client(host, port, client_id)
const client = new Paho.MQTT.Client(BROKER_HOST, BROKER_PORT, CLIENT_ID);
```

Passo 2 - Definir os Callbacks Principais: Informamos ao Paho quais das nossas funções ele deve chamar quando eventos importantes acontecerem.

```
// O QUE FAZER SE... a conexão for perdida
client.onConnectionLost = minhaFuncaoDeConexaoPerdida;
// O QUE FAZER SE... uma nova mensagem chegar (de um tópico que assinamos)
client.onMessageArrived = minhaFuncaoDeMensagemRecebida;
// --- E agora, definimos essas funções ---
function minhaFuncaoDeConexaoPerdida(responseObject) {
   if (responseObject.errorCode !== 0) {
      console.log("Conexão perdida: " + responseObject.errorMessage);
      // Aqui nós atualizaríamos nosso status na tela (Cap. 5)
   }
}
function minhaFuncaoDeMensagemRecebida(message) {
   // O 'message' é um objeto com toda a informação
   const topico = message.destinationName;
   const payload = message.payloadString; // A mensagem em si!
   console.log(`Mensagem recebida do tópico ${topico}: ${payload}`);
   // Aqui nós adicionaríamos a mensagem na nossa caixa (Cap. 5)
}
```

Passo 3 - Conectar-se ao Broker: Agora, tentamos de fato conectar. A função connect também é assíncrona, por isso ela também precisa de callbacks: onSuccess e onFailure.

```
// 1. Criar o objeto de opções de conexão
const connectOptions = {
   useSSL: true,       // Obrigatório para o HiveMQ Cloud (conexão segura)
   userName: BROKER_USER,
   password: BROKER_PASS,
   onSuccess: minhaFuncaoDeSucessoNaConexao, // Callback de sucesso
   onFailure: minhaFuncaoDeFalhaNaConexao,   // Callback de falha
   // Outras opções úteis:
   timeout: 3,         // Tempo em segundos para desistir
   cleanSession: true  // Começa uma sessão limpa, não armazena msgs offline
};
// 2. Tentar a conexão
console.log("Tentando conectar ao broker...");
client.connect(connectOptions);

// --- E definimos os callbacks de conexão ---
```

```
function minhaFuncaoDeSucessoNaConexao() {
   console.log("CONECTADO COM SUCESSO!");
   // Este é o lugar perfeito para ATUALIZAR O STATUS (Cap. 5)
   // E... para ASSINAR NOSSOS TÓPICOS! (Passo 4)
}
function minhaFuncaoDeFalhaNaConexao(responseObject) {
   console.log("FALHA AO CONECTAR: " + responseObject.errorMessage);
   // Aqui atualizaríamos o status para "Falha" (Cap. 5)
}
```

Passo 4 - Assinar Tópicos (Subscribe): Importante: Você só pode assinar um tópico depois que o onSuccess (sucesso na conexão) for disparado.

A chamada é simples: client.subscribe(topicName);

Vamos colocar isso dentro da nossa função de sucesso:

```
function minhaFuncaoDeSucessoNaConexao() {
   console.log("CONECTADO COM SUCESSO!");
   // O tópico que queremos ouvir
   const topicoParaAssinar = "meu/teste/senai";
   // Assina o tópico
   client.subscribe(topicoParaAssinar);
   console.log(`Assinatura enviada para o tópico: ${topicoParaAssinar}`);
}
```

A partir de agora, qualquer mensagem publicada no tópico meu/teste/senai por qualquer cliente no mundo, fará o broker disparar nossa função minhaFuncaoDeMensagemRecebida.

Passo 5 - Publicar Mensagens (Publish): Finalmente, o que fazer quando queremos enviar uma mensagem? Isso pode acontecer a qualquer momento depois que estamos conectados (por exemplo, no clique do nosso botão).

Publicar exige duas etapas:

- Criar um objeto Paho.MQTT.Message.
- Enviar (publicar) esse objeto com client.send().

```
function publicarUmaMensagem(topico, mensagem) {
   if (!client.isConnected()) {
      console.log("Não conectado. Não é possível publicar.");
      return;
   }
   // 1. Criar o objeto da mensagem
   // Paho.MQTT.Message(payload_da_mensagem)
   const pahoMessage = new Paho.MQTT.Message(mensagem);
   // 2. Definir o tópico de destino
   pahoMessage.destinationName = topico;
   // 3. Enviar a mensagem
   client.send(pahoMessage);
   console.log(`Mensagem enviada para o tópico ${topico}: ${mensagem}`);
}
// Exemplo de como usaríamos:
// publicarUmaMensagem("meu/teste/bruno", "Olá mundo do MQTT!");
```

Capítulo 5: A Aplicação Final - Criando o Painel MQTT Funcional

Objetivo deste Capítulo: Integrar o Paho MQTT ao nosso app.js para criar uma aplicação web totalmente funcional que publica e recebe mensagens em tempo real.

5.1 - Preparação: Suas Credenciais

Antes de tudo, tenha em mãos os dados do seu cluster gratuito do HiveMQ Cloud. Você vai precisar de:

- Host: (algo como seu-cluster.s1.eu.hivemq.cloud)
- Porta (Websockets): 8884 (para conexões seguras SSL/TLS)
- Username: (o usuário que você criou)
- Password: (a senha que você criou)

Vamos pegar o app.js que começámos no Capítulo 2 e modificá-lo drasticamente.

5.2 - O app.js Completo (Versão Final)

Abaixo está o código completo. Substitua todo o conteúdo do seu app.js por este.

app.js (Versão Final)

```
// --- 1. Seleção de Elementos do DOM (Do Cap. 2) ---
// (Não mudou nada aqui)
const topicInput = document.getElementById('topic');
const messageInput = document.getElementById('message');
const publishButton = document.getElementById('btn-publicar');
const statusText = document.getElementById('status-text');
const messagesBox = document.getElementById('messages-box');
const placeholder = document.querySelector('.mensagem-placeholder');

// --- 2. Configuração do Broker MQTT (Do Cap. 4) ---
// !!! PREENCHA COM SEUS DADOS DO HIVEMQ !!!
const BROKER_HOST = 'SEU_HOST.s1.eu.hivemq.cloud';
const BROKER_PORT = 8884; // Porta Websocket SSL
const BROKER_USER = 'SEU_USUARIO';
const BROKER_PASS = 'SUA_SENHA';

// Gera um ID de cliente único para cada sessão
const CLIENT_ID = "meuAppWeb_" + parseInt(Math.random() * 1000);

// --- 3. Funções Auxiliares de UI (Do Cap. 2) ---
/**
 * Atualiza o texto e a cor do status da conexão.
 */
function setStatus(texto, cor) {
   statusText.textContent = texto;
   statusText.style.color = cor;
}

/**
 * Adiciona uma nova linha de mensagem na caixa de mensagens.
 */
function addMessageToBox(topico, mensagem) {
   if (placeholder) {
      placeholder.style.display = 'none'; // Esconde o placeholder
   }
   const novaMensagemElemento = document.createElement('p');
   novaMensagemElemento.innerHTML = `<strong>[${topico}]:</strong> ${mensagem}`;
   messagesBox.appendChild(novaMensagemElemento);
   messagesBox.scrollTop = messagesBox.scrollHeight;
}

// --- 4. Lógica do Cliente MQTT (Do Cap. 4) ---
// Cria a instância do Cliente Paho
const client = new Paho.MQTT.Client(BROKER_HOST, BROKER_PORT, CLIENT_ID);

// Define os Callbacks do Paho
client.onConnectionLost = onConnectionLost;
client.onMessageArrived = onMessageArrived;

// Opções de Conexão
const connectOptions = {
   useSSL: true,
   userName: BROKER_USER,
   password: BROKER_PASS,
   onSuccess: onConnect, // Função que será chamada no sucesso
   onFailure: onFailure, // Função que será chamada na falha
```

```
   timeout: 3
};

// --- Funções de Callback do MQTT (O Coração da Lógica) ---
/**
 * Chamada quando a conexão é bem-sucedida (callback 'onSuccess')
 */
function onConnect() {
   console.log("Conectado ao broker MQTT com sucesso!");
   // ATUALIZA A UI (Cap. 2)
   setStatus("Conectado", "#28a745"); // Verde
   // ASSINA O TÓPICO (Cap. 4)
   // Vamos assinar o tópico que já está no campo de input
   const topicoPadrao = topicInput.value;
   client.subscribe(topicoPadrao);
   console.log(`Assinatura enviada para: ${topicoPadrao}`);
}

/**
 * Chamada quando a conexão falha (callback 'onFailure')
 */
function onFailure(responseObject) {
   console.log("Falha ao conectar: " + responseObject.errorMessage);
   // ATUALIZA A UI (Cap. 2)
   setStatus(`Falha: ${responseObject.errorMessage}`, "#d9534f"); // Vermelho
}

/**
 * Chamada quando a conexão é perdida (callback 'onConnectionLost')
 */
function onConnectionLost(responseObject) {
   if (responseObject.errorCode !== 0) {
      console.log("Conexão perdida: " + responseObject.errorMessage);
      // ATUALIZA A UI (Cap. 2)
      setStatus("Conexão Perdida!", "#d9534f"); // Vermelho
   }
}

/**
 * Chamada quando uma mensagem chega (callback 'onMessageArrived')
 */
function onMessageArrived(message) {
   const topico = message.destinationName;
   const payload = message.payloadString;
   console.log(`Mensagem recebida! Tópico: ${topico}, Payload: ${payload}`);
   // ATUALIZA A UI (Cap. 2)
   addMessageToBox(topico, payload);
}

// --- 5. Lógica de Publicação (Unindo Cap. 2 e Cap. 4) ---
// Adiciona o ouvidor de evento (Do Cap. 2)
publishButton.addEventListener('click', handlePublish);

/**
 * Função chamada quando o botão "Publicar" é clicado.
 */
function handlePublish() {
   // 1. Pega os valores da UI (Cap. 2)
   const topico = topicInput.value;
   const mensagem = messageInput.value;
   // 2. Valida os dados (Cap. 2)
   if (!topico || !mensagem) {
      alert("Por favor, preencha o Tópico e a Mensagem.");
      return;
   }
   // 3. Verifica se estamos conectados (Lógica do Paho)
   if (!client.isConnected()) {
      alert("Não estamos conectados ao broker.");
      return;
   }
   // 4. Publica a mensagem usando Paho (Cap. 4)
   console.log(`Publicando no tópico '${topico}': '${mensagem}'`);
```

```
   // Cria o objeto da mensagem
   const pahoMessage = new Paho.MQTT.Message(mensagem);
   pahoMessage.destinationName = topico;
   // Envia (Publica)
   client.send(pahoMessage);
   // 5. Limpa o input (Cap. 2)
   messageInput.value = "";
   messageInput.focus();
}

// --- 6. Inicia a Conexão! ---
// Esta é a linha que efetivamente liga a aplicação.
// Colocamos o status inicial e mandamos o Paho conectar.
setStatus("Conectando...", "#ffa500"); // Laranja
client.connect(connectOptions); // Tenta conectar
```

5.3 - O Teste Final (A Hora da Verdade)

Você terminou. Para ver a mágica, faça o seguinte:

Salve seu arquivo app.js com o código acima (e com as suas credenciais do HiveMQ).
Abra o index.html no seu navegador (ex: Chrome).
Abra o index.html em uma segunda aba ou segunda janela do navegador.
Coloque as duas janelas lado a lado.

Observe: Em ambas as janelas, o status deve mudar de "Conectando..." (laranja) para "Conectado" (verde). Isso prova que o callback onSuccess funcionou.

Na Janela 1, digite "Olá" no campo de mensagem e clique em "Publicar".

Instantaneamente, a mensagem [meu/teste/senai]: Olá deve aparecer na caixa de "Mensagens Recebidas" da Janela 2 (e também da Janela 1, já que ambas estão assinando o mesmo tópico).

Por que isso acontece?
Janela 1 (Publisher): handlePublish() -> client.send() -> Mensagem vai para o Broker HiveMQ.

HiveMQ (Broker): "Recebi uma mensagem em meu/teste/bruno. Quem está assinando isso?"

HiveMQ (Broker): "Ah, a Janela 1 e a Janela 2 estão. Vou enviar para elas."

Janela 1 e 2 (Subscriber): Recebem a mensagem -> onMessageArrived() é disparado -> addMessageToBox() é chamado -> A UI é atualizada.