Excelente pergunta! Entender esse "cronograma" é a chave para dominar a arquitetura que construímos. É uma "dança" perfeitamente coreografada entre o frontend e o backend, onde cada peça tem um papel muito específico.

Vamos seguir a jornada de um clique, passo a passo, desde o momento em que o usuário digita a senha até ver a tela do dashboard.

**O Cronograma do Login: Uma Viagem em 13 Passos**

**Personagens:**

* **Frontend (Next.js):** O "Garçom" que anota o pedido e interage com o cliente.
* **Backend (NestJS):** A "Cozinha" que prepara o pedido e tem as regras de negócio.
* **Você:** O "Cliente" no restaurante.

**Ato I: O Pedido (Frontend)**

**Passo 1: O Clique e a Validação do Formulário**

* **Onde:** login/page.tsx
* **O que acontece:** Você clica no botão "Entrar". O react-hook-form intercepta o evento e, usando o zodResolver, valida os dados do formulário contra o formSchema que definimos. Se o e-mail for inválido ou a senha for curta, a função onSubmit **nem é chamada**, e o Zod exibe a mensagem de erro no campo correspondente.

**Passo 2: O Hook Entra em Ação**

* **Onde:** useLogin.ts
* **O que acontece:** Com os dados validados, a função onSubmit é chamada, que por sua vez executa login(data). Isso aciona a mutationFn do nosso hook useLogin, que foi criado com o **TanStack Query**. O estado isPending do hook se torna true (o que faz o texto do botão mudar para "Entrando...").

**Passo 3: A Viagem pela Rede**

* **Onde:** useLogin.ts
* **O que acontece:** A função fetch é executada, enviando uma requisição POST para a URL do nosso backend (http://localhost:3001/api/auth/login), com o e-mail e a senha no corpo (body) em formato JSON. O frontend agora fica aguardando a resposta.

**Ato II: A Validação (Backend)**

**Passo 4: A Chegada na API**

* **Onde:** main.ts e Roteador do NestJS
* **O que acontece:** O servidor NestJS recebe a requisição. Ele aplica o prefixo global /api e procura um Controller que corresponda ao caminho /auth/login. Ele encontra nosso AuthController.

**Passo 5: O Segurança na Porta (Guard)**

* **Onde:** auth.controller.ts e local-auth.guard.ts
* **O que acontece:** Antes de executar o método login do controller, o NestJS vê o decorator @UseGuards(LocalAuthGuard). Este **Guard** intercepta a requisição. A função dele é ser o "segurança" da rota.

**Passo 6: A Verificação das Credenciais (Strategy)**

* **Onde:** local.strategy.ts e auth.service.ts
* **O que acontece:** O LocalAuthGuard aciona a LocalStrategy do Passport. A estratégia extrai o e-mail e a senha da requisição e chama o método this.authService.validateUser(email, password). É neste momento que o AuthService conversa com o UsersService para buscar o usuário no banco de dados e usa o argon2.verify para comparar a senha enviada com o hash salvo no banco. Se a senha estiver errada ou o usuário não existir, uma UnauthorizedException (erro 401) é lançada, e o processo para aqui.

**Passo 7: O Controller Recebe o Usuário Validado**

* **Onde:** auth.controller.ts
* **O que acontece:** Se a validação no passo anterior for um sucesso, a LocalStrategy retorna o objeto user completo. O Guard então anexa esse objeto à requisição. O decorator @GetUser() que criamos extrai esse objeto user e o passa como um argumento para o nosso método login do controller.

**Passo 8: A Criação do "Crachá" (JWT)**

* **Onde:** auth.service.ts
* **O que acontece:** O controller chama this.authService.login(user). Dentro do serviço, o **JwtService** é usado para criar o accessToken e o refreshToken. O **JWT** é gerado aqui, contendo o ID do usuário (sub) e outras informações no seu payload. O hash do refresh token é salvo no banco.

**Passo 9: A Resposta é Enviada**

* **Onde:** auth.controller.ts
* **O que acontece:** O AuthService retorna o objeto { user, accessToken, refreshToken }. O controller recebe isso, coloca o refreshToken em um cookie httpOnly e envia o { user, accessToken } de volta para o frontend com um status **200 OK**.

**Ato III: A Celebração (Frontend)**

**Passo 10: O Sucesso da Requisição**

* **Onde:** useLogin.ts
* **O que acontece:** A fetch no frontend recebe a resposta 200 OK. A promessa é resolvida, e o TanStack Query executa a função onSuccess do nosso useMutation.

**Passo 11: Quem Mantém o Estado? Zustand**

* **Onde:** useLogin.ts e authStore.ts
* **O que acontece:** A onSuccess chama setToken(data.accessToken) e setUser(data.user). Isso atualiza nosso "cofre" de estado global, o **Zustand**. Como nosso store usa o middleware persist, o Zustand automaticamente salva essa informação no localStorage do navegador, garantindo que você continue logado mesmo se fechar a aba.

**Passo 12: A Navegação para a Área Protegida**

* **Onde:** useLogin.ts
* **O que acontece:** A última linha do onSuccess, window.location.href = '/dashboard', é executada. O navegador é instruído a carregar a página do dashboard.

**Passo 13: O Porteiro do Dashboard (AuthGuard)**

* **Onde:** dashboard/page.tsx e AuthGuard.tsx
* **O que acontece:** A página /dashboard começa a carregar. A primeira coisa que ela renderiza é o nosso componente **AuthGuard**.
  + O AuthGuard olha para o **Zustand store**.
  + Graças ao recarregamento da página, o Zustand já teve tempo de se "hidratar" (ler os dados) do localStorage.
  + O AuthGuard vê que o token existe.
  + Ele então permite que seus {children} (o DashboardContent) sejam renderizados.
  + **Resultado Final:** Você vê a página do dashboard com a mensagem "Bem-vindo!".

**Resumo das Funções**

| Ferramenta | Papel no Login |
| --- | --- |
| **DTO** | Define a "forma" dos dados de login (email, password) para validação. |
| **Hook (useLogin)** | Orquestra a chamada de API e a reação ao sucesso/erro no frontend. |
| **Guard** | É o "segurança" do NestJS que intercepta a requisição e aciona a validação. |
| **Strategy** | Contém a lógica específica de como validar as credenciais (neste caso, email/senha). |
| **JWT** | É o "crachá de acesso" seguro que o backend gera para o frontend. |
| **Zustand** | É o "cofre" do frontend que guarda o crachá (token) para ser usado em futuras requisições. |
| **AuthGuard** | É o "porteiro" do frontend que verifica se o crachá está no cofre antes de dar acesso a uma página. |