**Geração e Validação**

A ideia de um sistema de licenciamento offline simples é criar uma "fechadura" (o algoritmo de validação) que só pode ser aberta por "chaves" (as licenças) que você, o desenvolvedor, cria com uma "chave mestra" (um segredo).

Parte 1: A Estrutura da Chave de Licença:

Nossa chave tem o formato EF25-XXXX-YYYY-ZZZZ. Vamos quebrar:

1. EF25- (Prefixo): Um identificador simples para seu produto ("Escala Fácil 2025"). É fácil de checar e validar.
2. XXXX-YYYY (O Corpo ou *Payload*): Esta é a parte única de cada chave. Ela é gerada aleatoriamente. Pense nela como o número de série de um produto.
3. ZZZZ (O *Checksum* ou Assinatura): Esta é a parte mais importante. Ela não é aleatória. Ela é um código calculado a partir do "Corpo" da chave e da sua "Chave Mestra Secreta".

Como a Geração Funciona (O que você faz):

1. Você gera um "Corpo" aleatório (ex: A1B2-C3D4).
2. Você combina esse corpo com sua Chave Mestra Secreta (ex: A1B2-C3D4 + EscalaFacil-2025-SegredoParaValidacaoUnica).
3. Você passa essa combinação por um "liquidificador" digital chamado algoritmo de hash (no nosso caso, SHA-1). Ele gera um código longo e complexo que é impossível de reverter (você não pode "des-liquidificar" para descobrir o segredo).
4. Você pega um pedaço pequeno desse código hash (ex: os 4 primeiros caracteres) para ser o seu Checksum.
5. Você junta tudo: EF25-A1B2-C3D4-[Checksum]. Essa é a chave que você vende.

Como a Validação Funciona (O que o aplicativo faz):

1. O cliente insere a chave completa no seu aplicativo.
2. O aplicativo ignora o checksum por um momento e pega apenas o "Corpo" (A1B2-C3D4).
3. Ele faz exatamente o mesmo cálculo que você fez: combina esse corpo com a mesma Chave Mestra Secreta que está escondida de forma segura dentro do código do aplicativo (no backend Rust).
4. Ele passa a combinação pelo mesmo "liquidificador" (hash SHA-1).
5. Ele pega os 4 primeiros caracteres do hash que ele mesmo acabou de calcular.
6. O teste final: Ele compara o checksum que *ele calculou* com o checksum que veio na chave do cliente. Se forem idênticos, a chave é autêntica! Se não, é uma chave falsa.

Parte 2: Como Foi Feito no Seu Projeto

A implementação foi dividida em três componentes lógicos:

1. O Gerador de Chaves (Sua Ferramenta Pessoal):
   * Arquivo: gerador\_de\_chaves.html.
   * Como foi feito: Este é um arquivo HTML simples que você abre no seu próprio navegador. Ele contém a lógica em JavaScript para gerar chaves válidas. A constante SEGREDO neste arquivo é a sua "Chave Mestra". Ele executa exatamente os passos de "Como a Geração Funciona" descritos acima.
2. O Validador (O Cofre Seguro no Tauri):
   * Arquivo: src/main.rs (o código do backend Rust).
   * Como foi feito: A função validate\_license\_key implementa a "fechadura". Ela contém a mesma constante SECRET\_KEY e executa os mesmos passos de validação descritos acima. Por estar em Rust (código compilado), é extremamente difícil para um usuário mal-intencionado descobrir qual é o seu segredo. Além disso, as funções check\_activation\_status e save\_activation\_status foram criadas para ler e escrever em um arquivo simples no computador do usuário, para que o aplicativo "lembre" que já foi ativado.
3. A Integração com o Aplicativo (A Tela de Ativação):
   * Arquivo: A-principal.js.
   * Como foi feito: A lógica de inicialização do aplicativo foi alterada. Agora, a primeira coisa que ele faz é chamar a função checkActivation().
     + Se checkActivation() retornar true (porque encontrou o arquivo de licença salvo), ele inicia o aplicativo normalmente chamando initMainApp().
     + Se retornar false, ele chama showActivationModal(), que exibe a tela para o usuário inserir a chave. O botão "Ativar" nesta tela chama a função validateLicenseKey no backend Rust para fazer a verificação.

**O Prompt Completo**

Assunto: Implementar um Sistema de Licenciamento Offline para um Aplicativo Tauri

Visão Geral:

O objetivo é criar um sistema de licenciamento offline para um aplicativo de desktop Tauri. O sistema deve usar um algoritmo de chave com *payload* e *checksum*, onde a validação ocorre no backend Rust para segurança. O aplicativo deve verificar o status da licença na inicialização e, se não estiver ativado, exibir um modal para que o usuário insira a chave. Adicionalmente, deve ser criada uma ferramenta separada (um arquivo HTML) para o desenvolvedor gerar chaves válidas.

Arquitetura Alvo:

1. Backend Rust (em src/main.rs):
   * Conterá a lógica de validação da chave, usando um segredo compartilhado.
   * Conterá funções para ler e escrever o status de ativação em um arquivo local no sistema de arquivos do usuário.
2. Frontend JavaScript (em A-principal.js e index.html):
   * O fluxo de inicialização será modificado para verificar o status de ativação antes de carregar a interface principal.
   * Um modal de ativação será criado no HTML e controlado via JavaScript para coletar a chave do usuário.
3. Gerador de Chaves (arquivo gerador\_de\_chaves.html):
   * Uma ferramenta de desenvolvedor, autônoma e externa ao projeto final, para gerar as chaves de licença a serem distribuídas.

Instruções Técnicas Detalhadas:

Passo 1: Criar a Ferramenta Geradora de Chaves

1. Crie um novo arquivo autônomo chamado gerador\_de\_chaves.html.
2. Lógica da Chave: A chave deve ter o formato EF25-XXXX-YYYY-ZZZZ.
   * XXXX-YYYY (Payload) deve ser gerado aleatoriamente.
   * ZZZZ (Checksum) deve ser os primeiros 4 caracteres de um hash SHA-1 da string payload + "-" + SEGREDO.
3. Implementação:
   * Crie uma constante SEGREDO (ex: "EscalaFacil-2025-SegredoParaValidacaoUnica").
   * Use a API crypto.subtle.digest('SHA-1', ...) do JavaScript para calcular o hash.
   * Crie uma interface simples com um botão "Gerar Chave" que exibe a chave gerada na tela.

Passo 2: Implementar o Backend Rust em src/main.rs

1. Adicionar Dependências: Adicione as crates sha1 e hex ao arquivo Cargo.toml.
2. Definir o Segredo: Crie uma constante SECRET\_KEY em Rust com o mesmo valor exato do SEGREDO do gerador de chaves.
3. Criar o Comando validate\_license\_key:
   * Esta função #[tauri::command] deve receber uma String (a chave).
   * Ela deve desmontar a chave, extrair o payload e o checksum.
   * Deve recalcular o checksum usando o payload e a SECRET\_KEY.
   * Deve comparar o checksum calculado com o da chave e retornar true ou false.
4. Criar Comandos de Status de Ativação:
   * Crie uma função #[tauri::command] fn check\_activation\_status(app\_handle: tauri::AppHandle) -> bool. Ela deve ler um arquivo de um local de dados padrão (usando app\_handle.path\_resolver().app\_data\_dir()) e retornar true se o conteúdo for "activated".
   * Crie uma função #[tauri::command] fn save\_activation\_status(app\_handle: tauri::AppHandle, activated: bool). Ela deve escrever "activated" ou "not\_activated" no mesmo arquivo.
5. Registrar os Comandos: Adicione todos os novos comandos ao invoke\_handler no tauri::Builder.

Passo 3: Integrar o Sistema de Licença no Frontend

1. Modificar index.html:
   * Adicione a estrutura HTML para um modal de ativação (activationModalBackdrop). Ele deve conter um campo de input para a chave, um botão "Ativar" e uma área para mensagens de erro. Ele deve estar oculto (hidden) por padrão.
2. Modificar A-principal.js:
   * Criar showActivationModal(): Esta função deve tornar o modal visível e configurar o onclick do botão "Ativar". O clique deve pegar o valor do input, chamar window.\_\_TAURI\_\_.invoke('validate\_license\_key', ...) e, se for válido, chamar window.\_\_TAURI\_\_.invoke('save\_activation\_status', ...) e iniciar o app. Se for inválido, deve exibir uma mensagem de erro.
   * Criar checkActivation(): Esta função deve chamar window.\_\_TAURI\_\_.invoke('check\_activation\_status') e, com base no resultado, chamar initMainApp() (se true) ou showActivationModal() (se false).
   * Modificar init(): A função init() existente deve ser substituída para chamar apenas checkActivation() no início.