Termfy

TERMFY

Guia do framework

Um binário em Bun, três modos: TUI, CLI e servidor MCP

Starter kit, registro de comandos, saída em texto e JSON, códigos de saída, cofre cifrado, design modern unix e as skills que orientam agentes.

Edição v0.15.0
Gerado em 16 de novembro de 2026
Um projeto da Promovaweb
Fonte editorial: docs/user/

Sumário

Termfy 10
Como ler este guia 10
Repositórios 12
Visão geral 13
As peças 13
Escolhas que o framework já fez 13
Público do guia 14
Começar 15
Antes de começar 15
1. Crie o repositório pelo template 15
2. Peça o setup ao agente 15
3. Confira os três modos 17
O que mudou no projeto 22
Manter as skills atualizadas 22
Se algo der errado 23
Próximo passo 23
Três modos 24
Despacho dos três modos 24
O que os modos compartilham 24
Quando usar cada modo 24
Documentação embutida 25
Skills 26
As dezoito skills 26
O CLISPEC.md e as specs 27
A sequência 28
As referências 28
Como chegam ao projeto 29
Compatibilidade 29
Escrever uma skill nova 29
Sequência de uso das skills 30
Visão geral 30
0. Entrar: `termfy` 30
1. Criar: `termfy-init` 31

2. Apresentar: `termfy-brand` ... 31
3. Descrever o binário: `termfy-clispec` ... 31
4. Especificar cada evolução: `termfy-spec` ou `termfy-api-import` ... 31
5. Implementar, uma entrega por vez ... 32
6. Fechar a spec: `termfy-review`, `termfy-docs` e `termfy-spec` ... 32
7. Publicar e ligar aos agentes: `termfy-release` e `termfy-mcp-client` ... 32
Quando sair da ordem ... 33
O que cada skill não faz ... 33
Descrever e especificar ... 34
1. A entrevista do CLISPEC ... 34
2. O CLISPEC.md gravado ... 36
3. A entrevista da spec ... 39
4. A spec gravada ... 40
5. O ciclo de uma spec ... 42
6. A conferência manual ... 43
Próximo passo ... 43
Integrar uma API ... 44
O contrato da uazapi ... 44
1. Importar a documentação ... 45
2. URL e token ... 46
3. O client HTTP ... 47
4. Commands com a instância por parâmetro ... 49
5. Tools e tela ... 51
6. Testar sem a API real ... 52
Próximo passo ... 53
Guardar dados ... 54
Quando pedir um banco ... 54
1. A spec com a tabela ... 54
2. Peça a implementação ... 55
3. A migration ... 55
4. O repositório ... 56
5. O command usa o repositório ... 56
6. Testes em memória ... 57
Próximo passo ... 58

Testar, revisar e publicar ... 59
1. Testes nos três modos ... 59
2. Revisão por gravidade ... 61
3. Fechar a spec ... 62
4. Publicar ... 62
5. Conectar o agente ... 62
Próximo passo ... 63
CLI ... 64
Registro declarativo ... 64
Argumentos e opções ... 65
O contrato de `run` ... 65
Saída em texto e em JSON ... 66
Códigos de saída ... 66
Commands de apoio ... 67
OpenCLI ... 67
Adicionar um comando ... 68
TUI ... 69
Estrutura da tela ... 69
Teclas ... 69
Paleta de comandos ... 70
Barra de status e toasts ... 70
Diálogos ... 70
Mouse ... 71
Layout responsivo ... 71
Arquivos e pausa ... 72
Ícones ... 72
Design modern unix ... 72
Componentes ... 73
Criar uma aba ... 73
Testar sem terminal ... 74
MCP ... 76
O servidor embutido ... 76
O que o starter kit expõe ... 76
Regras para tools ... 77
Economia de tokens ... 77

Adicionar uma tool 77
Configuração e credenciais 79
settings.json 79
Editar pela TUI 80
credentials.json 80
Chave mestra 81
O que a cifra protege 81
Usar uma credencial no código 81
Publicar 82
O binário derivado 82
Build local 82
Instalar o binário 82
O framework 82
No Hub da Promovaweb 83
Glossário e problemas comuns 84
Glossário 84
Problemas comuns 86
termfy 89
Quando usar 89
Como pedir 89
O que a skill faz 90
Exemplo 90
Regras que a skill segue 92
O que ela entrega 92
Próxima skill 93
termfy-init 94
Quando usar 94
Como pedir 94
O que a skill faz 95
Exemplo 95
Regras que a skill segue 97
O que ela entrega 97
Próxima skill 98
termfy-brand 99
Quando usar 99

Como pedir 99
O que a skill faz 100
Exemplo 100
Regras que a skill segue 101
O que ela entrega 102
Próxima skill 103
termfy-clispec 104
Quando usar 104
Como pedir 104
O que a skill faz 105
Exemplo 105
Regras que a skill segue 107
O que ela entrega 108
Próxima skill 108
termfy-spec 109
Quando usar 109
Como pedir 109
O que a skill faz 110
Exemplo 110
Regras que a skill segue 112
O que ela entrega 113
Próxima skill 113
termfy-api-import 114
Quando usar 114
Como pedir 114
O que a skill faz 115
Exemplo 115
Regras que a skill segue 117
O que ela entrega 118
Próxima skill 118
termfy-cli 119
Quando usar 119
Como pedir 119
O que a skill faz 119
Exemplo 120

Regras que a skill segue ... 122
O que ela entrega ... 122
Próxima skill ... 123
termfy-tui ... 124
Quando usar ... 124
Como pedir ... 124
O que a skill faz ... 125
Padrões de tela e checklist ... 125
Exemplo ... 126
Regras que a skill segue ... 129
O que ela entrega ... 130
Próxima skill ... 131
termfy-mcp ... 132
Quando usar ... 132
Como pedir ... 132
O que a skill faz ... 132
Exemplo ... 133
Regras que a skill segue ... 136
O que ela entrega ... 136
Próxima skill ... 137
termfy-config ... 138
Quando usar ... 138
Como pedir ... 138
O que a skill faz ... 138
Exemplo ... 139
Regras que a skill segue ... 140
O que ela entrega ... 141
Próxima skill ... 142
termfy-api ... 143
Quando usar ... 143
Como pedir ... 143
O que a skill faz ... 143
Exemplo ... 144
Regras que a skill segue ... 147
O que ela entrega ... 148

Próxima skill 148
termfy-database 149
Quando usar 149
Como pedir 149
O que a skill faz 150
Exemplo 150
Regras que a skill segue 152
O que ela entrega 153
Próxima skill 154
termfy-command-set 155
Quando usar 155
Como pedir 155
O que a skill faz 155
Exemplo 156
Regras que a skill segue 159
O que ela entrega 160
Próxima skill 160
termfy-test 161
Quando usar 161
Como pedir 161
O que a skill faz 162
Exemplo 162
Regras que a skill segue 164
O que ela entrega 164
Próxima skill 165
termfy-review 166
Quando usar 166
Como pedir 166
O que a skill faz 166
Exemplo 167
Regras que a skill segue 168
O que ela entrega 169
Próxima skill 169
termfy-docs 170
Quando usar 170

Como pedir 170
O que a skill faz 170
Exemplo 171
Regras que a skill segue 172
O que ela entrega 172
Próxima skill 173
termfy-release 174
Quando usar 174
Como pedir 174
O que a skill faz 174
Exemplo 175
Regras que a skill segue 176
O que ela entrega 177
Próxima skill 178
termfy-mcp-client 179
Quando usar 179
Como pedir 179
O que a skill faz 180
Exemplo 180
Regras que a skill segue 181
O que ela entrega 182
Próxima skill 182

Termfy

Este guia ensina você a construir uma ferramenta de terminal com o Termfy, o framework da Promovaweb para binários em Bun. Cada binário é um único executável com três modos: você navega pela TUI usando o teclado, executa comandos pela CLI e conecta agentes de IA pelo servidor MCP. A CLI também atende scripts e agentes com saída estruturada. Você descreve ao agente o que precisa construir e as 18 skills do Termfy conduzem cada etapa, da entrevista inicial à release.

Os exemplos do guia usam o `mycli`, um binário que consulta os pedidos de uma loja. O capítulo de integração usa a API de WhatsApp da uazapi. As saídas do esqueleto vêm do starter kit em execução. Na Parte 2, as saídas dos commands novos mostram o comportamento que a spec descreve.

Como ler este guia

O guia tem três partes. Se é a sua primeira vez com o Termfy, leia a Parte 1 e siga a Parte 2 na ordem. A Parte 3 é para consulta.

Parte 1. Fundamentos

O que o Termfy é, como um executável atende três públicos e como as skills orientam o agente.

Capítulo	Pergunta que responde
Visão geral	De que peças o Termfy é feito?
Começar	Como criar um binário e ver os três modos funcionando?
Três modos	Como um só executável decide entre TUI, CLI e MCP?
Skills	Como as skills orientam um agente e como são atualizadas?
Sequência das skills	Qual é a ordem das skills e o que cada uma entrega à seguinte?

Parte 2. Construir

A Parte 2 acompanha uma evolução desde a descrição ao agente até a publicação do binário, mostrando os documentos e o código de cada etapa.

Capítulo	O que você aprende
Descrever e especificar	a entrevista do `CLISPEC.md`, a spec de uma evolução e o ciclo dos status
Integrar uma API	importar a API da uazapi, guardar o token, escrever o client e passar a instância por parâmetro
Guardar dados	quando pedir um banco SQLite, migrations embutidas e repositórios
Testar, revisar e publicar	testes nos três modos, revisão por gravidade, release e conexão do agente

Parte 3. Referência

Página	Conteúdo
CLI	registry, options globais, saída em texto e `--json`, exit codes
TUI	abas, teclas, tema e componentes
MCP	tools, resources, prompts e regras de segurança
Configuração e credenciais	`settings.json`, cofre cifrado e master key
Publicar	versão, changelog, tag e binários
Glossário e problemas comuns	termos do framework e soluções para os erros frequentes

Uma página por skill

Cada página diz quando usar a skill, como pedir, o que ela faz, um exemplo do que ela produz, os contratos que segue e para onde o fluxo vai depois.

Etapa	Skills
começar	termfy, termfy-init, termfy-brand
descrever	termfy-clispec, termfy-spec, termfy-api-import
implementar	termfy-cli, termfy-tui, termfy-mcp, termfy-config, termfy-api, termfy-database, termfy-command-set
verificar	termfy-test, termfy-review, termfy-docs
entregar	termfy-release, termfy-mcp-client

Repositórios

- `promovaweb/termfy` : framework, skills e este guia.
- `promovaweb/termfy-starter-kit` : esqueleto do binário, template do GitHub.

Visão geral

Você pode usar uma ferramenta de terminal digitando comandos, chamando-a por um script ou conectando um agente de IA. O script precisa de saída estruturada e o agente precisa descobrir as funções disponíveis. O Termfy reúne essas formas de uso no mesmo projeto para compartilhar as funções entre os três modos.

O Termfy resolve isso com um único executável que atende os três:

Público	Modo	Como entra
pessoa	TUI	`mycli` sem argumentos, em terminal interativo
pessoa, script, agente	CLI	`mycli <comando> [opções]`, texto ou `--json`
agente de IA	MCP	`mycli mcp`, servidor Model Context Protocol via stdio

Os três modos compartilham a configuração e a identidade do binário. Também usam o mesmo cofre de credenciais e os mesmos códigos de saída. Um comando novo aparece no `--help`, no `--json`, na descrição OpenCLI e na documentação embutida sem duplicar texto.

As peças

Starter kit. O repositório `promovaweb/termfy-starter-kit` é um template do GitHub com o esqueleto pronto: `src/` com as camadas `app`, `cli`, `config`, `mcp`, `tui` e `skill`; `docs/` embutida no binário; stubs em `template/`; testes; workflows de CI e release. Ele já vem com as skills do framework instaladas em `.agents/skills/`.

Framework. O repositório `promovaweb/termfy` guarda as 18 skills que ensinam um agente a criar e evoluir um projeto derivado, as referências que elas usam (documentação do OpenTUI, da OpenCLI, do MCP) e este guia, publicado em PDF e EPUB.

Projeto derivado. É o repositório que você cria a partir do starter kit, renomeando o executável e acrescentando comandos, telas e tools. As atualizações das skills chegam pela manutenção conduzida por `termfy-init`.

Escolhas que o framework já fez

- **Bun** como runtime, bundler e compilador. `bun build --compile` gera um executável único por plataforma, com a biblioteca nativa do OpenTUI e a documentação embutidas.
- **OpenTUI** (`@opentui/core`) para a interface de terminal, com layout flexbox, renderização incremental e renderer de teste para capturar frames.

- **MCP TypeScript SDK** 1.x para o servidor stdio, com schemas Zod.
- **OpenCLI 0.1** como formato de descrição da CLI, gerado do registro de comandos.
- **AES-256-GCM** para o cofre de credenciais, com chave mestra em arquivo 0600 ou em variável de ambiente.
- **Inglês no código** e na interface do binário: arquivos, identificadores, comentários, commands, options, keys e campos do `--json`. Documentação, guia e skills em português do Brasil, com os termos técnicos em inglês.

Público do guia

O guia se destina aos desenvolvedores da Promovaweb e aos alunos que constroem ferramentas internas, integrações com APIs e utilitários para agentes. Você precisa ter familiaridade com terminal e TypeScript para acompanhar os exemplos. Para estudar Bun ou MCP do início, consulte a documentação oficial indicada pelas skills.

Começar

Neste capítulo você cria um binário a partir do template, dá a ele um nome e confere que os três modos respondem. O exemplo do guia é o `mycli`, uma ferramenta para consultar e atualizar os pedidos de uma loja. Ele ainda não tem nenhum command de pedidos: o objetivo aqui é ter o esqueleto rodando com a identidade certa, porque todo o resto parte dele.

Antes de começar

- Bun 1.4.4 ou superior. Para instalar: `curl -fsSL https://bun.sh/install | bash`.
- Git e acesso de leitura aos repositórios `promovaweb/termfy` e `promovaweb/termfy-starter-kit`.
- Um terminal com cores de 24 bits para a TUI. A CLI e o MCP funcionam em qualquer terminal ou pipe.
- Um agente de código (Claude Code, Codex ou OpenCode, por exemplo) que você abre na pasta do projeto.

1. Crie o repositório pelo template

No GitHub, abra `promovaweb/termfy-starter-kit` e clique em **Use this template**. Dê ao repositório novo o nome `mycli` e clone-o:

```
git clone https://github.com/<seu-usuario>/mycli.git
cd mycli
```

O template já traz as 18 skills do Termfy em `.agents/skills/`. Você não precisa instalar nada além do Bun para continuar.

2. Peça o setup ao agente

Abra o agente na pasta `mycli` e escreva:

> Use a skill termfy-init para preparar este projeto. O executável será mycli, com o título My CLI. A descrição é “Query and update shop orders.” O autor é Promovaweb e o repositório é `https://github.com/<seu-usuario>/mycli`.

A descrição aparece no `--help` e fica em inglês, assim como os demais textos do binário. Você continua conversando com o agente em português.

A skill confirma os campos que faltarem e executa estes passos:

Passo	Comando que o agente roda	Para que serve
dependências	`bun install`	instala OpenTUI, SDK do MCP e Zod
identidade	`bun run init -- --name mycli ...`	troca o nome do binário em `package.json`, `README.md`, `AGENTS.md`, `.env.example` e nos workflows
skills	`bun run skills:check`	confere se as 18 skills do Termfy estão instaladas
descrição OpenCLI	`bun run opencli`	regenera `docs/opencli.json` com o nome novo
validação	`bun run validate`	roda typecheck, testes, checagens da documentação e o build
README	edição do arquivo	troca a abertura do template pelo propósito do `mycli`

A saída do `init` mostra cada arquivo alterado:

```
✓ package.json: termfy → mycli
✓ .env.example
✓ README.md
✓ AGENTS.md
✓ CHANGELOG.md: history started at 0.1.0

Done. Check with: bun run dev -- --version && bun run dev -- --help
Then run: bun run validate
```

O nome que você escolheu no `package.json` também define a pasta `~/.mycli/`, onde ficam preferências e segredos. Define também as variáveis de ambiente `MYCLI_HOME` e `MYCLI_MASTER_KEY` e o prefixo `mycli://` das URIs que o servidor MCP publica. Por isso o nome aceita só letras minúsculas, números e hífen, entre 2 e 40 caracteres, começando por letra.

Repare também no que o `init` mantém. As skills continuam com os nomes `termfy-cli`, `termfy-spec` e os demais, porque vêm do framework e não do seu projeto. O `AGENTS.md` passa a dizer que o repositório é o binário `mycli`, criado com o starter kit do Termfy. O projeto começa na versão `0.1.0`, com um `CHANGELOG.md` próprio. O histórico do template fica no template.

No fim, a skill sugere seguir para a `termfy-clispec`, que entrevista você sobre o propósito do binário. Antes, confira o que o setup entregou.

3. Confira os três modos

Durante o desenvolvimento, você roda o binário com `bun run dev --` seguido dos argumentos. Comece pela versão:

```
bun run dev -- --version
```

```
mycli 0.1.0
```

CLI em texto

O `--help` lista o que o esqueleto já sabe fazer. Este é o começo da saída:

```
My CLI v0.1.0
Query and update shop orders.

Usage

  mycli <subcommand> [options]
  mycli                       opens the TUI (no arguments, interactive terminal)
  mycli mcp                   starts the MCP server over stdio

Commands

  tui                 Open the terminal interface (Overview and About). Requires a
TTY.
  about               Show version, executable, runtime, platform and credits.
  config              Read and write preferences in ~/.mycli/settings.json.
  credentials (cred)  Store encrypted secrets (AES-256-GCM) in ~/.mycli/
credentials.json.
  mcp                 Start the MCP server (stdio) with tools, resources and prompts
for agents.
  skill               Print the embedded documentation, whole or by topic. Same
effect as --skill.
  doctor              Check runtime, terminal, settings, key source and vault
readability without changing files.
  completion          Print a registry-generated completion script for bash, zsh or
fish.
```

Esses oito commands são o esqueleto. Os commands que você criar aparecem nesta mesma lista, com as mesmas options globais (`--json` , `--skill` , `--opencli` , `--no-color`). A ajuda termina com a tabela de exit codes, explicada no capítulo CLI.

CLI em JSON

Scripts e agentes leem a saída com `--json`. Todo command devolve o mesmo envelope, com `ok`, `command` e `data`:

```
bun run dev -- about --json
```

```
{
  "ok": true,
  "command": "mycli about",
  "data": {
    "bin": "mycli",
    "title": "My CLI",
    "version": "0.1.0",
    "description": "Query and update shop orders.",
    "author": "Promovaweb"
  }
}
```

A saída real traz mais campos em `data`: site, repositório, licença, executável, runtime, plataforma, pasta de dados e créditos. O trecho acima mostra os que vieram do seu pedido.

Quando algo falha, o envelope troca `data` por `error`. O exit code do processo é o mesmo de `error.code`:

```
bun run dev -- config get nope.key --json
```

```
{
  "ok": false,
  "command": "mycli config get",
  "error": {
    "code": 3,
    "name": "CONFIG",
    "message": "Key \"nope.key\" does not exist. See \"mycli config schema\".",
    "details": {
      "key": "nope.key"
    }
  }
}
```

Diagnóstico

O `doctor` confere o ambiente sem alterar arquivos:

```
bun run dev -- doctor
```

```
✓ bun: 1.4.4; minimum 1.4.4
! stdin: Not a TTY
! stdout: Not a TTY
! directory: Not created
! settings: Not created
! master-key: none
! credentials: Not created
✓ binary: mycli 0.1.0
✓ platform: linux-x64
```

Os itens com `!` são avisos que mantêm o exit code `0` do `doctor`. Num projeto recém-criado, a pasta `~/.mycli/` ainda não existe, porque o binário só a cria quando precisa gravar algo. Faça o teste: grave uma preferência e guarde uma credencial.

```
bun run dev -- config set tui.theme day
printf 'shp_live_1234567890abcdef' | bun run dev -- credentials set shop.token --
stdin
bun run dev -- credentials get shop.token
```

```
✓ tui.theme = "day"
✓ credential "shop.token" saved to ~/.mycli/credentials.json
shp_************ef
```

A gravação cria a pasta com permissão `0700` e salva a preferência em texto no `settings.json`. O token é cifrado com AES-256-GCM em `credentials.json`, com a master key gerada em `~/.mycli/master.key`. O `get` mostra o valor mascarado: para ver o valor inteiro, você precisa usar `--reveal`. O capítulo Configuração e credenciais explica o cofre.

MCP

Para ver o que um agente encontra quando se conecta ao binário:

```
bun run dev -- mcp --list
```

```
MCP server mycli v0.1.0

Tools:
  about  Version, executable, runtime, platform and credits of the binary.
  exit_codes  Process exit-code table.
  skill  Embedded documentation, whole or by topic.
  config_list  Every settings.json key with its effective value.
  config_get  Value of a settings.json key.
  credentials_status  Saved names and master key source, without values.
  doctor  Diagnose runtime, terminal and configuration health. Returns checks and ok
without credential values or file changes.

Resources:
  mycli://about  Binary identity and runtime as JSON.
  mycli://exit-codes  Exit-code table as JSON.
  mycli://skill  Full documentation in Markdown.
  mycli://skill/{topic}  One documentation section. Topics: cli, tui, mcp, config,
exit-codes, reference.

Prompts:
  operate  Guides the agent to read the documentation and run a task through the
CLI.
```

As tools do esqueleto só leem. Nenhuma devolve o valor de uma credencial nem grava arquivos. Você grava pela CLI, com a confirmação prevista por cada comando. O capítulo MCP mostra como acrescentar tools.

TUI

Por último, abra a interface num terminal interativo:

```
bun run dev
```

```
▌ My CLI  v0.1.0                                                    linux-
x64 · Bun 1.4.4

   1 ⌂ Overview    2 ⚙ Settings    3 ◈ Credentials    4 ≡ Docs    5 i About
   ──────────

   ENVIRONMENT                                    STATE
   ┌──────────────────────────────────────────────┐
┌──────────────────────────────────────────────┐
   │                                              │
│                                              │
   │  Binary     mycli v0.1.0                     │    │  Settings     •
defaults                  │
   │  Runtime    Bun 1.4.4                        │    │  Vault        •
empty                     │
   │  Platform   linux-x64                        │    │  Master key   not
created                 │
   │                                              │
│                                              │
   └──────────────────────────────────────────────┘
└──────────────────────────────────────────────┘

   NEXT STEPS

┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────┐

│
│
   │  → mycli --help                              { "command":
"mycli",                        │
   │  → mycli credentials set <name>                "args":
["mcp"] }                          │
   │  → mycli --
skill                                                                          │
   │  [c] copy MCP config    [r]
reload                                                    │

│
│

└──────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

```
   0 credentials · defaults  r reload  c copy MCP config  ? help  q quit
^K commands
```

Este frame mostra um projeto sem preferências gravadas. Cada aba tem número e ícone: `2` ou um clique em `Settings` abre a segunda aba. À esquerda da barra de baixo fica o resumo da aba. Depois vêm as ações da aba e as teclas globais. Um clique num item executa a ação, como a tecla. `Ctrl+K` abre a paleta com todos os comandos, `?` abre o help e `q` sai. Num pipe, o binário sem argumentos imprime a ajuda e sai com `2`. Já `mycli tui` sai com `8` (NO_TTY).

O que mudou no projeto

Arquivo	O que mudou
`package.json`	`name`, `bin` e o bloco `termfy` com `mycli`; versão `0.1.0`
`CHANGELOG.md`	histórico novo, começando em `0.1.0`
`README.md`	título `My CLI`, abertura com o propósito e exemplos com `mycli`
`AGENTS.md`	apresenta o repositório como o binário `mycli`; as skills seguem com os nomes `termfy-*`
`.env.example`	variáveis `MYCLI_HOME` e `MYCLI_MASTER_KEY`
`docs/opencli.json`	descrição OpenCLI regenerada com o nome novo

O código em `src/` não mudou. Ele lê o nome, o título e a versão do `package.json` quando roda, por isso o `--help`, a TUI e o MCP já mostram `mycli` sem que você edite nenhum arquivo de código.

Manter as skills atualizadas

As skills evoluem no framework. Para trazer a versão nova a um projeto que já existe, peça:

> Use termfy-init no modo de manutenção para atualizar as skills do Termfy deste projeto e conferir a compatibilidade com o starter kit.

O agente roda `bun run skills:update`, confere o `skills-lock.json` e os links dos agentes e roda `bun run skills:check`. Nesse modo, a skill preserva o nome do produto, o `CLISPEC.md` e as specs.

Se algo der errado

Sintoma	Causa provável	O que fazer
`bun: command not found`	Bun não instalado ou fora do `PATH`	instale com `curl -fsSL https://bun.sh/install | bash` e abra um terminal novo
`init` termina com exit code `2`	nome fora da regra	use só letras minúsculas, números e hífen, começando por letra
`bun run dev` imprime a ajuda e sai com `2`	o terminal não é interativo	abra a TUI num terminal de verdade; num pipe, use os commands da CLI
`skills:check` falha	skills ausentes ou alteradas	peça ao agente: “Use termfy-init no modo de manutenção das skills”

Próximo passo

O esqueleto roda, mas ninguém escreveu ainda o que o `mycli` deve fazer. O capítulo Descrever e especificar mostra a entrevista da `termfy-clispec`, o `CLISPEC.md` que ela grava e a primeira spec.

Três modos

Despacho dos três modos

`src/main.ts` decide o modo em três linhas:

1. Sem argumentos e com stdin e stdout em um TTY: abre a TUI.
2. Sem argumentos e sem TTY: imprime a ajuda e encerra com código 2. Um pipe ou um agente que esqueceu os argumentos recebe a resposta na hora, em vez de ficar preso esperando uma interface.
3. Com argumentos: roda o CLI. `mcp` é um comando como os outros, só que não devolve enquanto o transporte stdio estiver aberto.

Isso mantém um único executável e um único `--help`. Não existe `mycli-tui` nem `mycli-mcp`.

O que os modos compartilham

Peça	Arquivo	Usado por
identidade (nome, versão, autor, links)	`src/app/manifest.ts`	cabeçalho da TUI, `about`, `--version`, OpenCLI, nome do servidor MCP
códigos de saída	`src/app/exit-codes.ts`	`--help`, OpenCLI, tool `exit_codes`, resource `exit-codes`
erros de domínio	`src/app/errors.ts`	todos os comandos
preferências	`src/config/settings.ts`	tema e aba inicial da TUI, formato padrão do CLI, tools de leitura
cofre	`src/config/credentials.ts`	comandos que chamam APIs, `credentials_status`
documentação	`docs/` embutida via `src/skill/index.ts`	`--skill`, comando `skill`, tool e resource `skill`

Quando usar cada modo

TUI quando você quer navegar pelo estado do ambiente ou consultar uma lista e abrir os detalhes de um item. A interface apresenta os resultados das funções do binário e sugere os comandos correspondentes da CLI.

CLI em texto quando você digita comandos e lê o resultado no terminal. A saída usa tabelas alinhadas e cores por papel, com o código entre colchetes nas mensagens de erro.

CLI em JSON quando um script, um pipeline ou um agente lê. Envelope estável `{ ok, command, data }` em sucesso e `{ ok, command, error }` em exceções, em stdout nos dois casos. Nessas exceções, o exit code corresponde a `error.code`. `doctor` também pode devolver `data.ok: false` com envelope `ok: true` e saída 3. Confira o exit code além do envelope. `--opencli` imprime JSON próprio, sem esse envelope.

MCP quando um agente conversa por protocolo, usando tools para executar funções, resources para leitura e prompts para orientar a conversa. A tool `skill` devolve a mesma documentação do `--skill`, então o agente descobre o binário sem sair do protocolo.

Documentação embutida

`docs/` entra no binário em tempo de build como texto. `mycli --skill` imprime tudo em Markdown; `--skill cli` imprime só a seção da CLI, com a referência de comandos gerada do registro no final. Um agente que recebe o binário compilado, sem o repositório, lê a documentação completa com um comando.

O mesmo conteúdo sai pela tool MCP `skill`, pela resource `mycli://skill` e pelo comando `mycli skill [topic]`. Os tópicos são `cli`, `tui`, `mcp`, `config`, `exit-codes` e `reference`. Este último devolve apenas a referência gerada do registro. O texto completo inclui as páginas importadas em `src/skill/index.ts`, não todos os arquivos presentes em `docs/`.

Skills

Skills são pastas com um `SKILL.md` (instruções para o agente) e uma pasta `references/` (material de apoio). Um agente de IA que trabalha no projeto lê o `AGENTS.md`, escolhe a skill compatível com a tarefa e segue os passos.

As dezoito skills

Skill	Quando o agente usa
`termfy`	a pessoa descreve a ferramenta em uma frase; orquestra as demais com dois checkpoints
`termfy-api-import`	a pessoa envia a documentação de uma API; vira a seção Surface de uma spec
`termfy-api`	criar o client HTTP de um serviço externo, com token do cofre e testes locais
`termfy-command-set`	gerar uma família `recurso verbo` com tools e screen de uma vez
`termfy-test`	cobrir CLI, MCP, TUI e config com os testes de aceite da spec antes de marcar seus itens
`termfy-review`	auditar uma entrega e relatar por gravidade, sem editar
`termfy-release`	publicar versão, changelog, tag e release com binários
`termfy-mcp-client`	configurar o binário nos clientes de agente e validar
`termfy-brand`	ajustar nome, título, descrição, autor, links, créditos e paleta, em todas as superfícies do binário
`termfy-clispec`	entrevistar a pessoa para entender o que o binário precisa fazer e escrever o `CLISPEC.md`, a visão macro única: propósito, públicos, modos, fora de escopo e índice das specs
`termfy-spec`	entrevistar a pessoa a cada evolução e escrever `specs/NNNN-slug.md`: mudanças na superfície, como o binário vai funcionar, como o código deve ser feito, tarefas e testes
`termfy-database`	guardar dados entre execuções em SQLite, só quando a pessoa pede, com migrations embutidas no binário

Skill	Quando o agente usa
`termfy-init`	fazer o setup de um binário ou manter as skills de um projeto existente
`termfy-cli`	criar ou alterar comandos, opções, saída, códigos de saída e OpenCLI
`termfy-tui`	criar ou alterar telas, componentes, tema e teclas
`termfy-mcp`	criar ou alterar tools, resources e prompts; testar com o Inspector
`termfy-config`	declarar chaves de configuração, consumir credenciais, revisar o cofre
`termfy-docs`	melhorar a documentação embutida, passar no `docs:check` e reler o `--skill` como um agente

Cada `SKILL.md` segue a mesma forma: o que ler antes, regras que não mudam, passos, marcos de progresso e o que relatar ao final. As regras repetem o contrato do framework (toda opção documentada, segredos fora de argumentos, stdout só para dados), para que o agente não dependa de ter lido o guia.

O CLISPEC.md e as specs

Antes de implementar, o agente precisa saber o que construir. O contrato, descrito na skill `termfy-spec`, tem dois documentos em Markdown puro, sem front matter. O agente cria e edita os dois à mão; o binário não os lê nem os embute.

`CLISPEC.md`, um por binário, na raiz, guarda a visão macro. A skill `termfy-clispec` entrevista você sobre o problema que a ferramenta resolve e os públicos que ela atende. A conversa detalha as capacidades e os limites, incluindo armazenamento e integrações, para registrar a identidade e as seções `Purpose`, `Audiences`, `Out of scope`, `Surface`, `Specs` e `Notes`. `Surface` descreve em tabelas o que já foi entregue: commands, screens, MCP, settings, credenciais e, quando há banco, as tabelas SQLite em `### Storage`, com os nomes do código. `Specs` é o índice das specs com título e status. O `CLISPEC.md` não tem checklist: ele descreve o binário, não o trabalho pendente.

`specs/NNNN-slug.md`, um por evolução, guarda o que muda. A skill `termfy-spec` entrevista você para descrever o comportamento esperado e como implementá-lo. A spec registra a superfície alterada e relaciona as tarefas aos testes de aceite. O agente também grava a linha `Status:` (`draft`, `in_progress`, `done` ou `archived`) e as seções `Goal`, `Behavior` (como o binário vai funcionar), `Design` (como o código deve ser feito), `Surface` (o que entra, muda ou sai da superfície), `Tasks` (checklist com a tag da skill responsável no fim de cada item), `Tests` (checklist de aceite) e `Notes`.

As skills de implementação leem a spec aberta, seguem o Behavior e o Design e marcam `[x]` em cada tarefa e teste concluídos. Cada item de `Surface` precisa aparecer em uma tarefa ou um teste. Enquanto a spec está aberta, o código fica à frente do `CLISPEC.md`. Isso é esperado. Com todos os itens marcados, o agente fecha a spec: aplica `Surface` às tabelas do `CLISPEC.md`, troca o status para `done` na spec e no índice e registra a data em `Notes`.

O esqueleto guarda preferências e credenciais em arquivos. Você pode acrescentar um banco para preservar registros entre execuções. Nesse caso, `termfy-database` cria `~/.<bin>/data.db` com SQLite, muda o schema só por migrations embutidas no binário, faz backup antes de migrar e lista as tabelas em `Surface`, que chegam a `### Storage` quando a spec fecha.

Nenhum script confere esses documentos. `bun run validate` cobre o código e a documentação embutida; a correspondência entre `CLISPEC.md`, specs e código é conferida pela leitura, na revisão de `termfy-review`: cada linha de `Surface` existe no código, cada `getCredential("<name>")` aparece nas credenciais, as tabelas das migrations batem com `### Storage` e o índice tem uma linha por arquivo de `specs/`.

A sequência

A ordem de uso, do `termfy-init` à release, com o que cada skill deixa pronto para a seguinte, está no capítulo Sequência de uso das skills.

As referências

`references/` traz o que a skill cita:

- `termfy-cli`: contrato de comando, saída texto/JSON e o JSON Schema oficial da OpenCLI 0.1.
- `termfy-tui`: o guia de design modern unix, o guia de testes por frame e 39 páginas da documentação oficial do OpenTUI 0.8.3.
- `termfy-mcp`: contrato de tool, guia de testes, princípio de economia de tokens do OpenMCP, documentação do MCP TypeScript SDK 1.x e páginas da especificação MCP (conceitos de servidor, transporte stdio, tools, resources, prompts, segurança local, Inspector e depuração).
- `termfy-config`: modelo de segurança do cofre e declaração de chaves.
- `termfy-brand`: onde cada campo da identidade fica e que superfícies o leem, mais os papéis da paleta.
- `termfy-docs`: checklist de cobertura e forma da documentação embutida e o guia de escrita para pessoas e agentes.
- `termfy-init`: checklist do bootstrap.
- `termfy-clispec`: roteiro da entrevista e guia da visão macro (Purpose, Audiences, Out of scope e lista de specs iniciais).

- `termfy-spec`: resumo do contrato das specs, roteiro da entrevista por evolução e guia para escrever Behavior e Design.
- `termfy-database`: regras de migrations e de repositórios SQLite.

Pastas com documentação copiada de outros projetos levam um marcador `.terceiros`. Validadores de texto de instrução ignoram essas pastas, porque o texto não foi escrito aqui. As licenças originais acompanham as cópias.

Como chegam ao projeto

O starter kit já traz as skills em `.agents/skills/`, com links para os agentes configurados. O arquivo `skills-lock.json` registra a origem, o caminho e o hash de cada skill.

Peça ao agente para usar `termfy-init` no modo de manutenção das skills. Ela instala, atualiza ou repara as cópias do Termfy e confere o lock, os links dos agentes e a compatibilidade, preservando a identidade, o `CLISPEC.md` e as specs.

Por exemplo: “Use termfy-init para reparar as skills ausentes deste projeto”. O agente executa os comandos internos e apresenta o resultado. As mudanças nas instruções são feitas no framework e distribuídas para os projetos derivados.

Compatibilidade

As skills citam caminhos do starter kit (`src/cli/registry.ts`, `template/command/example.ts`). Cada `SKILL.md` declara, na seção “Compatibilidade”, a faixa do starter kit que cobre. Quando o starter kit muda a estrutura, as skills são atualizadas na mesma janela e as duas versões sobem juntas.

Escrever uma skill nova

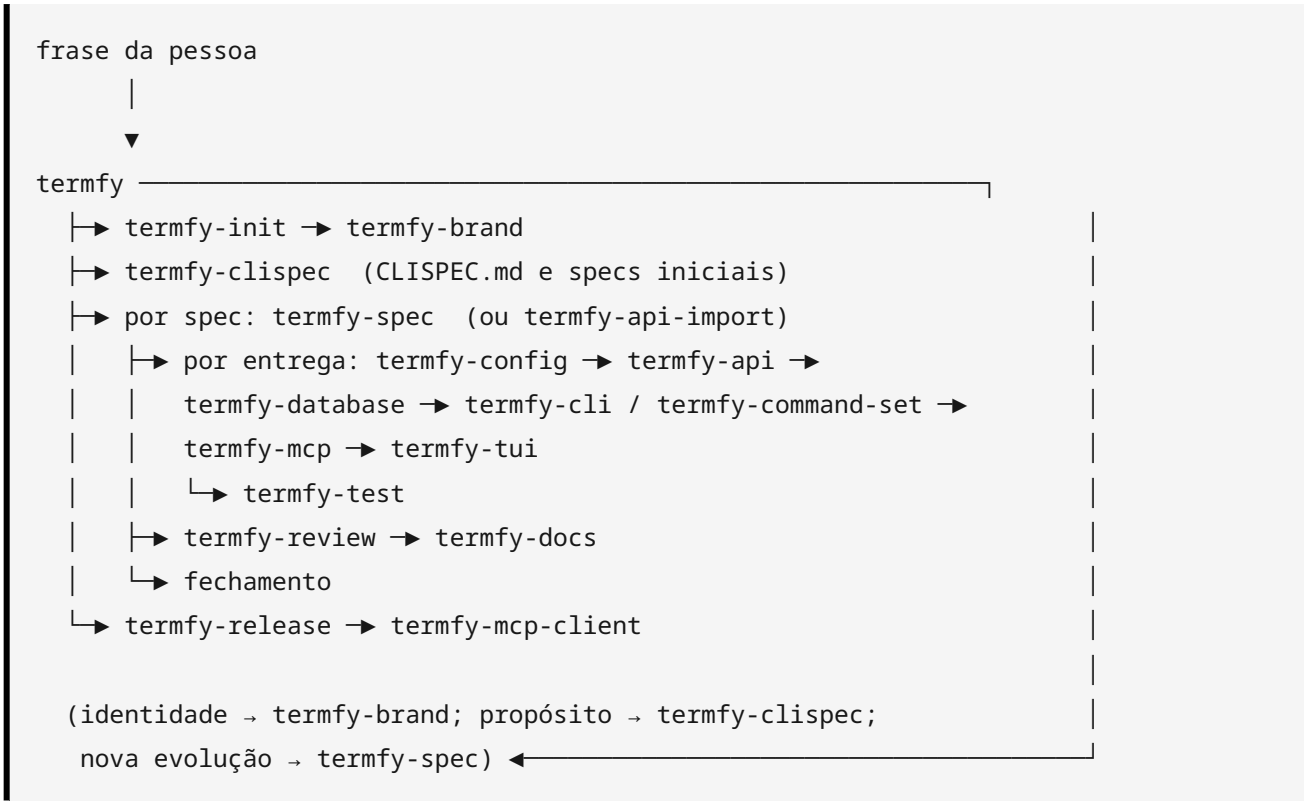
Quando um projeto derivado tem um domínio próprio (uma API, um formato de arquivo, um fluxo de aprovação), vale escrever uma skill do projeto, em `skills/` do próprio repositório ou em uma biblioteca mantida pelos desenvolvedores. Siga a mesma forma: frontmatter com `name` e `description`, leitura prévia, regras, passos, marcos e saída. Texto em prosa direta, verbos no imperativo, sem clichês de abertura ou fechamento. Valide a estrutura com um script como `scripts/validate-skills.ts` do framework.

Sequência de uso das skills

As dezoito skills cobrem o ciclo de um binário, da sua descrição inicial à release. Cada etapa produz os arquivos que a seguinte consulta para continuar.

O trabalho gira em torno de dois documentos. O `CLISPEC.md`, único por binário, guarda a visão macro: o que o binário é, a que públicos serve, o que fica de fora e a superfície já entregue. Cada evolução ganha uma spec em `specs/NNNN-slug.md`, com as mudanças na superfície, como o binário vai funcionar (Behavior), como o código deve ser feito (Design), as tarefas e os testes de aceite. A spec começa na entrevista, recebe as marcas das skills de implementação e, concluída, tem a superfície copiada para o `CLISPEC.md` pelo agente. Os dois documentos são Markdown puro, escritos à mão pelo agente; o binário não os lê.

Visão geral



0. Entrar: `termfy`

Use `termfy` para descrever o que você quer em linguagem natural quando ainda não souber qual skill chamar. Ela diagnostica o estágio do repositório, confirma nome, propósito e públicos em uma única mensagem e chama as demais skills na ordem abaixo. Para só duas vezes: na aprovação do `CLISPEC.md` com a primeira spec e antes da release.

1. Criar: `termfy-init`

No setup, logo depois de clicar em **Use this template**, peça ao agente para usar `termfy-init`. Ela aplica a identidade, mantém as skills e passa para `termfy-clispec`, confere `--version`, `about --json` e `mcp --list` e roda `bun run validate`.

Em um projeto existente, use a mesma skill no modo de manutenção para instalar, atualizar ou reparar as skills, preservando a identidade, o `CLISPEC.md` e as specs. O agente executa os comandos internos de cada modo.

2. Apresentar: `termfy-brand`

Quando o nome, o título, a descrição, o autor, os links, a licença, os créditos ou as cores mudarem. Cuida do que o `init` não altera: créditos, paleta, painel “Standards” da aba About, campos de identidade do `CLISPEC.md`. Confere as superfícies: `--version`, `about`, aba About, `info` do OpenCLI e nome do servidor MCP.

3. Descrever o binário: `termfy-clispec`

Use `termfy-clispec` no início do projeto e retome a entrevista quando o propósito, os públicos ou os limites mudarem. A skill entrevista você para entender o problema que o binário resolve e os modos que precisa atender. Ela detalha as capacidades, as integrações e o armazenamento necessário para gravar a visão macro no `CLISPEC.md`. Com você, monta a lista de specs iniciais, uma por capacidade entregável. Grava cada uma em `specs/` como `draft`, com a linha correspondente no índice.

Deixa pronto: o `CLISPEC.md` com Purpose, Audiences, Out of scope, a Surface do que o esqueleto já entrega e o índice das specs iniciais em `draft`.

4. Especificar cada evolução: `termfy-spec` ou `termfy-api-import`

Para cada evolução, `termfy-spec` entrevista em blocos curtos e escreve a spec: Goal, Behavior, Design, Surface com as mudanças na superfície, Tasks e Tests. Quando o ponto de partida é a documentação de uma API (OpenAPI, Postman, página ou texto), `termfy-api-import` lê a documentação, propõe commands por recurso, tools, settings e credenciais. Grava tudo em Surface de uma spec nova. Em ambos os casos, a spec segue o contrato antes de sair de `draft`: Goal, Behavior e Design com texto e ao menos uma tarefa. O que a spec planeja e o código ainda não tem é o trabalho da evolução.

Deixa pronto: uma tarefa com a tag da skill responsável por entrega e um teste de aceite por resultado esperado.

5. Implementar, uma entrega por vez

Ordem para um recurso novo:

1. `termfy-config` declara a key e a credencial que o recurso usa.
2. `termfy-api` cria o client do serviço externo, quando houver.
3. `termfy-database` cria as tabelas por migration e os repositórios, só quando a spec descreve registros que o binário guarda entre execuções.
4. `termfy-cli` cria um command avulso; `termfy-command-set` gera a família `recurso list|show|create|update|delete` de uma vez, com tools e screen.
5. `termfy-mcp` espelha o que faltou em tools, reutilizando as funções.
6. `termfy-tui` dá uma aba ao recurso, com `listPanel` quando for lista.
7. `termfy-test` escreve os testes de aceite da spec nos três modos antes de marcar o item.

Cada skill lê a spec aberta antes de mexer no código, segue o Behavior e o Design e, ao concluir e testar, marca `[x]` no item da spec. O `CLISPEC.md` não muda nessa etapa: o código fica à frente dele até o fechamento. Se o código ganhou algo que nenhuma spec previa, crie uma spec de adoção e descreva o que já existe em Surface; essas entradas chegam ao `CLISPEC.md` quando ela fechar.

6. Fechar a spec: `termfy-review`, `termfy-docs` e `termfy-spec`

`termfy-review` compara a entrega com o Behavior e o Design da spec e confere o `CLISPEC.md` e as specs contra o código. A revisão também procura segredos expostos e saída indevida no stdout do MCP. Ela verifica os contratos de saída, a documentação e os padrões de interface do framework. O relatório indica se a spec pode fechar e classifica as falhas por gravidade, sem editar. Corrigido o que ela apontou, `termfy-docs` relê `bun run dev -- --skill` como um agente leria, corrige a cobertura apontada por `bun run docs:check` e regenera `docs/opencli.json`. Com todos os itens marcados, o agente fecha a spec pelo fluxo de `termfy-spec`: aplica Surface às tabelas do `CLISPEC.md`, troca o status para `done` na spec e no índice e registra a data nas Notes do `CLISPEC.md`. A próxima spec da lista volta à etapa 4.

7. Publicar e ligar aos agentes: `termfy-release` e `termfy-mcp-client`

`termfy-release` publica depois que as specs da versão foram fechadas. Avança a versão, escreve o changelog (que pode citar os IDs das specs fechadas), valida e cria commit e tag. Depois confere a release que o workflow gera, com binários e checksums. `termfy-mcp-client` configura o binário em Claude Code, Codex, Cursor, OpenCode ou Gemini CLI e valida com `mcp --list` e o Inspector.

Quando sair da ordem

- Alteração pequena em um command existente: `termfy-spec` escreve uma spec curta com a alteração em Surface, uma tarefa e um teste; depois `termfy-cli`, `termfy-test` e o fechamento da spec.
- Só mudar texto da documentação: `termfy-docs` direto.
- Só mudar cores ou créditos: `termfy-brand` direto.
- Só revisar antes de um merge: `termfy-review` direto.
- Atualizar as skills para uma versão nova do framework: `termfy-init` no modo de manutenção das skills; se a faixa de compatibilidade mudou, leia o `CHANGELOG.md` do framework antes da próxima etapa.

O que cada skill não faz

- `termfy` não implementa; orquestra.
- `termfy-init` não escreve o `CLISPEC.md` do produto; passa para `termfy-clispec`.
- `termfy-clispec` não planeja tarefas; descreve o binário e cria as specs iniciais.
- `termfy-spec` e `termfy-api-import` não implementam; descrevem e conferem.
- `termfy-database` só entra quando você solicita armazenamento local; o esqueleto guarda apenas settings e credenciais.
- As skills de implementação não inventam requisitos nem marcam itens no `CLISPEC.md`; o que falta vai para `## Notes` da spec.
- `termfy-review` não edita; relata.
- `termfy-docs` não documenta o framework nem as skills; documenta o binário.
- Só `termfy-release` faz commit, tag e release. Ela faz isso quando a você solicita a publicação.

Descrever e especificar

O Termfy registra o que construir em dois documentos em Markdown puro: o `CLISPEC.md` descreve o binário inteiro e cada arquivo de `specs/` detalha uma evolução que o agente vai implementar. Neste capítulo você acompanha as duas entrevistas do `mycli` e vê os documentos que elas gravam.

Documento	Quantos	Responde	Skill
`CLISPEC.md`	um, na raiz	o que o binário é, a que públicos serve, o que já entrega	termfy-clispec
`specs/NNNN-slug.md`	um por evolução	o que muda agora, como funciona, como o código deve ser feito	termfy-spec

Esses documentos são escritos para você e para o agente, sem leitura pelo binário. O agente consulta a spec antes de mexer no código e marca o que entregou, para você conferir cada tarefa e teste no próprio arquivo.

1. A entrevista do CLISPEC

Peça ao agente:

> Use a skill termfy-clispec para descrever o mycli. É uma ferramenta para a equipe da loja consultar pedidos sem abrir o painel web.

A skill pergunta em blocos curtos, na ordem problema, públicos e modos, capacidades, dados e integrações, limites. Quando a sua frase já responde um bloco, ela confirma o que entendeu e segue. Um trecho da conversa:

```
Agente: Que tarefa a equipe faz hoje sem o mycli, e o que dá trabalho nela?
Você:   Abre o painel da loja para ver pedidos abertos. É lento e não dá
        para usar em script.
Agente: Que públicos vão usar o binário? Recomendo três:
        1. operador no terminal (TUI e CLI texto) (recomendado)
        2. só scripts e CI (CLI --json)
        3. só agentes (MCP)
Você:   O 1, mais o CI exportando pedidos do dia e um agente que só consulta.
Agente: Entendi: operador na TUI e na CLI, CI no --json, agente no MCP só
        para leitura. O binário fala com a API da loja com um token?
Você:   Sim, um token de API do painel.
```

A cada bloco respondido, a skill escreve a seção correspondente e mostra o trecho. No fim, monta com você a lista de specs iniciais, uma por capacidade que dá para entregar sozinha.

2. O CLISPEC.md gravado

```
# My CLI

`mycli`: Query and update shop orders.

## Purpose

A equipe da loja consulta pedidos no painel web, que é lento e não serve
para scripts. O `mycli` lista e detalha pedidos pela API da loja, no
terminal, em texto ou JSON. Um agente de IA pode consultar os pedidos pelo
MCP, sem alterar nada.

## Audiences

| Público | Modo | Necessidade |
| --- | --- | --- |
| operador | TUI e CLI texto | ver pedidos abertos e o detalhe de um pedido |
| CI | CLI `--json` | exportar os pedidos do dia com saída estável |
| agente de IA | MCP | consultar pedidos, só leitura |

## Out of scope

- Criar, cancelar ou reembolsar pedidos.
- Várias lojas no mesmo binário.

## Surface

### Commands

| Command | Função | Options | `data` no `--json` | Exit codes | Credenciais |
| --- | --- | --- | --- | --- | --- |
| `about` | Show version, executable, runtime, platform and credits. | nenhuma |
`{ bin, title, version, ... }` | 0 | nenhuma |
| `config list\|get\|set\|unset\|path\|schema` | Read and write preferences in
`~/.mycli/settings.json`. | `key`, `value` | `{ key, value }` e variações | 2, 3 |
nenhuma |

### Screens

| Screen | Função | Teclas |
| --- | --- | --- |
| `overview` | ambiente, estado da configuração e próximos passos | `←` `→`, `?`,
`q` |
```

```
| `about` | identidade, projeto e créditos | `←` `→`, `?`, `q` |

### MCP

| Tipo | Nome | Função | Efeito |
| --- | --- | --- | --- |
| tool | `about` | identidade e runtime | `read` |
| tool | `config_get` | valor de uma key | `read` |

### Settings

| Key | Tipo | Padrão | Função |
| --- | --- | --- | --- |
| `output.format` | `enum` (`text`, `json`) | `text` | formato padrão da saída |
| `tui.theme` | `enum` (`terminal`, `night`, `day`) | `terminal` | paleta da TUI |

## Specs

| ID | Título | Status |
| --- | --- | --- |
| 0001 | Listar pedidos | draft |
| 0002 | Detalhar um pedido | draft |

## Notes

- 2026-11-12: visão macro registrada em entrevista.
```

O exemplo encurta as tabelas de Surface. No arquivo real, a skill lista tudo o que o esqueleto entrega, lido do código: os commands do registry, as screens de `src/tui/screens/index.ts`, o que aparece em `mcp --list` e as keys de `SETTINGS_SCHEMA`. Assim a Surface começa igual ao código.

Observe como o documento registra a superfície e a identidade:

- **Surface descreve só o que já existe.** O command `orders list` ainda não aparece nela, porque nenhuma spec foi fechada. Trabalho pendente fica nas specs.
- **A linha de identidade copia o `package.json`.** Para mudar nome ou descrição, use a termfy-brand.
- **Nomes do código em inglês, prosa no seu idioma.** `orders list`, `shop.token` e `tui.theme` aparecem como no código.

Cada spec aprovada vira um arquivo `specs/0001-list-orders.md` com `Status: draft` e as seções vazias, mais uma linha no índice `## Specs`.

3. A entrevista da spec

Peça ao agente:

> Use a skill termfy-spec para detalhar a spec 0001.

A entrevista registra o objetivo e o comportamento esperado, depois detalha a superfície, o armazenamento e as integrações. Por fim, define como implementar e testar a evolução. As perguntas agora são concretas: que options o command aceita, que campos vão no `--json`, que exit code sai quando o token falta, qual arquivo guarda o client HTTP.

4. A spec gravada

````
# 0001 Listar pedidos

Status: in_progress

## Goal

O operador e o CI precisam ver os pedidos da loja sem abrir o painel.
Filtrar por status entra; paginação além de 100 pedidos fica para depois.

## Behavior

`mycli orders list` mostra os pedidos abertos numa tabela com número,
cliente, total e data. `--status closed` troca o filtro. `--json` devolve
`{ count, orders[] }`. Sem `shop.token` no cofre, sai com 4 e a mensagem
manda rodar `mycli credentials set shop.token`. Falha da API sai com 5.
A tool `orders_list` devolve o mesmo `data`.

## Design

O client fica em `src/api/shop.ts`, com `listOrders({ status, limit })`,
usado pelo command e pela tool. A URL vem da key `shop.baseUrl`; o token,
de `getCredential("shop.token")`. GET com até três tentativas e timeout
de 5 segundos. O command segue `template/command/example.ts`; a tool,
`template/tool/example.ts`.

## Surface

- Adiciona o command `orders list`: List shop orders;
  `--status <open|closed>`, `--limit <n>`; `data` `{ count, orders[] }`;
  exit codes 4 e 5; credencial `shop.token`.
- Adiciona a tool `orders_list`: lista pedidos; efeito `network`.
- Adiciona a setting `shop.baseUrl`: `string`, padrão
  `https://api.example.com`, URL base da API.
- Adiciona a credencial `shop.token`: usada por `orders list` e
  `orders_list`; painel da loja, seção API.

## Tasks

- [ ] Key `shop.baseUrl` em `SETTINGS_SCHEMA` `#termfy-config`
- [ ] Client `listOrders` com timeout e retry `#termfy-api`
- [ ] Command `orders list` `#termfy-cli`
- [ ] Tool `orders_list` `#termfy-mcp`
````

```
- [ ] `orders list` em `docs/cli.md` `#termfy-docs`

## Tests

- [ ] `mycli orders list --json` devolve `{ count, orders[] }`
- [ ] Sem `shop.token`, `orders list` sai com exit code 4
- [ ] API com 500 três vezes faz `orders list` sair com 5
- [ ] `orders_list` devolve o mesmo `data` do command

## Notes

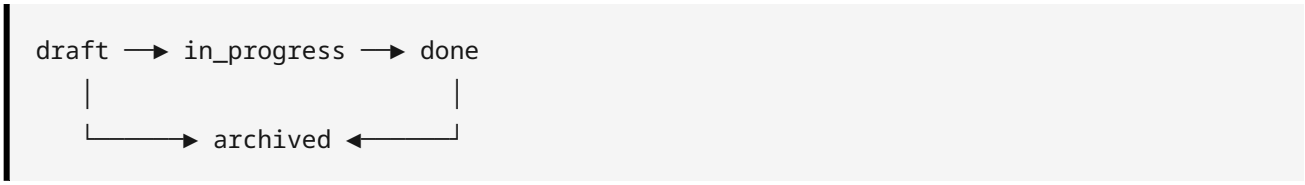
- 2026-11-12: limite de 100 pedidos por chamada, conforme a API.
```

Cada seção tem um papel:

Seção	Responde	Leitor
`Goal`	por que a mudança existe e o que fica de fora	você, ao aprovar
`Behavior`	como o binário funciona para a pessoa que usa	as skills de implementação e de teste
`Design`	onde e como o código deve ser feito	as skills de implementação
`Surface`	o que entra, muda ou sai da superfície	a revisão e o fechamento
`Tasks`	uma entrega por linha, com a tag da skill	o agente, para saber a próxima skill
`Tests`	um teste de aceite por linha, como comando e resultado	a termfy-test
`Notes`	dúvidas e combinados, com data	todos

Cada item de Surface aparece pelo nome em ao menos uma tarefa ou teste, para relacionar a alteração prevista à sua implementação ou verificação.

5. O ciclo de uma spec



Status	Quando	O que o agente faz
`draft`	a spec existe, mas ainda tem dúvidas	entrevista e escreve
`in_progress`	Behavior e Design aprovados	implementa e marca `[x]` em cada tarefa e teste entregues
`done`	todas as tarefas e testes marcados	aplica a Surface às tabelas do `CLISPEC.md` e registra a data nas Notes
`archived`	a spec saiu do plano	mantém o arquivo como histórico

O status muda na spec e na linha do índice do `CLISPEC.md` ao mesmo tempo. Uma spec parada por algo externo continua `in_progress`, com o motivo datado em Notes.

Ao fechar a 0001, o `CLISPEC.md` ganha a linha de `orders list` em Commands, `orders_list` em MCP, `shop.baseUrl` em Settings e uma subseção Credentials com `shop.token`. Nas Notes entra:

```
- 2026-11-14: spec 0001 (Listar pedidos) closed.
```

6. A conferência manual

Nenhum script compara os documentos com o código. A conferência é uma leitura, feita pela skill que escreve e de novo na termfy-review:

- as seções estão na ordem do contrato;
- a linha de identidade bate com o `package.json`;
- há uma linha no índice para cada arquivo de `specs/`, com o mesmo título e status;
- cada item de Surface é citado em tarefa ou teste;
- depois do fechamento, cada command, tool, key e credencial do código aparece nas tabelas do `CLISPEC.md`.

Para adotar código que já existe sem spec, peça uma spec de adoção. Ela lista esse código na Surface e é fechada logo em seguida, o que leva as entradas ao `CLISPEC.md`.

Próximo passo

Com a spec em `in_progress`, a primeira tarefa pendente indica a skill. Na 0001, a primeira é `#termfy-config`, seguida do client HTTP. O capítulo Integrar uma API mostra esse trabalho com uma API real.

Integrar uma API

Muitos binários existem para conversar com um serviço externo. Este capítulo mostra o caminho completo com um exemplo concreto: o `mycli` passa a operar a API de WhatsApp da uazapi, para avisar os clientes da loja sobre os pedidos. Você vê a especificação virar spec, o token ir para o cofre, o client HTTP centralizar as requests e os commands receberem a instância por parâmetro.

Etapa	Skill	Resultado
ler a documentação da API	termfy-api-import	Surface e Tests da spec preenchidos
URL e token	termfy-config	key `uazapi.baseUrl` e credencial `uazapi.admintoken`
client HTTP	termfy-api	`src/api/uazapi.ts` com timeout, retry e erros
commands	termfy-cli	`instances list`, `instances status`, `messages send`
tools e tela	termfy-mcp e termfy-tui	leitura pelo agente e aba Instances

O contrato da uazapi

A API usa dois tokens com papéis diferentes. Entender isso antes de pedir qualquer código evita guardar segredo demais no cofre.

Header	Para que serve	Endpoints usados no exemplo
`admintoken`	administrar o servidor: listar e criar instâncias	`GET /instance/all`
`token`	operar uma instância (um número de WhatsApp)	`GET /instance/status`, `POST /instance/connect`, `POST /send/text`

O servidor fica em `https://{subdomain}.uazapi.com`, com `free` ou `api` no lugar do subdomínio. A resposta de `GET /instance/all` traz, para cada instância, `id`, `name`, `status` e o `token` dela:

```
[
  {
    "id": "r183e2ef9597845",
    "name": "instancia-1",
    "token": "abc123xyz",
    "status": "connected",
    "profileName": "Meu WhatsApp"
  },
  {
    "id": "r283e2ef9597846",
    "name": "instancia-2",
    "token": "def456xyz",
    "status": "disconnected"
  }
]
```

Daí sai a escolha do exemplo: só o `admintoken` vai para o cofre. Quando um command recebe `--instance instancia-1`, o client lista as instâncias, acha o token pelo nome e usa esse token em memória, sem gravá-lo. Trocar o token de uma instância no painel não exige nenhuma ação no binário.

1. Importar a documentação

Peça ao agente:

> Use a skill termfy-api-import com a especificação `https://docs.uazapi.com/openapi-bundled.json`. Quero listar instâncias, ver o status e conectar uma instância e enviar texto. O admintoken fica no cofre e cada command recebe a instância por –instance.

A skill lê a especificação OpenAPI, separa as operações pedidas e preenche a spec com o que encontrou. Ela não implementa nada: o resultado é texto na spec, para você aprovar.

```
## Surface

- Adiciona a setting `uazapi.baseUrl`: `string`, padrão
  `https://free.uazapi.com`, URL do servidor uazapi.
- Adiciona a credencial `uazapi.admintoken`: usada por `instances list`,
  `instances status`, `instances connect`, `messages send` e
  `instances_list`; painel da uazapi.
- Adiciona o command `instances list`: List the instances on the server;
  sem options; `data` `{ count, instances[] }`; exit codes 3, 4 e 5.
- Adiciona o command `instances status`: Show whether one instance is
  connected; `--instance <name>`; `data` `{ name, connected, loggedIn,
  phone }`; exit codes 2, 4, 5 e 6.
- Adiciona o command `messages send`: Send a text message;
  `--instance <name>`, `--number <phone>`, `--text <message>`; `data`
  `{ instance, id, chatId, status }`; exit codes 2, 4, 5 e 6.
- Adiciona a tool `instances_list`: lista instâncias sem tokens; efeito
  `network`.
```

Neste exemplo, `messages send` fica restrito à CLI, porque o envio altera um serviço externo. As tools MCP apenas consultam as instâncias e você executa o envio pelo comando.

2. URL e token

A termfy-config declara a key em `src/config/settings.ts`, junto das keys do esqueleto:

```
{
  key: "uazapi.baseUrl",
  type: "string",
  default: "https://free.uazapi.com",
  description: "Base URL of the uazapi server, such as https://free.uazapi.com or
https://api.uazapi.com.",
},
```

Você grava o token uma vez, sem que ele passe por argumento nem fique no histórico do shell:

```
mycli config set uazapi.baseUrl https://api.uazapi.com
mycli credentials set uazapi.admintoken
```

Sem `--stdin`, o `credentials set` pede o valor no terminal, sem eco. Num CI, use `--stdin` com o segredo vindo do ambiente.

3. O client HTTP

A termfy-api cria `src/api/uazapi.ts` como o único arquivo que chama `fetch`. Commands, tools e a tela chamam as funções dele, então os três modos falam com a API do mesmo jeito.

A base é uma função de request que aplica o contrato do client: timeout por tentativa, retry só para leitura e erro sem segredo.

```
const TIMEOUT_MS = 5_000;
const READ_ATTEMPTS = 3;

type Auth = { admintoken: string } | { token: string };

async function request(method: "GET" | "POST", path: string, auth: Auth, body?:
unknown): Promise<unknown> {
  const url = new URL(path, baseUrl());
  const attempts = method === "GET" ? READ_ATTEMPTS : 1;
  let last: AppError | undefined;
  for (let attempt = 1; attempt <= attempts; attempt++) {
    let response: Response;
    try {
      response = await fetch(url, {
        method,
        redirect: "error",
        signal: AbortSignal.timeout(TIMEOUT_MS),
        headers: { accept: "application/json", ...(body ? { "content-type":
"application/json" } : {}), ...auth },
        body: body ? JSON.stringify(body) : undefined,
      });
    } catch {
      last = error.network(`uazapi did not answer ${method} ${path}.`, { service:
"uazapi", method, path, status: null, attempts: attempt });
      continue;
    }
    const text = (await response.text()).slice(0, 8 * 1024);
    if (response.ok) return JSON.parse(text);
    last = error.network(`uazapi answered ${response.status} for ${method} $
{path}.`, { service: "uazapi", method, path, status: response.status, attempts:
attempt });
    if (response.status < 500) break;
  }
  throw last!;
}
```

Quatro escolhas aparecem no código:

- **POST tem uma tentativa só.** Se a rede cair depois que o servidor enviou a mensagem, repetir enviaria duas vezes.
- **4xx não repete.** Um 401 por token errado não melhora na segunda tentativa.
- **`redirect: "error"`.** Um redirect poderia levar o token para outra origem.

- **`error.network` sai com exit code 5** e leva método, caminho e status nos `details`. O token não entra na mensagem.

Sobre essa base ficam as funções de domínio. A resolução da instância por nome é o centro do exemplo:

```
function admin(): Auth {
  return { admintoken: getCredential("uazapi.admintoken") };
}

/** Token of one instance, looked up by name. NOT_FOUND (6) when absent. */
async function instanceToken(name: string): Promise<string> {
  const found = parseInstances(await request("GET", "/instance/all",
admin())).find((i) => i.name === name);
  if (!found) throw error.notFound(`Instance "${name}" not found. Run: mycli
instances list`, { instance: name });
  return found.token;
}

/** Lists the server instances, without their tokens. */
export async function listInstances(): Promise<Instance[]> {
  return parseInstances(await request("GET", "/instance/all",
admin())).map(({ token: _token, ...rest }) => rest);
}

/** Sends a text message. Never retried: a repeat would send twice. */
export async function sendText(name: string, number: string, text: string):
Promise<SentMessage> {
  const data = await request("POST", "/send/text", { token: await
instanceToken(name) }, { number, text });
  return toSentMessage(data, number);
}
```

`getCredential` lança o erro de credencial (exit code 4) quando o token não está no cofre, antes de qualquer request. `listInstances` remove o `token` de cada item: nenhum command, tool ou tela recebe o segredo de uma instância.

4. Commands com a instância por parâmetro

A termfy-cli cria a família `instances` e o command `messages send`. A option `--instance` é declarada uma vez e reaproveitada:

```
export const instanceOption: OptionDefinition = {
  name: "--instance",
  aliases: ["-i"],
  argument: { name: "name" },
  description: "Instance name, as shown by `instances list`.",
};

export function requireInstance(options: Record<string, unknown>): string {
  const name = options.instance;
  if (typeof name !== "string" || !name) throw error.usage("Missing --instance
<name>. Run: mycli instances list");
  return name;
}

const status: Command = {
  name: "status",
  description: "Show whether one instance is connected and logged in.",
  options: [instanceOption],
  exitCodes: [ExitCode.USAGE, ExitCode.NOT_FOUND, ExitCode.CREDENTIALS,
ExitCode.NETWORK_OR_API],
  examples: [`${manifest.bin} instances status --instance sales`, `${manifest.bin}
instances status -i sales --json`],
  run: async (ctx) => {
    const s = await instanceStatus(requireInstance(ctx.options));
    return {
      data: s,
      text: () => table([["Instance", s.name], ["Connected", s.connected ? "yes" :
"no"], ["Phone", s.phone ?? "-"]]),
    };
  },
};
```

O command não sabe nada de HTTP. Ele valida a entrada, chama o client e devolve `data` para o `--json` e `text` para o terminal. O registry cuida do envelope, da ajuda e do exit code.

Você lista as instâncias e usa o nome escolhido nas chamadas seguintes:

```
mycli instances list
mycli instances status --instance instancia-1 --json
mycli messages send -i instancia-1 --number 5511999999999 --text "Pedido enviado"
```

```
● instancia-1      connected     Meu WhatsApp
○ instancia-2      disconnected
```

```
{
  "ok": true,
  "command": "mycli instances status",
  "data": {
    "name": "instancia-1",
    "connected": true,
    "loggedIn": true,
    "phone": "5511999999999"
  }
}
```

E os erros chegam com o código certo:

Situação	Mensagem	Exit code
faltou `--instance`	`Missing --instance <name>. Run: mycli instances list`	2
token não está no cofre	`Credential "uazapi.admintoken" not found. Save it with: mycli credentials set uazapi.admintoken`	4
`admintoken` inválido	`uazapi answered 401 for GET /instance/all.`	5
instância com outro nome	`Instance "vendas" not found. Run: mycli instances list`	6

Um script decide o que fazer pelo exit code, sem interpretar a mensagem.

5. Tools e tela

A termfy-mcp registra `instances_list` chamando a mesma `listInstances`. Falha prevista vira `isError: true`, com a mensagem do `AppError`:

```
server.registerTool(
  "instances_list",
  {
    title: "List uazapi instances",
    description:
"Use when the user asks which WhatsApp instances exist or which are connected.
Returns name, status and profile, without tokens.",
    inputSchema: {},
  },
  async () => {
    try {
      const instances = await listInstances();
      const d = { count: instances.length, instances };
      return { content: [{ type: "text", text: JSON.stringify(d, null, 2) }],
structuredContent: d };
    } catch (e) {
      const message = e instanceof Error ? e.message : String(e);
      return { content: [{ type: "text", text: message }], isError: true };
    }
  },
);
```

A termfy-tui monta a aba Instances com `listPanel`, carregando a lista no `onEnter` e mostrando o status da instância selecionada. Ao criar a aba, a skill acrescenta `instances` aos valores aceitos de `tui.initialTab`, para que você possa abrir a TUI direto nela.

6. Testar sem a API real

Nenhum teste chama a uazapi de verdade. A termfy-api sobe um servidor local com `Bun.serve`, que responde como a especificação. Depois, aponta `uazapi.baseUrl` para ele. HTTP sem TLS é aceito só em `127.0.0.1` e `localhost`, justamente para esses testes.

```
const server = Bun.serve({
  port: 0,
  fetch(req) {
    const url = new URL(req.url);
    if (url.pathname === "/instance/all") {
      if (req.headers.get("admintoken") !== "admin-test") return new Response("{}",
{ status: 401 });
      return Response.json([{ id: "1", name: "sales", token: "inst-test", status:
"connected" }]);
    }
    if (url.pathname === "/send/text") {
      sends++;
      return new Response("{}", { status: 500 });
    }
    return new Response("{}", { status: 404 });
  },
});
```

Com esse servidor, os testes de aceite da spec viram verificações diretas: o `admintoken` vai no header certo, a lista sai sem tokens, um nome desconhecido sai com 6 e um 500 no envio resulta em uma única request (`sends === 1`), sem reenvio.

Próximo passo

Com as tarefas e os testes marcados, a spec fecha e o `CLISPEC.md` ganha a superfície nova. O capítulo Guardar dados mostra o caso de um binário que precisa guardar informações entre execuções.

Guardar dados

O esqueleto guarda preferências em `settings.json` e segredos no cofre. Um banco SQLite entra quando a spec descreve registros que precisam continuar disponíveis entre execuções, como notas criadas pelo CLI. Este capítulo mostra esse caminho com a tabela de notas que o starter kit traz como stub em `template/db/`.

Quando pedir um banco

Você precisa guardar	Onde fica	Skill
uma preferência (tema, URL, formato)	`settings.json`	termfy-config
um token ou senha	`credentials.json`, cifrado	termfy-config
registros do domínio que crescem com o uso	`~/.<bin>/data.db`	termfy-database

Se a necessidade aparecer no meio do trabalho, volte à spec antes. A skill de banco recusa criar tabelas que a spec não lista, porque a tabela faz parte da superfície do binário.

1. A spec com a tabela

Na entrevista da termfy-spec, diga que o binário precisa guardar os registros. A Surface ganha a tabela com colunas e tipos:

```
## Surface

- Adiciona a tabela `notes`: `id INTEGER` chave primária,
  `title TEXT NOT NULL`, `body TEXT NOT NULL` (padrão vazio),
  `created_at TEXT NOT NULL`.
- Adiciona o command `notes add`: Save a note; `title`, `--body <text>`;
  `data` `{ id, title, body, createdAt }`; exit codes 2, 3 e 7.

## Tasks

- [ ] Migration `0001-create-notes` e repositório `src/db/notes.ts`
  `#termfy-database`
- [ ] Command `notes add` `#termfy-cli`
```

2. Peça a implementação

> Use a skill termfy-database para a spec 0003. Copie os stubs de template/db e crie a tabela notes.

A skill copia `database.ts` e o registro de migrations para `src/db/`, cria a migration da spec e o repositório, liga o repositório aos commands e escreve os testes.

3. A migration

Cada mudança de schema é um arquivo novo em `src/db/migrations/`, numerado em sequência:

```
import type { Migration } from "./index.ts";

export const createNotes: Migration = {
  version: 1,
  name: "create_notes",
  up: `
    CREATE TABLE notes (
      id INTEGER PRIMARY KEY,
      title TEXT NOT NULL,
      body TEXT NOT NULL DEFAULT '',
      created_at TEXT NOT NULL
    ) STRICT;
    CREATE INDEX notes_created_at ON notes (created_at);
  `,
};
```

O registro `src/db/migrations/index.ts` importa cada arquivo de forma estática. Isso é o que faz o SQL entrar no binário compilado: o executável roda em máquinas onde o repositório não existe.

```
import { createNotes } from "./0001-create-notes.ts";

export const migrations: readonly Migration[] = checkMigrations([createNotes]);
```

`checkMigrations` recusa versão repetida, buraco na sequência e SQL vazio quando o módulo carrega. Um erro desses quebra todos os testes na hora, antes de chegar à release.

O schema evolui com migrations e backup do arquivo existente:

- **Migration publicada não muda.** Precisa de uma coluna nova? Crie a `0002` com `ALTER TABLE`.

- **Não há rollback.** Antes de migrar um banco que já tem schema, o binário copia o arquivo para `data.db.v<versão>.bak`.
- **Banco mais novo que o binário é recusado** com exit code 3. Um binário antigo não grava sobre um schema que ele não conhece.

4. O repositório

O SQL fica só no repositório. Commands, tools e telas chamam as funções dele:

```
/** Inserts a note and returns it with its id. */
export function addNote(db: Database, input: { title: string; body?: string }, now
= new Date()): Note {
  const row = db.query<NoteRow, { title: string; body: string; createdAt: string }>(
    "INSERT INTO notes (title, body, created_at) VALUES ($title, $body, $createdAt)
RETURNING id, title, body, created_at",
  ).get({ title: input.title, body: input.body ?? "", createdAt:
now.toISOString() });
  return toNote(row!);
}

/** Returns one note or raises NOT_FOUND. */
export function getNote(db: Database, id: number): Note {
  const row = db.query<NoteRow, { id: number }>("SELECT id, title, body, created_at
FROM notes WHERE id = $id").get({ id });
  if (!row) throw new AppError(ExitCode.NOT_FOUND, `Note ${id} not found.`);
  return toNote(row);
}
```

Os parâmetros são nomeados (`$title`), sem concatenar valores no SQL. As linhas viram objetos em camelCase (`createdAt`), que é a forma que o `--json` devolve. Linha ausente vira `NOT_FOUND` , exit code 6.

5. O command usa o repositório

O command abre o banco no `run` e fecha no `finally` :

```
run: (ctx) => {
  const db = openDatabase();
  try {
    const note = addNote(db, { title: ctx.args[0]!, body: ctx.options.body as
string | undefined });
    return { data: note, text: () => `Saved note ${note.id}.` };
  } finally {
    db.close();
  }
},
```

`openDatabase()` abre `~/.<bin>/data.db`, liga foreign keys, usa WAL e aplica as migrations pendentes. Você não precisa rodar nenhum passo de instalação: o primeiro command que abre o banco cria o arquivo.

Se a spec pedir, a skill também copia o command `db`, com `db status` (versão do schema e migrations pendentes) e `db migrate`.

6. Testes em memória

Cada teste abre o seu banco em memória, sem mexer em `~/.<bin>/`:

```
describe("notes repository", () => {
  let db: Database;
  beforeEach(() => { db = openDatabase(":memory:"); });
  afterEach(() => db.close());

  test("adds, lists, reads and removes", () => {
    const note = addNote(db, { title: "First" }, new
Date("2026-11-10T12:00:00.000Z"));
    expect(listNotes(db)).toEqual([note]);
    removeNote(db, note.id);
    expect(listNotes(db)).toEqual([]);
  });

  test("a database newer than the binary is refused", () => {
    db.run("PRAGMA user_version = 999");
    expect(() => migrate(db)).toThrow(AppError);
  });
});
```

Por fim, a skill roda `bun run build` e executa o binário de `dist/` numa pasta vazia, com `<BIN>_HOME` temporário. Se o banco for criado ali, as migrations estão embutidas.

Próximo passo

O capítulo Testar, revisar e publicar fecha o ciclo: testes nos três modos, revisão por gravidade e release.

Testar, revisar e publicar

Uma spec só fecha quando os testes de aceite passam. Uma versão só sai quando as specs dela estão fechadas e a revisão não encontrou problema grave. Este capítulo mostra as três skills que cuidam disso: termfy-test, termfy-review e termfy-release.

1. Testes nos três modos

Peça:

> Use a skill termfy-test para cobrir os testes da spec 0001.

A skill lê a seção `## Tests` da spec e escreve um teste para cada item, no modo certo. O starter kit já traz um exemplo de cada padrão em `tests/`.

Modo	Como o teste roda	Arquivo de referência
CLI	o binário num subprocesso, com pasta de dados temporária	`tests/cli.test.ts`
MCP	uma conversa real pelo stdio, com o `Client` do SDK	`tests/mcp.test.ts`
TUI	a interface no renderer de teste do OpenTUI, conferida pelo frame	`tests/tui.test.ts`
config	settings e cofre numa pasta temporária	`tests/config.test.ts`

CLI num subprocesso

O teste roda `src/main.ts` como você rodaria e confere exit code, stdout e stderr. A variável `<BIN>_HOME` aponta para uma pasta temporária. Assim o teste não mexe no seu `~/.mycli/`:

```
async function run(args: string[], stdin?: string) {
  const proc = Bun.spawn(["bun", "run", "src/main.ts", ...args], {
    cwd: join(import.meta.dir, ".."),
    env: { ...process.env, [HOME_VARIABLE]: dir, NO_COLOR: "1" },
    stdin: stdin === undefined ? "ignore" : new Response(stdin),
    stdout: "pipe",
    stderr: "pipe",
  });
  const [stdout, stderr] = await Promise.all([new Response(proc.stdout).text(), new
Response(proc.stderr).text()]);
  return { code: await proc.exited, stdout, stderr };
}

test("config get of an unknown key exits with 3", async () => {
  const r = await run(["config", "get", "does.not.exist", "--json"]);
  expect(r.code).toBe(3);
  expect(JSON.parse(r.stdout).error.name).toBe("CONFIG");
});
```

MCP pelo stdio

O teste abre o servidor como um cliente de agente abriria e chama as tools pelo protocolo:

```
client = new Client({ name: "test", version: "0.0.0" });
await client.connect(new StdioClientTransport({
  command: "bun",
  args: ["run", "src/main.ts", "mcp"],
  env: { ...(process.env as Record<string, string>), [HOME_VARIABLE]: dir },
  stderr: "ignore",
}));

test("credentials_status does not expose values", async () => {
  const r = await client.callTool({ name: "credentials_status", arguments: {} });
  expect(Object.keys(r.structuredContent as object)).not.toContain("values");
});
```

Esse teste pega um problema comum: um `console.log` esquecido escreve texto em stdout, quebra o JSON-RPC e a conversa com o agente cai.

TUI pelo frame

O renderer de teste desenha a interface em memória. O teste envia teclas e confere o texto do frame:

```
const setup = await createTestRenderer({ width: 120, height: 32 });
mountTui(setup.renderer, {});
await setup.renderOnce();
expect(setup.captureCharFrame()).toContain("Overview");
```

Com cada teste verde, a skill marca `[x]` no item correspondente de `## Tests`. Um item só é marcado depois que o teste roda.

2. Revisão por gravidade

Peça:

> Use a skill termfy-review para revisar a entrega da spec 0001.

A skill lê o diff, a spec e o código e roda as verificações. Ela não edita nada: relata cada achado com gravidade, arquivo, linha, efeito e correção sugerida.

Nível	Quando	Exemplo
P0	segredo real exposto ou perda de dados imediata	valor do cofre devolvido por uma tool
P1	fluxo principal quebrado ou proteção essencial ausente	texto em stdout interrompe o JSON-RPC
P2	comportamento errado localizado ou contrato descumprido	exit code errado; option sem description
P3	apresentação ou manutenção, sem mudar o resultado	comentário em português no código

Um relatório tem esta forma:

```
1. [P2] src/cli/commands/orders.ts:31 | Sem shop.token, orders list sai com 5 em vez de 4. | Deixar o erro de getCredential subir sem convertê-lo em erro de rede.
2. [P3] src/api/shop.ts:12 | Comentário em português. | Traduzir para inglês.

Testes: bun run validate passou (62 testes).
Conferência manual: orders_list está no código e falta na Surface da spec.
```

Você escolhe o que corrigir. Depois das correções, peça uma revisão nova até não restar P0, P1 nem P2.

3. Fechar a spec

Com todas as tarefas e testes marcados, peça à termfy-spec para fechar a spec. O agente aplica a Surface às tabelas do `CLISPEC.md`, troca o status para `done` e registra a data. O capítulo Descrever e especificar mostra o ciclo.

4. Publicar

Peça:

> Use a skill termfy-release para publicar a versão 0.2.0 do mycli.

A skill segue esta ordem:

1. Confere `git status`, a branch e o índice de specs. Uma spec `in_progress` com código no repositório indica entrega incompleta.
2. Atualiza `version` no `package.json` e move as entradas de `Unreleased` para a versão nova no `CHANGELOG.md`.
3. Roda `bun run opencli` se o registry mudou e `bun run validate`.
4. Faz o commit `release: mycli v0.2.0` e cria a tag `v0.2.0`.
5. Envia a `main` e só essa tag.
6. Acompanha o workflow `release.yml`, que compila os cinco executáveis e gera o `SHA256SUMS`.
7. Baixa o executável da sua plataforma, confere o checksum e roda `--version`.

O changelog segue o formato Keep a Changelog e pode citar as specs:

```
## [0.2.0] - 2026-11-14

### Adicionado

- Command `orders list` e tool `orders_list` para consultar pedidos
  (spec 0001).
```

Uma tag publicada não se move. Se o workflow falhar, a correção sai numa versão nova.

5. Conectar o agente

Com o binário instalado no `PATH`, a termfy-mcp-client grava a configuração no cliente de agente que você usa e confere a conexão. A configuração mínima é a mesma que a aba Overview sugere:

```
{ "command": "mycli", "args": ["mcp"] }
```

Próximo passo

A Parte 3 traz a referência completa: CLI, TUI, MCP, configuração, publicação e uma página por skill.

CLI

Registro declarativo

Cada comando é um objeto `Command` em `src/cli/commands/<name>.ts`:

```
import type { Command } from "../registry.ts";
import { ExitCode } from "../../app/exit-codes.ts";

export function createOrdersCommand(listOrders: Command): Command {
  return {
    name: "orders",
    description: "Query shop orders.",
    subcommands: [listOrders],
    exitCodes: [ExitCode.CREDENTIALS, ExitCode.NETWORK_OR_API],
  };
}
```

O registro reúne a definição do comando e a função `run`, que executa seu comportamento. A definição inclui os nomes e argumentos aceitos, as opções e os subcomandos. Ela também documenta exemplos e códigos de saída para os consumidores abaixo:

Consumidor	Arquivo	Saída
parser	`src/cli/parser.ts`	argumentos e opções validados
ajuda	`src/cli/help.ts`	`--help` da raiz e de cada comando
OpenCLI	`src/cli/opencli.ts`	`--opencli` e `docs/opencli.json`
completion	`src/cli/completion.ts`	scripts de autocompletar
documentação	`src/skill/index.ts`	referência gerada no `--skill cli`

`OptionDefinition.default` é metadado para documentação; aplique o padrão em `run`, pois o parser não o preenche em `ctx.options`.

`OptionDefinition` exige `description` no tipo TypeScript. O parser resolve opções pelo nome e pelos aliases; não testa a presença da descrição.

Argumentos e opções

- `--key value` e `--key=value` são equivalentes.
- Aliases curtos (`-h`, `-V`, `-y`) existem, mas não se agrupam.
- `--` encerra a leitura de flags. O parser ainda pode reconhecer um subcommand enquanto nenhum argumento posicional tiver sido consumido.
- Opções globais (`--help`, `--version`, `--json`, `--skill`, `--opencli`, `--no-color`) valem em qualquer posição.
- Valores fora da lista `values`, argumentos obrigatórios ausentes e opções desconhecidas encerram com código 2 e uma mensagem que aponta o `--help`.

O contrato de `run`

```
import type { Command } from "../registry.ts";
import { table } from "../format.ts";
import { getCredential } from "../../config/credentials.ts";
import { ExitCode } from "../../app/exit-codes.ts";
import { manifest } from "../../app/manifest.ts";

type Order = { id: string; status: string };
type FetchOrders = (token: string) => Promise<Order[]>;

export function createListOrders(fetchOrders: FetchOrders): Command {
  return {
    name: "list",
    description: "List shop orders.",
    exitCodes: [ExitCode.CREDENTIALS, ExitCode.NETWORK_OR_API],
    examples: [`${manifest.bin} orders list`],
    run: async () => {
      const token = getCredential("shop.token");
      const orders = await fetchOrders(token);
      return {
        data: { count: orders.length, orders },
        text: () => table(orders.map((order) => [order.id, order.status])),
      };
    },
  };
}
```

Os exemplos usam imports relativos a `src/cli/commands/`. Ao reuni-los em `src/cli/commands/orders.ts`, mantenha cada import uma só vez. Implemente `fetchOrders` no projeto, lançando `error.network()` quando a API falhar. Depois, registre `createOrdersCommand(createListOrders(fetchOrders))`.

`data` contém o resultado usado em `--json` e `text` formata esse resultado para você ler no terminal. A camada de entrada escolhe o formato da saída. O comando devolve o resultado e lança `AppError` quando algo previsto falha, deixando a montagem do JSON e o encerramento do processo para essa camada.

Saída em texto e em JSON

Em texto, o resultado vai para stdout e os erros para stderr, em uma linha com o código e o nome entre colchetes:

```
error Credential "shop.token" not found. Save it with: credentials set shop.token [4 CREDENTIALS]
```

Em JSON, resultados e erros vão para stdout; progresso e avisos permanecem em stderr. Uma exceção gera este envelope:

```
{ "ok": false, "command": "mycli orders list", "error": { "code": 4, "name": "CREDENTIALS", "message": "..." } }
```

`doctor` devolve o diagnóstico em `data` e pode sair com 3 mantendo `ok: true` no envelope. Nesse caso, consulte `data.ok`.

`output.format = json` em settings.json torna o JSON padrão. `--json` tem prioridade sobre essa key.

Códigos de saída

Código	Nome	Quando
0	`SUCCESS`	operação concluída
1	`GENERIC_ERROR`	falha inesperada
2	`USAGE`	argumentos ou opções inválidos; binário sem argumentos fora de TTY
3	`CONFIG`	settings.json inválido, valor recusado, chave ausente em `config get`

Código	Nome	Quando
4	`CREDENTIALS`	cofre ilegível, chave mestra ausente ou credencial não encontrada
5	`NETWORK_OR_API`	rede ou resposta de API fora do esperado
6	`NOT_FOUND`	recurso inexistente
7	`PERMISSION`	permissão negada no sistema de arquivos
8	`NO_TTY`	TUI sem TTY
9	`MCP`	servidor MCP falhou ou perdeu o transporte
130	`INTERRUPTED`	cancelado pelo usuário

A fonte é `src/app/exit-codes.ts`. Uma família nova entra lá e em `src/app/errors.ts`, com um atalho `error.<family>()`. Números não se reutilizam.

Commands de apoio

`<bin> doctor` verifica o runtime e o terminal, além dos arquivos de configuração e do cofre. O diagnóstico também identifica a versão e a plataforma, mostrando ✓, ! ou ✗ em cada item. Uma falha resulta no código 3, mas os avisos mantêm a saída 0. O retorno é `{ checks, ok }`, com `name`, `status` e `detail` em cada check, e está disponível também pela tool MCP `doctor`.

`<bin> completion bash|zsh|fish` gera o script de autocompletar a partir do registry, com subcommands, options e valores enumerados. Para instalar na sessão Zsh com saída texto, use `eval "$(mycli completion zsh)"`; no Fish, `mycli completion fish | source`. Se `output.format` for `json`, restaure `text` ou extraia `data.script` antes de carregar o script. `<bin> --skill reference` devolve só a referência gerada do registry, para agentes que já conhecem o método e querem economizar a janela de contexto do modelo.

OpenCLI

A OpenCLI Specification descreve a interface de um CLI em JSON para que você e seus scripts saibam invocá-lo sem ler o código. `mycli --opencli` gera o documento a partir do registro; `bun run opencli` grava `docs/opencli.json`; `bun run opencli:check` falha quando o arquivo diverge do registro. Um agente pode preferir esse JSON ao `--help`: nomes, aliases, valores aceitos e códigos de saída vêm sem formatação para interpretar.

Adicionar um comando

1. Copie `template/command/example.ts` para `src/cli/commands/<name>.ts`.
2. Preencha a definição e implemente `run`.
3. Registre em `src/cli/commands.ts`.
4. Escreva o teste em `tests/cli.test.ts`, que roda o binário em subprocesso e confere stdout, stderr e código de saída.
5. Rode `bun run opencli` e `bun run validate`.
6. Se o comando trouxer conceito novo, escreva um parágrafo em `docs/cli.md`.

A skill `termfy-cli` percorre esses passos com o agente.

TUI

Estrutura da tela

Esquema ilustrativo da disposição, com rótulos traduzidos para leitura:

```
▌ My CLI v<version>  │  Consulta e atualiza pedidos da loja.     linux-x64 • Bun
1.4.4
──────────────────────────────────────────────────────────────────────────────

  1 ⌂ Overview   2 ⚙ Settings   3 ◈ Credentials   4 ≡ Docs   5 i About
┌─ Ambiente ──────────────────────────────┐┌─ Estado ──────────────────────────┐
│ Binário     mycli v<version>            ││ settings.json     • padrões       │
│ Runtime     Bun 1.4.4                   ││ credentials.json  • vazio         │
└─────────────────────────────────────────┘└───────────────────────────────────┘

 settings: padrões · cofre: 0    r recarregar   ? ajuda   q sair      ^K commands
```

`src/tui/index.ts` monta quatro faixas: cabeçalho de uma linha, barra de abas com número e ícone, conteúdo da aba ativa e barra de status. Cada aba é um arquivo em `src/tui/screens/` que exporta um objeto `Screen` com `id`, `title`, `icon`, `description`, `actions` e `mount(ctx)` e, quando precisa, `watch`, `onEnter`, `onLeave`, `onPause`, `onResume` e `onKey`.

O starter kit traz cinco abas:

Aba	Conteúdo
Overview	ambiente, estado da configuração e do cofre, configuração em vigor e próximos passos
Settings	tabela com key, valor e padrão, com painel de detalhe; edita cada key pelo controle do tipo
Credentials	nomes do cofre, origem da master key e total de credenciais; valores não aparecem
Docs	índice dos tópicos da documentação embutida e o documento em Markdown
About	binário, projeto, créditos, padrões seguidos e links, com os mesmos campos do comando `about`

Teclas

Tecla	Ação
`←` `→` `[` `]`	aba anterior ou seguinte

Tecla	Ação
`Tab` `Shift+Tab`	próximo painel ou painel anterior da tela
`1` … `9`	ir para a aba de mesmo número
`Ctrl+K` `:`	abrir a paleta de comandos
`?`	mostrar ou esconder a ajuda
`Esc`	fechar o diálogo ou a ajuda; sem modal aberto, sair
`q` `Ctrl+C`	sair

Dentro de um campo de texto, letras e números vão para o campo e `Esc` sai do campo. Fora dele, `Tab` e `Shift+Tab` movem o foco entre os painéis da tela, sem trocar de aba. O painel em foco usa a borda de destaque.

A tela declara os próprios comandos em `actions`: tecla, descrição, ícone e a função que roda. Eles aparecem na barra inferior antes das globais, no help e na paleta. O clique no item da barra roda a mesma ação. O renderer é destruído em toda saída, devolvendo o terminal ao shell.

Paleta de comandos

`Ctrl+K` ou `:` abre a paleta. Ela busca por nome entre as ações da aba, as entradas `Go to <aba>`, as ações das outras abas (troca a aba e executa), "Reload this screen", a ajuda e a saída. A busca aceita as letras em sequência, sem exigir a palavra inteira.

`↑` `↓` e `j` / `k` movem, `PgUp` / `PgDn` e `Home` / `End` pulam, `Enter` executa e `Esc` fecha. A linha responde ao clique. O item `^K commands`, no canto direito da barra de status, também abre a paleta.

Barra de status e toasts

A barra inferior tem o resumo da tela à esquerda (`ctx.status`), as ações da tela no meio e `^K commands` à direita. No layout compacto ficam só `? help` e `q quit` no meio.

`ctx.notify(message, tone)` mostra um toast no canto inferior direito. No máximo três toasts ficam visíveis; o mais antigo fecha primeiro. Cada toast some em 4 segundos ou com um clique e não recebe teclado. O tom pode ser `info`, `ok`, `warn` ou `error`.

Diálogos

O shell desenha os diálogos e devolve a resposta à tela:

```
const name = await ctx.prompt({ title: "Add credential", label: "Name" });
const ok = await ctx.confirm({ title: "Remove", message: `Remove ${name}?`, danger:
true });
const index = await ctx.choose({ title: "Theme", items: ["terminal", "night",
"day"] });
```

Diálogo	Campos	Interação
`confirm`	`title`, `message`, `confirmLabel?`, `danger?`	`Enter` no botão em foco, `y` e `n`, `←` `→` e `Tab` entre botões, `Esc` cancela
`prompt`	`title`, `label`, `initial?`, `placeholder?`, `secret?`, `validate?`	digitação, paste, `Backspace`, `Ctrl+U` limpa, `Enter` envia por `validate`, `Esc` cancela
`choose`	`title`, `items`, `initial?`	`↑` `↓`, `j` / `k`, `PgUp` / `PgDn`, `Home` / `End`, `Enter`, `Esc`

`confirm` resolve `true` ao aceitar e `false` ao cancelar. `prompt` resolve o texto ou `undefined` no `Esc`. `choose` resolve o índice ou `undefined`. `danger` pinta o botão com a cor de erro. `secret` mostra bolinhas e o valor não chega ao frame. `validate` devolve uma mensagem para manter o diálogo aberto. Um clique fora do diálogo cancela.

Use esses três diálogos em vez de montar uma tela de formulário. A tela não ganha layout, foco nem validação próprios para um ou dois campos.

Mouse

Com `tui.mouse` em `true`, o padrão, a TUI responde ao mouse. O clique numa aba abre a aba, numa linha de tabela ou de lista seleciona e ativa o item, num botão, num item da barra, numa linha da paleta ou da ajuda e num botão de diálogo roda a mesma ação da tecla. A roda rola listas e troca de aba sobre a barra de abas. O item sob o ponteiro recebe destaque e o ponteiro vira mão sobre o que é clicável.

Com o mouse ligado, use `Shift` e arraste para selecionar texto no terminal. Com `tui.mouse` em `false`, o terminal volta a selecionar com o mouse e a TUI fica só com o teclado.

Layout responsivo

`ctx.layout()` devolve três faixas, medidas em colunas: `wide` a partir de 120, `medium` a partir de 80 e `compact` abaixo de 80. O shell remonta a tela quando a faixa muda, então `mount` precisa funcionar nas três. Em `wide` os painéis ficam lado a lado. Em `medium` o painel secundário continua ao lado, com menos largura. Em `compact` a tela empilha em uma coluna, esconde o conteúdo secundário e a aba inativa mostra só o número e o ícone.

Abaixo de 60×16 o shell troca a tela pela mensagem “Terminal too small” até o terminal crescer. A screen não repete essa mensagem.

Arquivos e pausa

`Screen.watch()` devolve os arquivos que a tela lê. Quando outro processo muda um deles, o shell recarrega a tela e mostra o toast “Updated from disk”. Overview, Settings e Credentials observam settings.json, credentials.json e master.key.

Quando a janela do terminal perde o foco, o shell chama `onPause`; ao recuperar, chama `onResume`. A tela para timers e animações em `onPause`. O título da janela mostra `<título> · <aba>`. O teclado Kitty é ligado quando o terminal suporta.

A setting `tui.motion` (boolean, padrão `true`) liga as animações curtas de diálogos e toasts. Com `false`, a interface fica estática.

Ícones

`src/tui/icons.ts` guarda três conjuntos com os mesmos nomes de papel: `nerd` (Nerd Font), `unicode` e `ascii`. Os papéis seguem o significado, como `overview`, `about`, `settings`, `terminal`, `copy`, `refresh`, `ok`, `warn` e `error`.

`tui.icons` escolhe o conjunto. `auto` usa `nerd` no Ghostty, no Kitty, no WezTerm e no Omarchy, `ascii` no console Linux e `unicode` nos demais terminais. As telas leem os ícones de `ctx.icons`, então uma troca de setting muda a TUI inteira.

Design modern unix

O visual segue a geração de ferramentas como lazygit, bottom, gitui, k9s e OpenCode: denso, legível, com hierarquia por cor e peso em vez de espaço em branco.

- Bordas arredondadas com o título na própria borda. Painéis vizinhos sem gap; padding interno de uma célula.
- Rótulos à esquerda, apagados, alinhados pela maior chave; valores à direita em cor plena. Títulos de seção em negrito.
- Uma cor de acento para o que está ativo; uma secundária para links e origens externas. Verde, âmbar e vermelho só para estado, com texto ao lado.
- Barra de status com teclas em blocos invertidos.
- A tela atualiza ao receber um resultado novo, com rolagem controlada pelo teclado.
- Glifos de uma célula. Emoji fora, porque a largura varia por terminal.
- Ícones de papel vêm de `ctx.icons`, sem glifo fixo na tela.

As cores vivem em `src/tui/theme.ts`, em três paletas: `terminal` (padrão), `night` (derivada do Tokyo Night) e `day` (clara). Os nomes são de papel: `accent`, `textDim`, `border`, `ok`, `warn`, `error`. No tema `terminal`, cada cor é uma referência à paleta do terminal ou ao texto e fundo padrão, então a TUI acompanha o tema do terminal. Uma tela não escreve hexadecimal: usa os nomes de papel.

Componentes

`src/tui/components.ts` oferece `panel`, `keyValueTable`, `list`, `heading`, `dim`, `spacer`, `button`, `listPanel`, `table`, `stateView`, `spinner`, `progressBar` e `link`.

Componente	Uso
`panel`	caixa com borda, título e `active` para o painel em foco
`keyValueTable`	rótulo apagado e valor alinhados
`table`	cabeçalho que ordena no clique, linha clicável, teclado e `setRows`
`listPanel`	lista com rolagem, seleção e clique na linha
`stateView`	estados de carregando, vazio e erro, com botão opcional
`spinner`	indicador animado com `start`, `stop` e `setLabel`
`progressBar`	barra de 0 a 1 com `set(value, label?)`
`link`	hyperlink OSC 8; o clique copia a URL por OSC 52

`table` devolve `{ box, move, select, selected, sortBy, setRows }`. O primeiro clique na linha seleciona. O segundo clique ativa, assim como o `Enter`. `setRows` troca as linhas e preserva o índice quando ele ainda existe. `listPanel` devolve `{ box, move, select, selected }`; `move(delta)` muda o destaque sem chamar `onSelect`, `select(index)` também chama e `selected()` devolve -1 na lista vazia.

`spacer(renderer, height)` é a exceção que não recebe tema. `button` recebe `key`, `label`, `icon` e `onPress`. O clique chama `onPress`. A linha de `listPanel` e a de `table` também respondem ao clique. Para um elemento clicável próprio, use `clickable` de `src/tui/pointer.ts`, que também pede o ponteiro de mão ao terminal. Um componente novo entra nesse arquivo e fica disponível para todas as telas.

Criar uma aba

1. Copie `template/screen/example.ts` para `src/tui/screens/<id>.ts`.

2. Monte o conteúdo com os componentes e informações de `src/config/` ou dos coletores já usados pelo CLI. Leia `ctx.layout()` e componha para as três faixas.
3. Declare o `icon` da aba e as `actions` da tela, com tecla, descrição e a função que roda.
4. Ligue todo elemento clicável à mesma função de uma ação, com `button` e com a linha de `table` ou de `listPanel`.
5. Use `ctx.status` para o resumo da barra, `ctx.notify` para o resultado de uma ação e `ctx.confirm`, `ctx.prompt` e `ctx.choose` para perguntar.
6. Declare `watch()` para os arquivos lidos. Inicie trabalho em `onEnter`, pare em `onLeave` e em `onPause`, retome em `onResume`.
7. Registre em `src/tui/screens/index.ts`. A ordem da lista é a ordem das abas.
8. Se o `id` puder ser aba inicial, acrescente-o em `tui.initialTab` e em `--tab`.
9. Escreva o teste, com tecla, clique e um frame por faixa de largura.

A skill `termfy-tui` traz os cinco padrões de tela e a checklist de qualidade em `skills/termfy-tui/references/`.

Testar sem terminal

O OpenTUI tem um renderer de teste. `tests/tui.test.ts` monta a interface em memória, envia teclas e confere o frame como texto:

```
import { expect } from "bun:test";
import { createTestRenderer } from "@opentui/core/testing";
import { mountTui } from "../src/tui/index.ts";

const setup = await createTestRenderer({ width: 120, height: 32 });
const closed = mountTui(setup.renderer, {});
await setup.renderOnce();
expect(setup.captureCharFrame()).toContain("Environment");
setup.mockInput.pressKey("2");
await setup.renderOnce();
expect(setup.captureCharFrame()).toContain("Settings");
setup.mockInput.pressKey("q");
await closed;
```

Capture um frame em 60×16, 80×24 e 120×32 para cobrir as três faixas. `mountTui` aceita qualquer renderer; `startTui` cria o renderer real. Essa separação é o que permite testar a TUI em CI. `setup.mockMouse` envia clique, hover e roda, para conferir o caminho de mouse da mesma ação.

Os diálogos também entram no teste: aceite e negue um `confirm`, envie um valor recusado por `validate` e capture o frame de um `prompt` com `secret` para conferir a ausência do valor.

A skill `termfy-tui` traz o guia de design completo e a documentação do OpenTUI em `references/opentui/`.

MCP

O servidor embutido

`mycli mcp` conecta um `McpServer` ao transporte stdio. A partir daí, stdout pertence ao JSON-RPC e qualquer diagnóstico vai para stderr. O processo só termina quando o cliente fecha o transporte.

Configuração em um cliente:

```
{ "mcpServers": { "mycli": { "command": "mycli", "args": ["mcp"] } } }
```

No Claude Code: `claude mcp add mycli -- mycli mcp`. Em desenvolvimento, sem compilar: `"command": "bun", "args": ["run", "src/main.ts", "mcp"]`.

O que o starter kit expõe

Tipo	Nome	Função
tool	`doctor`	`{ checks, ok }` do diagnóstico, sem gravar arquivos
tool	`about`	identidade, runtime, plataforma, pasta de dados, créditos
tool	`exit_codes`	tabela de códigos de saída
tool	`skill`	documentação embutida, inteira ou por tópico
tool	`config_list`	chaves de settings.json em vigor e o esquema
tool	`config_get`	valor de uma chave
tool	`credentials_status`	arquivo, origem da chave mestra, quantidade e nomes
resource	`mycli://about`	identidade em JSON
resource	`mycli://exit-codes`	códigos em JSON
resource	`mycli://skill`	documentação completa
resource	`mycli://skill/{topic}`	uma seção; o template lista os tópicos

Tipo	Nome	Função
prompt	`operate`	instrui o agente a ler `skill cli` e cumprir uma tarefa pelo CLI com `--json`

Nenhuma tool devolve o valor de uma credencial e nenhuma grava. Escrita continua no CLI. A confirmação depende do comando: `credentials clear` pede confirmação, enquanto `config set` e `credentials rotate-key` não.

`mycli mcp --list` enumera tudo sem abrir o transporte. O servidor mantém um catálogo durante o registro, então a lista e o que o cliente vê são iguais.

Regras para tools

- Nome em `snake_case`, `<grupo>_<acao>`.
- `title`, `description` que diz quando usar e o que devolve, `inputSchema` em Zod com `.describe()` em cada campo.
- Para saída estruturada, use `content` em texto e `structuredContent` com o mesmo objeto. A tool `skill` devolve apenas texto Markdown. Falha prevista devolve `isError: true` com a ação sugerida.
- Tools que gravam ou chamam rede dizem isso na descrição.
- Respostas curtas. Paginação por `limit` e `cursor` quando a lista cresce.
- Tools que consomem APIs leem o token com `getCredential(name)` internamente.

Economia de tokens

O OpenMCP, padrão para converter APIs web em servidores MCP, formaliza algo que vale para qualquer servidor: o agente lê todas as descrições de tool a cada conversa, então cada palavra a mais custa em todas as chamadas. O Termfy aplica isso com uma tool de descoberta (`skill`) em vez de dezenas de tools de documentação, descrições de duas frases e resources para consultar conteúdo sem executar ações.

Adicionar uma tool

1. Copie `template/tool/example.ts` para `src/mcp/tools/<grupo>.ts`.
2. Implemente reutilizando funções de `src/config/` e coletores do CLI.
3. Acrescente ao `catalog.tools` e registre em `registerAll` (`src/mcp/server.ts`).
4. Teste em `tests/mcp.test.ts`, que sobe o binário e conversa pelo `Client` do SDK. Para testar à mão, use `npx @modelcontextprotocol/inspector bun run src/main.ts mcp`.
5. Atualize a tabela em `docs/mcp.md`.

A skill `termfy-mcp` traz o contrato de tool, o guia de testes e a documentação do SDK e da especificação MCP em `references/`.

Configuração e credenciais

Tudo que o binário persiste fica em `~/.mycli/`:

Arquivo	Conteúdo	Permissão
`settings.json`	preferências em texto claro	0600
`credentials.json`	envelope AES-256-GCM com os segredos	0600
`master.key`	chave mestra em hexadecimal, criada na primeira gravação	0600

A pasta é criada com 0700. `MYCLI_HOME` troca a raiz, o que isola testes e CI.

settings.json

Chaves em notação de ponto, com esquema tipado em `SETTINGS_SCHEMA` (`src/config/settings.ts`). O starter kit reconhece `output.format`, `output.colors`, `tui.initialTab`, `tui.theme`, `tui.icons`, `tui.motion`, `tui.mouse` e `telemetry.enabled`.

`tui.initialTab` aceita `overview` (padrão), `settings`, `credentials`, `docs` ou `about`.

`tui.theme` aceita `terminal` (padrão), `night` ou `day`. O `terminal` herda as cores do tema do terminal. `night` e `day` são paletas fixas em RGB.

`tui.icons` aceita `auto` (padrão), `nerd`, `unicode` ou `ascii`. O `auto` usa `nerd` no Ghostty, no Kitty, no WezTerm e no Omarchy, `ascii` no console Linux e `unicode` nos demais terminais. `tui.mouse` é boolean, com `true` padrão; `false` devolve a seleção de texto ao terminal. `tui.motion` também é boolean, com `true` padrão; `false` desliga as animações curtas de diálogos e toasts.

```
mycli config list
mycli config set output.format json
mycli config get tui.theme
mycli config unset telemetry.enabled
mycli config schema
```

`config set` converte o texto para o tipo declarado e recusa valores fora de um `enum` com código 3. Chaves fora do esquema são aceitas como texto, para dar espaço a extensões; para ganhar tipo, padrão e documentação, a chave entra no esquema.

Segredos ficam fora do settings.json: o lugar deles é o cofre.

Editar pela TUI

A aba **Settings** mostra uma tabela com key, valor e padrão e um painel com a descrição, o tipo e os valores aceitos. `Enter` ou `e` edita a key selecionada pelo controle do tipo: `enum` abre o diálogo de escolha, boolean alterna entre os dois valores e texto ou número abre um campo com validação. `x` ou `Delete` volta ao padrão depois de uma confirmação. `r` recarrega.

A gravação é a mesma de `config set` e `config unset`. Uma key `tui.*` vale na próxima abertura da TUI; a tela avisa no toast. `tui.motion` e `tui.theme` seguem a mesma regra.

A aba **Credentials** lista só os nomes do cofre. Os valores ficam fora da tela. O painel ao lado mostra o arquivo, a origem da master key e o total de credenciais. `a` adiciona um nome e depois o valor secreto. `Enter` ou `e` troca o valor. `d` ou `Delete` remove com confirmação. `r` recarrega. O valor digitado não entra no frame, no toast nem no resumo da barra.

credentials.json

O arquivo guarda um envelope:

```
{ "version": 1, "algorithm": "aes-256-gcm", "iv": "...", "tag": "...", "data":
"..." }
```

`data` é o JSON `{ "name": "value" }` cifrado. A tag de autenticação garante que um arquivo editado ou cifrado com outra chave falhe ao decifrar, com código 4.

```
mycli credentials set shop.token               # digita sem eco
printf '%s' "$TOKEN" | mycli credentials set shop.token --stdin
mycli credentials get shop.token               # mascarado
mycli credentials get shop.token --reveal
mycli credentials list
mycli credentials status
mycli credentials remove shop.token
```

O valor não passa por argumento de linha de comando. Em terminal, a digitação é oculta; em pipe, vem pela primeira linha do stdin. Assim ele não entra no histórico do shell nem na lista de processos.

Nomes seguem `service.purpose` em minúsculas: `shop.token`, `openai.api_key`, `smtp.password`.

Chave mestra

32 bytes. Fonte, em ordem:

1. `MYCLI_MASTER_KEY`, 64 caracteres hexadecimais (`openssl rand -hex 32`);
2. `~/.mycli/master.key`, gerado na primeira gravação.

Com a variável definida, o arquivo é ignorado. Em CI e em servidores, use a variável quando não quiser criar uma chave em disco. A rotação é uma exceção: `mycli credentials rotate-key` exige um cofre existente, gera uma chave aleatória e a grava em `master.key` com 0600. Depois, remova a variável antiga do shell, se estava definida.

Com `--from-env`, a chave antiga vem de `master.key` e a nova vem de `MYCLI_MASTER_KEY`. A nova chave também é gravada em disco. Os dois modos recifram o cofre; não execute gravações concorrentes durante a rotação.

O que a cifra protege

Protege contra leitura casual do arquivo, cópia em backup ou sincronização de pasta e varredura do disco por ferramentas que procuram tokens em texto claro.

Não protege contra outro processo do mesmo usuário do sistema, que lê `master.key` e decifra, nem contra um usuário root, nem contra a memória do processo enquanto o valor está em uso. Para esses casos, injete a chave por variável a partir de um gerenciador de segredos.

Usar uma credencial no código

```
import { getCredential } from "../config/credentials.ts";

const token = getCredential("shop.token"); // Throws AppError with code 4 when
missing
```

O valor não vai para log, erro, `data` do `--json` sem `--reveal`, resposta de tool MCP nem frame da TUI. `mask(value)` mascara tudo até oito caracteres; para valores maiores, mantém quatro iniciais e dois finais.

A skill `termfy-config` detalha o modelo de segurança e o passo a passo para declarar chaves e consumir credenciais.

Publicar

Toda alteração publicada sai com versão, changelog, tag e GitHub Release. Isso vale para o binário derivado e para o framework.

O binário derivado

1. Atualize `version` em `package.json`.
2. Registre a mudança em `CHANGELOG.md`, no formato Keep a Changelog.
3. Rode `bun run validate`.
4. Commit `release: mycli vX.Y.Z` e tag `vX.Y.Z`. Envie a `main` e só essa tag: `git push origin main vX.Y.Z`.
5. O workflow `release.yml` compila os binários para linux-x64, linux-arm64, darwin-x64, darwin-arm64 e windows-x64, gera `SHA256SUMS` e publica a GitHub Release com as notas do changelog.
6. Confira a release e baixe um binário para testar `--version`.

Uma tag publicada não se move. Correção posterior recebe nova versão.

Build local

```
bun run build          # dist/mycli para a plataforma atual
bun run build:all    # dist/mycli-<plataforma> para todas
./dist/mycli --version
```

`bun build --compile` embute a biblioteca nativa do OpenTUI, a documentação de `docs/` e o `package.json`. Para compilar para outra plataforma, instale antes os pacotes nativos de todos os alvos: `bun install --frozen-lockfile --os="*" --cpu="*" --libc="*"`. O comando não cita pacote de propósito: com `@opentui/core` no lugar, o Bun 1.4 não instala `@opentui/core-linux-arm64-musl` e o alvo linux-arm64 falha.

Instalar o binário

Copie o executável para um diretório do `PATH` ou publique a release e oriente `curl -L <url> -o /usr/local/bin/mycli && chmod +x /usr/local/bin/mycli`. Nenhum runtime precisa estar instalado na máquina de destino.

O framework

1. Atualize `version` em `package.json` e, se `docs/user/` mudou, `ebook/VERSION`.

2. Registre em `CHANGELOG.md`.
3. Rode `bun run ebook` quando o guia mudou e `bun run validate`.
4. Commit `release: Termfy vX.Y.Z`, tag e push.
5. O workflow publica a release com o PDF e o EPUB do guia.
6. Nos projetos derivados, peça ao agente para usar `termfy-init` no modo de manutenção das skills para receber a versão publicada.

No Hub da Promovaweb

Framework e starter kit são submódulos do Hub. Depois de publicar um deles, o gitlink, a versão, o changelog, a tag e a release do Hub são atualizados, conforme `RELEASES.md`.

Glossário e problemas comuns

Glossário

Termo	Significado no Termfy
binário	o executável único gerado pelo `bun build --compile`, com TUI, CLI e MCP
starter kit	o template `promovaweb/termfy-starter-kit`, ponto de partida de cada binário
projeto derivado	o repositório criado a partir do starter kit, como o `mycli`
skill	pasta com `SKILL.md` e `references/` que ensina o agente a fazer uma etapa
`CLISPEC.md`	visão macro única do binário: propósito, públicos, limites, superfície entregue e índice das specs
spec	arquivo `specs/NNNN-slug.md` que descreve uma evolução, com tarefas e testes
Surface	a superfície do binário: commands, screens, tools, resources, prompts, settings, credenciais e tabelas
command	entrada da CLI no registry, como `config get` ou `orders list`
registry	a árvore de commands em `src/cli/commands.ts`; gera ajuda, OpenCLI, completion e referência
envelope	a forma do `--json`: `{ ok, command, data }` ou `{ ok, command, error }`
exit code	número com que o processo termina; cada família de falha tem um fixo
screen	uma aba da TUI, registrada em `src/tui/screens/index.ts`
action	comando de uma screen, com tecla, descrição, ícone e a função que roda; aparece na barra inferior, no help, na paleta e no clique
paleta de comandos	busca de ações de todas as abas, aberta com `Ctrl+K` ou `:`; executa pela tecla ou pelo clique

Termo	Significado no Termfy
toast	mensagem curta no canto inferior direito, empilhada até três, que some em 4 segundos ou com um clique
layout responsivo	composição da tela por faixa de largura: `wide` a partir de 120 colunas, `medium` a partir de 80 e `compact` abaixo de 80
Nerd Font	fonte com glifos de ícone; o conjunto `nerd` da TUI depende dela
tool	função que o servidor MCP oferece ao agente, com schema Zod
resource	conteúdo que o agente lê pelo MCP sem efeito, como `mycli://skill`
prompt	roteiro que o servidor MCP oferece para guiar uma conversa
efeito	o que uma tool faz: `read`, `write` ou `network`
setting	preferência em `settings.json`, declarada em `SETTINGS_SCHEMA`
tema `terminal`	tema padrão da TUI, que herda as cores do terminal
cofre	`credentials.json`, cifrado com AES-256-GCM
master key	chave do cofre, em `~/.<bin>/master.key` ou em `<BIN>_MASTER_KEY`
OpenCLI	formato JSON que descreve a CLI; o binário gera com `--opencli`
documentação embutida	`docs/` dentro do binário, lida com `--skill` ou pela tool `skill`
migration	arquivo que muda o schema do SQLite, embutido no binário

Problemas comuns

Instalação e setup

Sintoma	Causa provável	O que fazer
`bun: command not found`	Bun fora do `PATH`	instale com `curl -fsSL https://bun.sh/install | bash` e abra um terminal novo
`init` sai com 2	nome fora da regra	use letras minúsculas, números e hífen, começando por letra
`skills:check` falha	skill ausente ou alterada à mão	peça a `termfy-init` no modo de manutenção das skills
o agente não acha as skills	links dos agentes ausentes	rode a manutenção da `termfy-init`, que confere os links

CLI

Sintoma	Causa provável	O que fazer
o binário sem argumentos imprime a ajuda e sai com 2	não há TTY (pipe, CI, agente)	passe um command; a TUI precisa de terminal interativo
`mycli tui` sai com 8	NO_TTY	abra num terminal de verdade
`Unknown option` e exit code 2	option fora do registry daquele command	confira com `mycli <command> --help`
`opencli:check` falha no `validate`	o registry mudou e `docs/opencli.json` não	rode `bun run opencli`
`docs:check` falha	command ou key sem documentação embutida	peça a termfy-docs

Configuração e cofre

Sintoma	Causa provável	O que fazer
exit code 3 em `config get`	key inexistente	`mycli config schema` lista as keys válidas
exit code 4 com `Credential "x" not found`	o segredo não foi gravado	`mycli credentials set x`
exit code 4 ao ler o cofre em outra máquina	master key diferente	defina `<BIN>_MASTER_KEY` com a chave original ou grave os segredos de novo

Sintoma	Causa provável	O que fazer
exit code 7	pasta `~/.<bin>/` sem permissão	confira o dono da pasta; o binário cria com `0700`

MCP

Sintoma	Causa provável	O que fazer
o cliente conecta e cai em seguida	texto escrito em stdout durante a conversa	mande diagnóstico para stderr; stdout é só do JSON-RPC
o cliente não acha o binário	executável fora do `PATH` do cliente	use o caminho absoluto em `command`
a tool nova não aparece	não foi registrada em `registerAll`	confira com `mycli mcp --list`

Para depurar com uma interface, rode o MCP Inspector:

```
npx @modelcontextprotocol/inspector bun run src/main.ts mcp
```

TUI

Sintoma	Causa provável	O que fazer
cores diferentes do esperado	o tema `terminal` segue o tema do terminal; `night` e `day` pedem cor de 24 bits	troque o tema do terminal ou fixe com `tui.theme night` num terminal com truecolor
a aba nova não abre com `tui.initialTab`	o id não está nos valores da key	acrescente o id em `values` da key no `SETTINGS_SCHEMA`
a tecla da tela não responde	a ação não está em `actions`, ou `onKey` devolve `false`	declare a ação em `actions` com `run`, ou devolva `true` quando a tela consumir a tecla
o ícone aparece como quadrado	a fonte do terminal não tem Nerd Font	use `tui.icons unicode` ou instale uma Nerd Font
o mouse não seleciona texto	o mouse da TUI está ligado	use `Shift` e arraste, ou `tui.mouse false`
“Terminal too small” aparece	a janela está abaixo de 60×16	aumente a janela ou reduza o zoom do terminal; a tela volta ao crescer
o link não abre ou a cópia da URL falha	o terminal não suporta OSC 8 nem OSC 52	use o command equivalente no CLI; o link mostra a URL no texto

Sintoma	Causa provável	O que fazer
o ponteiro não vira mão sobre o clicável	o terminal não aceita formato de ponteiro	siga pela tecla; a ação funciona igual
o toast some antes da leitura	a duração é fixa em 4 segundos	repita a ação; o toast não recebe teclado
Tab muda de aba em vez de mudar o foco	a tela está com a versão antiga do shell	atualize o starter kit; `Tab` move o foco entre painéis

Release

Sintoma	Causa provável	O que fazer
o workflow falha no build	pacotes nativos de outros alvos ausentes	o workflow instala com `bun install --frozen-lockfile --os="*" --cpu="*" --libc="*"`; reproduza localmente com `bun run build:all`
a tag já existe	versão repetida	escolha a próxima versão; tag publicada não se move
o binário baixado mostra outra versão	`package.json` não foi atualizado antes da tag	publique uma versão nova com o número certo

termfy

A skill `termfy` é a porta de entrada do framework. Você descreve a ferramenta em uma frase e ela conduz o projeto desde o setup e a especificação até a publicação. Cada evolução passa pela implementação, pelos testes e pela revisão, com a documentação atualizada ao fechar a spec. Para realizar esse trabalho, ela escolhe a ordem, chama as outras skills e para em dois checkpoints para você aprovar. No fim, você tem um binário com specs fechadas, os três modos conferidos e um resumo de como instalar e rodar.

Quando usar

Chame a `termfy` nestas situações:

- Você tem só a ideia, em linguagem natural (“quero um CLI que liste os pedidos da loja e avise quando chegar um novo”).
- Você quer continuar o trabalho de um binário e não sabe a etapa onde ele parou. A skill lê o repositório e descobre sozinha.
- Você não sabe qual skill específica pedir.

Prefira outra skill nestes casos:

- Só setup, instalação ou reparo das skills: use termfy-init. A `termfy` encaminha para ela e não abre a entrevista do produto.
- Mudar nome, cores ou créditos: use termfy-brand.
- Uma única evolução, com o `CLISPEC.md` já escrito: vá direto à termfy-spec.

Como pedir

Para começar do zero, com só a ideia:

> Use a skill termfy para criar um CLI chamado mycli que liste os pedidos da loja, mostre o detalhe de um pedido e avise no terminal quando chegar um pedido novo.

Para retomar um projeto parado:

> Use a skill termfy para continuar o mycli de onde ele parou.

Para uma evolução que atravessa CLI e MCP:

> Use a skill termfy para fazer o mycli exportar os pedidos do dia em CSV para o CI, com uma tool MCP para o agente pedir o mesmo arquivo.

O que a skill faz

1. Confirma a identidade em uma única mensagem: nome do executável, uma frase de propósito e os públicos (você no terminal, o script que roda, o agente). O resto vem das entrevistas das outras skills.
2. Faz o diagnóstico do repositório com `git status --short`, `bun run dev -- --version` e `bun run skills:check`. Lê o `CLISPEC.md` e o `Status:` de cada arquivo de `specs/` para achar o estágio atual.
3. No esqueleto, chama a termfy-init e, se houver cor, créditos ou links a ajustar, a termfy-brand.
4. Chama a termfy-clispec. Antes, extrai da sua frase as respostas que a entrevista pede, para não perguntar de novo.
5. Chama a termfy-spec para a primeira spec da lista, ou a termfy-api-import quando o ponto de partida é a documentação de uma API.
6. Checkpoint 1: mostra o Purpose, os públicos, a lista de specs e a tabela de commands, screens e tools da primeira spec. Você aprova.
7. Implementa cada tarefa pendente na ordem config, api, database, cli, mcp e tui, com a skill de cada uma. Depois de cada tarefa, roda a `termfy-test` para os itens de Tests.
8. Roda a `termfy-review`, corrige o que ela apontar e passa pela `termfy-docs`. Com todos os itens marcados, fecha a spec e segue para a próxima da lista.
9. Checkpoint 2: mostra o `--help`, o frame da TUI de teste e o `mcp --list`. Você aprova a release.
10. Chama a `termfy-release` e, se você vai usar o binário com um agente, a `termfy-mcp-client`.

Exemplo

A frase “quero um CLI que liste meus pedidos da loja, mostre o detalhe de um pedido e avise quando chegar um novo” vira a mensagem de confirmação abaixo. A skill sugere a forma final e espera o seu sim.

```
Antes de começar, confirme três pontos:

1. Executável: mycli, título "My CLI". Pode ser?
2. Propósito: "Query shop orders from the terminal."
3. Públicos: você no terminal, com TUI para acompanhar pedidos novos.
   Algum script ou agente vai usar o mycli?

Proposta de specs, na ordem de entrega:

  0001  shop-client     cliente da API da loja com shop.token
  0002  orders-list     orders list, screen orders, tool orders_list
  0003  orders-show     orders show <id>, detalhe na screen orders
  0004  orders-watch    aviso de pedido novo na TUI

Fora do escopo: criar ou cancelar pedidos.
```

Depois do checkpoint 1, cada marco chega em três linhas:

```
Pronto: command orders list (Tasks da 0002 marcada).
Próximo: tool orders_list com a termfy-mcp.
Teste agora: bun run dev -- orders list --json
```

Os estágios que a skill percorre ficam nesta tabela, resumida da referência `flow.md`:

Estágio	Skill	Checkpoint
Diagnóstico	`termfy`	não
Identidade	`termfy-init`, `termfy-brand`	não
Visão macro	`termfy-clispec`	não
Spec	`termfy-spec` ou `termfy-api-import`	sim, na primeira spec
Implementação	`termfy-config` até `termfy-tui`	não
Testes, revisão, docs	`termfy-test`, `termfy-review`, `termfy-docs`	não
Fechamento	`termfy-spec`	não
Release	`termfy-release`	sim, antes
Agentes	`termfy-mcp-client`	não

Regras que a skill segue

- **Poucas perguntas, cedo.** No máximo três perguntas além da identidade, em uma mensagem, com um caminho recomendado em cada. Assim você responde uma vez e o resto corre sem interrupção.
- **Sugerir, não escolher sozinha.** Nome do executável, commands, credenciais e a divisão em specs chegam com a forma final para você aprovar (“`orders list`, pode ser?”). Até o checkpoint 1, nada vira código.
- **Dois checkpoints fixos.** Um depois do `CLISPEC.md` e da primeira spec, outro antes da release. Esses são os pontos que mudam o produto; o resto é execução.
- **Uma entrega por vez.** Cada tarefa termina com teste verde e item marcado na spec antes da próxima. Isso deixa o progresso visível no próprio arquivo da spec.
- **Reutiliza as skills.** A `termfy` lê a saída de cada skill e decide o passo seguinte, sem refazer o trabalho delas.
- **Para depois de duas falhas.** Se teste, `validate` ou build falham duas vezes seguidas no mesmo passo, ela mostra o erro e pergunta. Insistir em loop gasta tempo sem corrigir a causa.
- **Segredo fora do chat.** Quando um command precisa de credencial, ela orienta `mycli credentials set shop.token` no seu terminal e segue com o restante. O valor fica no cofre cifrado, longe do histórico da conversa.
- **Volta à skill certa.** Se você pede algo que contradiz a spec aprovada, ela volta à `termfy-spec`. Se muda o propósito, os públicos ou os limites, volta à `termfy-clispec`.

O que ela entrega

No fim, a skill relata:

- o estágio alcançado;
- os commands, screens e tools entregues;
- as specs fechadas e as abertas;
- os testes executados;
- as pendências registradas em `## Notes`;
- os comandos para você experimentar os três modos.

Os arquivos que mudam são os das skills chamadas: `package.json`, `CLISPEC.md`, `specs/`, código em `src/`, testes, `docs/` e `CHANGELOG.md`. Commit, tag e GitHub Release só acontecem na etapa da `termfy-release`, depois do seu aceite no checkpoint 2.

Próxima skill

A `termfy` não tem uma próxima skill: ela chama as demais. Para entender cada etapa, siga a ordem do ciclo a partir da termfy-init, depois a termfy-clispec e a termfy-spec.

termfy-init

A skill `termfy-init` prepara um repositório criado a partir do starter kit e mantém as skills do framework instaladas. No modo de setup, ela aplica a identidade do seu binário, instala as dependências, confere as skills e os três modos e roda a validação completa. No modo de manutenção, ela instala, atualiza ou repara as skills sem mexer no produto. No fim, você tem um projeto com o nome certo em `package.json`, as skills conferidas e o caminho aberto para a termfy-clispec.

Quando usar

Use a `termfy-init` nestas situações:

- Você acabou de criar o repositório com **Use this template** e o `bun run dev -- --version` ainda responde `termfy`.
- As skills de `.agents/skills/` estão desatualizadas, faltando ou com arquivos quebrados.
- O `bun run skills:check` falha dentro do `bun run validate`.

Prefira outra skill nestes casos:

- O setup já foi feito e você quer trocar título, cores, créditos ou o próprio nome do executável: use termfy-brand.
- Você quer descrever o que o binário faz: use termfy-clispec.
- Você só tem a ideia e quer que o agente conduza tudo: use termfy, que chama a `termfy-init` quando precisa.

Como pedir

Para o setup de um projeto novo, com todos os campos de uma vez:

> Use a skill termfy-init para preparar este projeto com o nome mycli, o título My CLI, a descrição “Query shop orders from the terminal.”, o autor Loja Exemplo e o repositório `https://github.com/loja-exemplo/mycli`.

Para atualizar as skills de um projeto que já existe:

> Use a skill termfy-init para atualizar as skills do Termfy no mycli.

Para reparar uma instalação quebrada:

> Use a skill termfy-init para reparar as skills: o skills:check diz que falta a pasta references da termfy-spec.

O que a skill faz

Primeiro, a skill informa o modo escolhido (setup ou manutenção) e os arquivos que vai conferir. Ela só pergunta quando o pedido não deixa claro qual dos dois você quer.

No modo de setup:

1. Pergunta os campos que faltarem: nome do executável (minúsculas, números e hífen), título, descrição em uma frase, autor, URL do repositório e site.
2. Roda `git status --short` e preserva as mudanças locais.
3. Roda `bun install`.
4. Roda o bootstrap com os campos conhecidos: `bun run init -- --name mycli --title "My CLI" ...`.
5. Executa a manutenção das skills (descrita abaixo) e confere a compatibilidade.
6. Roda `bun run opencli` para regenerar `docs/opencli.json` com a nova identidade.
7. Confere os três modos com `bun run dev -- --version`, `bun run dev -- about --json` e `bun run dev -- mcp --list`. Sem terminal interativo, usa o teste `tests/tui.test.ts` para a TUI.
8. Roda `bun run validate`.
9. Reescreve a abertura do `README.md` com o propósito do produto e apaga o trecho sobre o template.
10. Passa para a termfy-clispec.

No modo de manutenção:

1. Confere o Git, a versão em `package.json`, `.agents/skills/` e `skills-lock.json`. Avisa antes de sobrescrever alterações locais.
2. Lê os scripts `skills:install`, `skills:update` e `skills:check` e confirma a origem `https://github.com/promovaweb/termfy`.
3. Roda `bun run skills:update` para atualizar ou `bun run skills:install` para instalar e reparar. Quando os scripts alcançam outras bibliotecas, restringe a operação ao Termfy.
4. Roda `bun run skills:check`, lê os avisos e confere os links dos agentes, as origens no lock e o diff.
5. Reabre os arquivos atualizados e encerra. A identidade fica como estava.

Exemplo

O bootstrap que a skill roda para o `mycli` e a saída que ele imprime:

```
bun run init -- --name mycli --title "My CLI" \
  --description "Query shop orders from the terminal." \
  --author "Loja Exemplo" \
  --repository https://github.com/loja-exemplo/mycli \
  --site https://loja-exemplo.com
```

```
✓ package.json: termfy → mycli
✓ .env.example
✓ README.md
✓ AGENTS.md
✓ CHANGELOG.md: history started at 0.1.0

Done. Check with: bun run dev -- --version && bun run dev -- --help
Then run: bun run validate
```

Depois do bootstrap, o bloco `termfy` do `package.json` fica assim:

```
{
  "name": "mycli",
  "version": "0.1.0",
  "bin": { "mycli": "src/main.ts" },
  "termfy": {
    "bin": "mycli",
    "title": "My CLI",
    "author": "Loja Exemplo",
    "site": "https://loja-exemplo.com",
    "repository": "https://github.com/loja-exemplo/mycli",
    "framework": "https://github.com/promovaweb/termfy"
  }
}
```

A conferência dos modos confirma o nome novo:

```
$ bun run dev -- --version
mycli 0.1.0
$ bun run dev -- config path
  Directory         /home/voce/.mycli
  settings.json     /home/voce/.mycli/settings.json
  credentials.json  /home/voce/.mycli/credentials.json
  master.key        /home/voce/.mycli/master.key
```

Regras que a skill segue

- **O agente roda os comandos.** A skill executa o setup e a manutenção e não entrega a você uma lista de comandos para copiar. Isso evita passos esquecidos e mantém o relato fiel ao que rodou.
- **O nome do executável segue `^[a-z][a-z0-9-]{1,39}$`.** O bootstrap recusa outro formato com exit code 2, porque o nome vira a pasta `~/.mycli/`, o prefixo `MYCLI_HOME` e `MYCLI_MASTER_KEY` e o esquema `mycli://` das URIs MCP.
- **O bootstrap não cria `CLISPEC.md` nem `specs/`.** O starter kit não traz nenhum dos dois. A visão macro sai da entrevista da `termfy-clispec`. A primeira spec recebe o número 0001.
- **O bootstrap não altera `src/` nem `docs/`.** Os dois leem a identidade do manifest em tempo de execução, então não precisam de reescrita.
- **O projeto novo começa em 0.1.0.** O bootstrap troca a versão e cria um `CHANGELOG.md` próprio. O histórico do template não pertence ao seu binário.
- **Manutenção preserva o produto.** Ela não roda o bootstrap, não renomeia o binário e não recria o `CLISPEC.md` nem as specs. Assim, você recebe as instruções atualizadas sem alterar o produto.
- **Compatibilidade conferida, não presumida.** A skill só declara a instalação compatível depois de ler os avisos do `skills:check`.
- **Setup não publica.** Commit, push, tag e GitHub Release ficam com a `termfy-release`. Elas só acontecem quando você pedir.
- **Validação vermelha não vira produto pronto.** Com `bun run validate` falhando, a skill relata que só a preparação terminou.

O `bun run validate` encadeia typecheck, testes, `skills:check`, `opencli:check`, `docs:check` e build. Ele não lê o `CLISPEC.md` nem as specs: a conferência desses arquivos com o código é manual.

O que ela entrega

No setup, a skill relata:

- o nome do binário e os caminhos de configuração (`~/.mycli/`);
- a origem das skills instaladas (o `skills-lock.json` registra origem e hash, sem versão SemVer);
- os comandos conferidos e os testes executados;
- o que ficou pendente.

Arquivos que mudam no setup: `package.json`, `.env.example`, `README.md`, `AGENTS.md`, `CHANGELOG.md`, `.github/workflows/*.yml`, `docs/opencli.json`, `.agents/skills/` e `skills-lock.json`.

Na manutenção, a skill relata as skills alteradas. Mudam só `.agents/skills/`, os links dos agentes e o `skills-lock.json`.

Próxima skill

Depois do setup, siga para a termfy-clispec, que entrevista você e escreve o primeiro `CLISPEC.md`. Se quiser ajustar cores ou créditos antes, passe pela termfy-brand.

termfy-brand

A skill `termfy-brand` ajusta a identidade do binário depois do setup: nome do executável e os campos que o apresentam no terminal e nos agentes. Ela também altera os créditos e a aparência da TUI, usando os arquivos responsáveis por cada campo. A tabela deste guia mostra essas origens. A skill mantém essa fonte em ordem para que `--version`, `about`, a aba About, o OpenCLI e o servidor MCP mostrem os mesmos valores. No fim, você tem os campos novos aplicados, as superfícies conferidas e o `bun run validate` verde.

Quando usar

Use a `termfy-brand` nestas situações:

- A loja mudou de nome e o `mycli` precisa de outro título ou descrição.
- Você quer a cor da marca no destaque da TUI.
- Você quer acrescentar um crédito, um site ou uma licença à aba About.
- Você precisa renomear o executável depois que o projeto já existe.

Prefira outra skill nestes casos:

- O repositório acabou de sair do template e ainda responde `termfy`: use termfy-init, que faz o rename inicial.
- Você quer mudar o propósito, os públicos ou os limites do binário: use termfy-clispec. Identidade é nome e aparência, não escopo.
- Você quer uma screen nova ou um layout diferente: use a `termfy-tui`.

Como pedir

Para trocar a cor e o autor:

> Use a skill termfy-brand para usar a cor #e4572e como destaque da TUI do mycli e trocar o autor para Loja Exemplo Ltda.

Para acrescentar um crédito:

> Use a skill termfy-brand para acrescentar aos créditos do mycli a API da loja, com o papel “orders and customers API” e o site da loja.

Para renomear o executável:

> Use a skill termfy-brand para renomear o mycli para lojacli, mantendo o título My CLI.

O que a skill faz

1. Lê o estado atual com `git status --short` e `bun run dev -- about --json`.
2. Mostra cada campo que muda, com o valor atual e o proposto lado a lado. Depois, espera a sua confirmação.
3. Aplica os campos no `package.json` (bloco `termfy` e campos raiz). Para créditos, edita `src/app/manifest.ts`. Para cores e glifos, edita `src/tui/theme.ts`.
4. Se o `bin` mudou, roda `bun run init -- --name <bin>` e ajusta o título e a linha do `bin` no `CLISPEC.md`.
5. Roda `bun run opencli` e `bun run validate`.
6. Confere as superfícies: `bun run dev -- --version`, `bun run dev -- about`, `bun run dev -- --help | head`, `bun run dev -- mcp --list` e a aba About no teste `tests/tui.test.ts` (ou `bun run dev` em um terminal).
7. Registra a mudança no `CHANGELOG.md` do projeto.

Exemplo

Cada campo vive em um lugar e alimenta várias superfícies. A skill segue esta tabela de origens:

Campo	Onde fica	Onde aparece
`bin`	`termfy.bin` no `package.json`	`~/.<bin>/`, `<BIN>_HOME`, `<bin>://`, `--version`, `--help`
`title`	`termfy.title`	cabeçalho da TUI, aba About, `about`, OpenCLI
`description`	`description`	`--help`, cabeçalho da TUI, OpenCLI, MCP
`author`, `site`	bloco `termfy`	`about`, aba About, OpenCLI
`license`	`license`	`about`, aba About, OpenCLI
créditos	`credits` em `src/app/manifest.ts`	`about`, aba About, tool MCP `about`
paleta	`DEFAULT_TERMINAL_SLOTS`, `themes.night` e `themes.day` em `src/tui/theme.ts`	toda a TUI

Para a cor de marca do primeiro exemplo, a skill altera só os papéis de destaque. No tema `terminal`, ela escolhe o slot ANSI mais próximo em `DEFAULT_TERMINAL_SLOTS`. Nas paletas `night` e `day`, ela grava o hex.

```
export const themes: Record<"night" | "day", Theme> = {
  night: {
    name: "night",
    // ...
    borderActive: "#e4572e",
    accent: "#e4572e",
    accentSecondary: "#f08a6c",
    keyBackground: "#e4572e",
    // ...
  },
  day: {
    name: "day",
    // ...
    borderActive: "#c43d16",
    accent: "#c43d16",
    accentSecondary: "#e4572e",
    keyBackground: "#c43d16",
    // ...
  },
};
```

O relato chega como uma tabela de antes e depois:

```
Campo              Antes          Depois
author             Loja Exemplo   Loja Exemplo Ltda
accent (terminal)  ANSI 4         ANSI 1
accent (night)     #7aa2f7        #e4572e
accent (day)       #2e7de9        #c43d16

Arquivos: package.json, src/tui/theme.ts, docs/opencli.json, CHANGELOG.md
Conferido: --version, about, --help, aba About, mcp --list
```

Regras que a skill segue

- **Uma fonte de identidade.** Os valores ficam no bloco `termfy` do `package.json` mais `name`, `version`, `description` e `license`. O `src/app/manifest.ts` lê dali e as superfícies leem do manifest. Um valor copiado em outro arquivo fica velho na próxima mudança.
- **Créditos ficam em `credits`.** Cada crédito tem nome, papel e URL. A skill não remove Bun, OpenTUI e MCP SDK enquanto o binário os usa, porque o crédito descreve o que roda de fato.

- **Mudar `bin` é um rename.** A skill usa `bun run init -- --name <bin>`, que reescreve `package.json`, `.env.example`, `README.md`, `AGENTS.md`, `CHANGELOG.md` e workflows. A pasta `~/.<bin>/`, o prefixo `<BIN>_` e o esquema `<bin>://` mudam junto. A skill avisa que as configurações e as credenciais antigas ficam na pasta antiga.
- **O título fica no rename.** Sem `--title`, o script mantém o título atual ("My CLI" continua "My CLI"). Para trocar os dois, passe `--name` e `--title` na mesma chamada.
- **`description` em inglês, abaixo de 80 caracteres.** Ela aparece no `--help`, no cabeçalho da TUI e no OpenCLI. Acima disso, o cabeçalho corta o texto.
- **Cores por papel, nas três paletas.** No tema `terminal`, a marca escolhe o slot ANSI em `DEFAULT_TERMINAL_SLOTS`, por exemplo `accent: ANSI.cyan`, sem hex. Cor hex da marca entra em `night` e `day`. A cor entra como `accent` (e `accentSecondary` quando houver uma segunda cor). Ela não vira campo novo usado por uma screen só. Assim, trocar de tema mantém a marca.
- **Contraste mínimo de 4.5:1.** A skill confere `text` sobre `surface` e `keyText` sobre `keyBackground`. Abaixo disso, a leitura fica difícil em vários terminais.
- **About e `about` mostram o mesmo.** Os dois leem `aboutData()` em `src/cli/commands/about.ts`. Um campo novo entra nessa função e aparece nas duas superfícies.
- **Sem logo em ASCII.** O cabeçalho da TUI usa o glifo ▌ mais o título. Para um glifo próprio, a skill troca `glyphs.block` em `src/tui/theme.ts`, que ocupa uma célula em vez de várias linhas.
- **OpenCLI regenerado.** Depois de qualquer mudança, a skill roda `bun run opencli`, porque o bloco `info` do `docs/opencli.json` muda junto e o `opencli:check` falharia.
- **`CLISPEC.md` alinhado à mão.** O título, o `bin` e a `description` do `CLISPEC.md` precisam bater com o `package.json`. A skill não altera a Surface nem o índice `## Specs`, porque identidade não muda o que o binário faz.

O que ela entrega

A skill relata:

- os campos alterados, com o valor anterior e o novo;
- os arquivos alterados;
- as superfícies conferidas;
- o aviso do rename da pasta `~/.<bin>/` quando o `bin` mudou.

Arquivos que podem mudar: `package.json`, `src/app/manifest.ts`, `src/tui/theme.ts`, `src/tui/screens/about.ts` (painel "Standards"), `src/cli/commands/about.ts`, `docs/opencli.json`, `CLISPEC.md` e `CHANGELOG.md`. No rename, também `.env.example`, `README.md`, `AGENTS.md` e os workflows.

Próxima skill

Se o projeto ainda não tem `CLISPEC.md`, siga para a termfy-clispec. Se ele já tem, volte ao trabalho da spec aberta com a termfy-spec ou deixe a termfy escolher o próximo passo.

termfy-clispec

A skill `termfy-clispec` entrevista você para entender o que o binário precisa fazer e escreve o `CLISPEC.md`, a visão macro única do binário, em Markdown puro na raiz do repositório. O arquivo guarda a identidade, o propósito, os públicos e seus modos, o que fica fora do escopo, a superfície já entregue e o índice das specs. No fim, você tem o `CLISPEC.md` escrito e os arquivos das specs iniciais criados em `specs/` como `draft`, prontos para a termfy-spec.

Quando usar

Use a `termfy-clispec` nestas situações:

- O projeto acabou de passar pela termfy-init e ainda não tem `CLISPEC.md`. O starter kit não traz esse arquivo.
- O propósito do `mycli` mudou: além de consultar pedidos, ele passa a atender os operadores da expedição.
- Um público novo chegou, como um agente que vai operar o binário pelo MCP.
- Um limite mudou: algo que estava fora do escopo passa a entrar.
- Você quer uma visão geral escrita do que o binário é.

Prefira outra skill nestes casos:

- Detalhar uma evolução (um command novo, uma screen, uma tool): use a termfy-spec. O `CLISPEC.md` não planeja trabalho.
- Trocar nome, título ou descrição: use a termfy-brand.
- Partir da documentação de uma API: escreva o `CLISPEC.md` aqui primeiro e depois use a termfy-api-import.

Como pedir

Para o primeiro `CLISPEC.md`, logo depois do setup:

> Use a skill termfy-clispec para descrever o mycli: ele lista os pedidos da loja, mostra o detalhe de um pedido e avisa quando chega um novo. Eu uso no terminal e o CI exporta os pedidos do dia.

Para registrar um público novo:

> Use a skill termfy-clispec para incluir no mycli um agente que responde perguntas sobre pedidos pelo MCP.

Para revisar os limites:

Use a skill termfy-clispec para tirar do escopo do mycli o suporte a várias lojas e registrar o motivo.

O que a skill faz

1. Lê o estado de partida com `git status --short`, o `CLISPEC.md` (quando existe) e os arquivos de `specs/`.
2. Conduz a entrevista em cinco blocos, nesta ordem:
   - **Problema:** que tarefa você faz hoje sem o binário e o que dá trabalho nela.
   - **Públicos e modos:** a pessoa que digita, o script ou CI que chama o binário, o agente que opera pelo MCP.
   - **Capacidades:** o que o binário precisa saber fazer, em verbos. A skill pergunta qual delas entra primeiro.
   - **Integrações e armazenamento:** serviços externos, credenciais, preferências ajustáveis e, só se as capacidades sugerirem, histórico guardado entre execuções.
   - **Limites:** o que o binário não faz, mesmo parecendo próximo.
3. A cada bloco respondido, escreve a seção correspondente e mostra o trecho para você.
4. Preenche `## Surface` com o que o esqueleto já entrega, lido do código: `bun run dev -- --opencli` para os commands, `src/tui/screens/index.ts` para as screens, `bun run dev -- mcp --list` para tools, resources e prompts, `SETTINGS_SCHEMA` para as keys e as chamadas a `getCredential()` para as credenciais.
5. Monta com você a lista de specs iniciais, uma por capacidade entregável, na ordem de entrega. Para cada spec aprovada, grava `specs/NNNN-slug.md` com o próximo número livre e `Status: draft`. Depois, acrescenta a linha ao índice `## Specs`.
6. Faz a conferência manual: seções na ordem, identidade igual ao `package.json` e uma linha do índice por arquivo de `specs/`, com título e status iguais aos da spec.
7. Indica a termfy-spec para detalhar a primeira spec.

Exemplo

O `CLISPEC.md` do `mycli` logo depois da entrevista. A Surface traz só o que o esqueleto já entrega; os pedidos chegam a ela quando as specs forem fechadas.

```
# My CLI

`mycli`: Query shop orders from the terminal.

## Purpose

A pessoa que opera a loja consulta pedidos abrindo o painel web várias
vezes por dia. O binário traz pedidos, status e exportação para o terminal
e para o CI, usando a API da loja com um token guardado no cofre. Ele não
guarda pedidos localmente.

## Audiences

| Público | Modo | Necessidade |
| --- | --- | --- |
| operador da loja | TUI e CLI texto | ver pedidos abertos e o status de cada um |
| CI | CLI `--json` | exportar os pedidos do dia |
| agente | MCP | responder perguntas sobre pedidos |

## Out of scope

- Criar, editar ou cancelar pedidos: o binário só lê.
- Várias lojas na mesma instalação.

## Surface

### Commands

| Command | Função | Options | `data` no `--json` | Exit codes | Credenciais |
| --- | --- | --- | --- | --- | --- |
| `tui` | Open the terminal interface. | `--tab <overview\|about>` | | 8 | |
| `about` | Show version, executable, runtime and credits. | | `{ bin, title,
version, ... }` | | |
| `config set` | Write a value to settings.json. | | | 3, 6, 7 | |
| `credentials set` | Save or replace a credential. | `--stdin` | | 4, 7, 130 | |
| `mcp` | Start the MCP server over stdio. | `--list` | | 9 | |

### Screens

| Screen | Função | Teclas |
| --- | --- | --- |
| `overview` | visão geral do binário | `←` `→`, `q` |
```

```
| `about` | identidade, versão e créditos | `←` `→`, `q` |

### MCP

| Tipo | Nome | Função | Efeito |
| --- | --- | --- | --- |
| tool | `about` | identidade e runtime | `read` |
| tool | `config_get` | valor de uma key | `read` |
| resource | `mycli://exit-codes` | tabela de exit codes | `read` |
| prompt | `operate` | guia o agente pela CLI | |

### Settings

| Key | Tipo | Padrão | Função |
| --- | --- | --- | --- |
| `output.format` | `enum` (`text`, `json`) | `text` | formato padrão da saída |
| `tui.theme` | `enum` (`terminal`, `night`, `day`) | `terminal` | paleta da TUI |

## Specs

| ID | Título | Status |
| --- | --- | --- |
| 0001 | Cliente da API da loja | draft |
| 0002 | Listar pedidos | draft |
| 0003 | Exportar pedidos | draft |

## Notes

- 2026-11-12: visão macro registrada em entrevista.
- 2026-11-12: dúvida aberta: a API da loja pagina por cursor ou por página?
```

O exemplo mostra parte das linhas do esqueleto para caber na página. No seu projeto, a Surface lista cada command, screen, tool, resource, prompt e key que o código registra.

Regras que a skill segue

- **Entrevista antes de escrever.** Uma seção sem resposta vira pergunta ou nota datada em `## Notes`. Suposição escrita como fato engana as skills que vêm depois.
- **Blocos curtos.** Poucas perguntas por vez, confirmação em uma frase e até três caminhos por escolha aberta, com o recomendado primeiro e espaço para resposta livre. Quando a sua frase já responde um bloco, a skill confirma o que entendeu em vez de perguntar de novo.

- **Seções fixas, nesta ordem.** `## Purpose`, `## Audiences`, `## Out of scope`, `## Surface`, `## Specs` e `## Notes`, com Notes presente mesmo vazia. A ordem fixa facilita a conferência manual.
- **Sem tarefas no `CLISPEC.md`.** Checklist, `## Tasks` e testes ficam nas specs. O `CLISPEC.md` descreve o binário, não o trabalho pendente.
- **Surface descreve o entregue.** Ela muda quando uma spec é fechada, pela aplicação da Surface da spec às tabelas. Assim, cada linha corresponde a algo que existe no código.
- **Código antigo entra por uma spec de adoção.** Para registrar algo que já existe sem spec, a skill cria uma spec que lista esse código na Surface e a fecha. O histórico fica na spec.
- **Identidade vem do `package.json`.** A linha abaixo do título traz o `bin` e a `description`. Para mudar a identidade, a skill encaminha para a `termfy-brand`.
- **Armazenamento local só quando você pede.** O padrão é guardar só settings e credenciais. Se você precisa de histórico, a skill anota em `## Purpose` o que é guardado e indica a `termfy-database` na spec que criar as tabelas.
- **Prosa no seu idioma, nomes em inglês.** Commands, keys, tools e screens aparecem como no código, porque é por esses nomes que a conferência acontece.
- **Specs iniciais pequenas.** Cada uma entrega algo testável com um comando, cabe em um ciclo de implementação e depende só das anteriores da lista. Slugs em kebab-case e em inglês (`shop-client`, `orders-list`); títulos no seu idioma.

O que ela entrega

A skill relata:

- o caminho do `CLISPEC.md`;
- os públicos e os modos registrados;
- as specs criadas, com número e título;
- as dúvidas abertas em `## Notes`;
- o resultado da conferência manual;
- a spec sugerida para a `termfy-spec`.

Arquivos que mudam: `CLISPEC.md` e os arquivos novos de `specs/`, como `specs/0001-shop-client.md`, `specs/0002-orders-list.md` e `specs/0003-orders-export.md`. O código não muda nesta etapa.

Próxima skill

Siga para a termfy-spec com a primeira spec da lista. Quando o ponto de partida é a documentação de uma API, use a termfy-api-import no lugar.

termfy-spec

A skill `termfy-spec` entrevista você sobre uma evolução do binário e escreve a spec dela em `specs/NNNN-slug.md`, em Markdown puro. A spec diz como o binário vai funcionar depois da mudança (`## Behavior` e `## Surface`) e como o código deve ser feito (`## Design`). As seções `## Tasks` e `## Tests` transformam isso em entregas verificáveis. No fim, você tem uma spec que as skills de implementação leem antes de alterar o código, sem precisar voltar a perguntar.

Quando usar

Use a `termfy-spec` nestas situações:

- Antes de implementar um command, screen, tool, setting, credencial ou tabela, quando nenhuma spec aberta cobre o pedido.
- Para detalhar uma spec `draft` criada pela termfy-clispec.
- Para adotar código que já existe sem spec, com uma spec de adoção.
- Para fechar uma spec com todos os itens marcados ou arquivar uma spec abandonada.

Prefira outra skill nestes casos:

- O projeto ainda não tem `CLISPEC.md`: passe antes pela termfy-clispec.
- O ponto de partida é a documentação de uma API: use a termfy-api-import, que grava a spec no mesmo formato.
- A mudança afeta o propósito, os públicos ou os limites do binário: volte à termfy-clispec.

Como pedir

Para detalhar a spec criada pela visão macro:

> Use a skill termfy-spec para detalhar a spec 0002 do mycli, a listagem de pedidos com filtro por status.

Para uma evolução nova:

> Use a skill termfy-spec para o mycli exportar os pedidos de um período em CSV, com uma tool MCP para o agente pedir o mesmo arquivo.

Para fechar uma spec concluída:

> Use a skill termfy-spec para fechar a spec 0002 do mycli.

O que a skill faz

1. Roda `git status --short` e lê o `CLISPEC.md` e os arquivos de `specs/`. Sem `CLISPEC.md`, encaminha para a `termfy-clispec`.
2. Abre a spec que a `termfy-clispec` já criou. Se ela não existe, escolhe o próximo número livre de quatro dígitos, grava `specs/NNNN-slug.md` com `Status: draft` e acrescenta a linha ao índice `## Specs`.
3. Conduz a entrevista em seis blocos: objetivo, comportamento, superfície, integrações e armazenamento, desenho do código e aceite. No comportamento, pede um exemplo concreto de uso, do jeito que você digitaria.
4. A cada bloco respondido, escreve a seção correspondente e mostra o trecho para você.
5. Faz a conferência manual: título e status iguais no índice, seções na ordem, Surface com os campos das tabelas e cada item de Surface citado em uma tarefa ou teste.
6. Relê a spec inteira com você e lista as dúvidas de `## Notes`.
7. Ao seguir para a implementação, troca o status para `in_progress` na spec e no índice. Indica a skill da primeira tarefa pendente.
8. No fim da implementação, com todos os itens `[x]`, fecha a spec: aplica a Surface às tabelas do `CLISPEC.md`, troca o status para `done` e acrescenta a linha datada às Notes do `CLISPEC.md`.

Exemplo

A spec `specs/0002-orders-list.md` do `mycli`, no meio da implementação:

```
# 0002 Listar pedidos

Status: in_progress

## Goal

O operador da loja precisa ver os pedidos abertos sem abrir o painel web.
Criar ou cancelar pedidos fica fora desta spec.

## Behavior

`mycli orders list` imprime uma tabela com número, cliente, total e status
dos pedidos abertos. `--status closed` troca o filtro. Com `--json`,
devolve `{ count, orders[] }`. Sem `shop.token` no cofre, sai com 4 e
sugere `mycli credentials set shop.token`. Com a API fora do ar depois de
três tentativas, sai com 5. Na TUI, a screen `orders` mostra a lista em um
`listPanel`; `↑` e `↓` movem a seleção e `Enter` abre o detalhe. A tool
`orders_list` recebe `{ status? }`, devolve o mesmo `{ count, orders[] }`
e marca `isError` quando a API falha.

## Design

- `shop/orders.ts`: `listOrders(status)`, usada pelo command, pela tool e
  pela screen.
- `shop/client.ts`: client HTTP da spec 0001; sem mudança.
- `cli/commands/orders.ts`: grupo `orders` com o subcommand `list`.
- `mcp/tools/orders.ts`: tool `orders_list`.
- `tui/screens/orders.ts`: screen `orders`.
- Sem dependência nova.
- Token ausente vira `AppError` 4; falha de rede, 5.

## Surface

- Adiciona o command `orders list`: List shop orders;
  `--status <open|closed>` (padrão `open`); `data` `{ count, orders[] }`;
  exit codes 4 e 5; credencial `shop.token`.
- Adiciona a screen `orders`: pedidos com o detalhe do selecionado;
  teclas `↑` `↓`, `Enter`.
- Adiciona a tool `orders_list`: espelha `orders list`; efeito `network`.
- Altera a credencial `shop.token`: usada por `orders list` e
  `orders_list`.
```

```
## Tasks

- [x] `listOrders` com filtro de status `#termfy-api`
- [x] Command `orders list` `#termfy-cli`
- [ ] Tool `orders_list` `#termfy-mcp`
- [ ] Screen `orders` `#termfy-tui`

## Tests

- [x] `mycli orders list --json` devolve `{ count, orders[] }`
- [x] Sem `shop.token`, `orders list` sai com exit code 4
- [ ] `mcp --list` mostra `orders_list`
- [ ] A screen `orders` abre o detalhe com `Enter`

## Notes

- 2026-11-12: `--status` aceita só `open` e `closed`; a API não tem outro.
```

A spec percorre este ciclo, por edição direta dos arquivos:

Passo	O que muda
criar	arquivo novo como `draft` e linha no índice do `CLISPEC.md`
começar	`Status: in_progress` na spec e no índice
avançar	`[x]` em cada tarefa e teste concluídos
fechar	Surface aplicada ao `CLISPEC.md`, status `done` e linha `- 2026-11-20: spec 0002 (Listar pedidos) closed.` nas Notes
arquivar	status `archived`, motivo em Notes e Surface descartada

Regras que a skill segue

- **Entrevista antes de escrever.** A spec registra o que você pediu, não o que o agente imagina. Dúvida sem resposta vai datada para `## Notes` e a spec fica em `draft`.
- **Uma spec por evolução entregável.** Se o pedido junta capacidades independentes, a skill propõe specs separadas. Assim, cada uma fecha sozinha.
- **Surface com os nomes do código.** Cada item começa por Adiciona, Altera ou Remove. Um command traz função, options, `data` do `--json`, exit codes e credenciais. Uma tool traz o efeito (`read`, `write` ou `network`). Os campos batem com as tabelas do `CLISPEC.md` para

o fechamento ser uma cópia direta. Spec sem mudança de superfície escreve `Nenhuma mudança de superfície.`

- **Behavior e Design com texto antes de `in_progress`.** Behavior responde "como se usa"; Design responde "onde e como se constrói". Comentário HTML ou texto de exemplo vale como vazio, porque a skill que implementa precisaria voltar a perguntar.
- **Exit codes de `src/app/exit-codes.ts`.** Uma família nova entra como tarefa da spec, implementada antes do command que a usa. Scripts e agentes reagem ao número, então ele precisa existir antes.
- **Banco só quando você pede.** SQLite entra quando você diz que precisa guardar algo entre execuções. Nesse caso, Surface lista tabelas, colunas e tipos. O Design descreve `~/.<bin>/data.db` e a migration nova. A tarefa leva a tag `#termfy-database`.
- **Uma tarefa por entrega, com a tag da skill.** Cada item de `## Tasks` é um checkbox GFM com a tag no fim (`#termfy-cli`, `#termfy-mcp`). Cada item de `## Tests` é um comando com o resultado esperado. Item novo entra no fim da seção.
- **Marcar só o que foi feito.** A entrevista cria itens `[ ]`. A skill que implementa e testa troca para `[x]`. Descrever uma entrega não a conclui.
- **Spec parada continua `in_progress`.** Quando uma dependência externa trava o trabalho, o motivo entra datado em Notes. O status reflete o estado real do trabalho.
- **Markdown puro.** Sem front matter, schema ou IDs de item. O binário não lê esses arquivos e não tem command para editá-los; o agente edita direto.

O que ela entrega

A skill relata:

- o caminho e o status da spec;
- a contagem de itens de Surface (adiciona, altera, remove);
- o número de tarefas e de testes;
- o resultado da conferência manual;
- as dúvidas abertas;
- a skill sugerida para a primeira tarefa.

Arquivos que mudam: `specs/NNNN-slug.md` e o índice `## Specs` do `CLISPEC.md`. No fechamento, também as tabelas de Surface e as Notes do `CLISPEC.md`.

Próxima skill

Siga pela tag da primeira tarefa pendente, na ordem config, api, database, cli, mcp e tui: a `termfy-config`, a `termfy-api`, a `termfy-database`, a `termfy-cli` (ou a `termfy-command-set` para uma família `recurso verbo`), a `termfy-mcp` e a `termfy-tui`. Depois de cada tarefa, a `termfy-test` marca os itens de Tests. Para voltar ao ciclo completo, use a termfy.

termfy-api-import

A skill `termfy-api-import` recebe a documentação de uma API (OpenAPI ou Swagger, coleção Postman, página de referência, PDF ou texto colado) e a converte em uma spec de evolução em `specs/NNNN-slug.md`. Ela lê a documentação inteira e propõe a forma da CLI, com commands agrupados por recurso. A `## Surface` registra os comandos e as tools, incluindo as settings e credenciais que eles usam. As demais seções explicam o comportamento esperado e relacionam a implementação aos testes. No fim, você tem uma spec pronta para a `termfy-api` criar o client e para a `termfy-command-set` gerar os commands.

Quando usar

Use a `termfy-api-import` nestas situações:

- Você tem a especificação OpenAPI de um serviço e quer um CLI para ele.
- Você tem só exemplos de `curl` colados de uma página de referência.
- Você quer acrescentar ao binário um grupo de commands que espelha uma API externa.

Prefira outra skill nestes casos:

- O projeto ainda não tem `CLISPEC.md`: passe antes pela termfy-clispec.
- A evolução não parte de uma documentação de API: use a termfy-spec.
- A spec já existe e falta escrever o client HTTP: use a `termfy-api`.

Como pedir

Os exemplos usam a API de WhatsApp da uazapi. O `mycli` da loja ganha o envio de mensagens para avisar os clientes sobre os pedidos.

Com a especificação OpenAPI pública:

> Use a skill termfy-api-import para converter `https://docs.uazapi.com/openapi-bundled.json` em uma spec do mycli. Quero listar as instâncias, ver o status, conectar e enviar texto.

Com um recorte menor, só leitura:

> Use a skill termfy-api-import para criar uma spec do mycli só com a listagem e o status das instâncias da uazapi, a partir da especificação em `https://docs.uazapi.com/openapi-bundled.json`.

Com exemplos colados:

> Use a skill termfy-api-import com estes exemplos de `curl` da uazapi: `GET /instance/all` com o header `admintoken` e `POST /send/text` com o header `token`.

O que a skill faz

1. Roda `git status --short`, lê o `CLISPEC.md` e lista os arquivos de `specs/`. Sem `CLISPEC.md`, encaminha para a `termfy-clispec`.
2. Obtém a documentação. Uma URL é baixada para `tmp/`, fora do controle de versão. Com OpenAPI, lê o arquivo com `bun -e` e percorre `servers`, `paths`, `parameters`, `requestBody`, `responses` e `components.securitySchemes`.
3. Monta uma tabela de trabalho com recursos, endpoints, parâmetros, autenticação, servers, paginação e erros.
4. Aplica o mapeamento: agrupa endpoints por recurso, espelha cada command de leitura em uma tool MCP e define settings e credenciais.
5. Mostra a proposta como tabela (command, endpoint, options, credencial). Você escolhe o que entra agora e o que fica para depois.
6. Cria a spec pelo ciclo da `termfy-spec`: próximo número livre, `Status: draft` e linha no índice do `CLISPEC.md`. Se uma spec aberta já cobre a API, edita essa spec.
7. Grava a Surface, o Goal com o recorte, o Behavior por command, o Design com client, paginação e rate limit, as Tasks por recurso, os Tests e as Notes com a fonte e as dúvidas.
8. Relê a spec contra o contrato da `termfy-spec` e indica a sequência de skills.

Exemplo

A proposta que a skill mostra antes de gravar:

Command	Endpoint	Options	Credencial
`instances list`	`GET /instance/all`		`uazapi.admintoken`
`instances status`	`GET /instance/status`	`--instance <name>`	`uazapi.admintoken`
`instances connect`	`POST /instance/connect`	`--instance <name>`	`uazapi.admintoken`
`messages send`	`POST /send/text`	`--instance`, `--number`, `--text`	`uazapi.admintoken`

O mapeamento sugere `uazapi.api_key` para uma API key em header. Neste exemplo, foi escolhido `uazapi.admintoken`, o nome do header na documentação. A skill registrou a escolha. Trechos da spec gravada em `specs/0004-uazapi-whatsapp.md`:

```
# 0004 Mensagens por WhatsApp com a uazapi

Status: draft

## Goal

O operador quer avisar clientes pelo WhatsApp sem sair do terminal.
Esta spec cobre instâncias (listar, status, conectar) e envio de texto.
Fora desta evolução: grupos, mídia e webhooks.

## Behavior

`mycli instances list` imprime nome e status de cada instância; com
`--json`, devolve `{ count, instances[] }`. `mycli messages send
--instance loja --number 5511999990000 --text "Pedido enviado"` imprime
o id da mensagem; com `--json`, devolve `{ message }`. Sem
`uazapi.admintoken` no cofre, sai com 4 e sugere
`mycli credentials set uazapi.admintoken`. Instância inexistente sai com
6. Resposta 429 ou 5xx sai com 5, com `details.status`.

## Design

- `uazapi/client.ts`: base URL de `uazapi.baseUrl`; header `admintoken`
  para `GET /instance/all` e header `token` para as rotas da instância.
- `uazapi/instances.ts`: resolve o `token` da instância pelo nome
  recebido em `--instance`.
- Rate limit: conferir os cabeçalhos de retry na especificação.

## Surface

- Adiciona o command `instances list`: List WhatsApp instances; `data`
  `{ count, instances[] }`; exit codes 4 e 5; credencial
  `uazapi.admintoken`.
- Adiciona o command `messages send`: Send a text message;
  `--instance <name>`, `--number <phone>`, `--text <text>`; `data`
  `{ message }`; exit codes 4, 5 e 6; credencial `uazapi.admintoken`.
- Adiciona a tool `instances_list`: espelha `instances list`; efeito
  `network`.
- Adiciona a tool `messages_send`: espelha `messages send`; efeito
  `write`, pedida pela pessoa.
- Adiciona a setting `uazapi.baseUrl`: `string`, padrão
  `https://{subdomain}.uazapi.com`.
```

```
- Adiciona a credencial `uazapi.admintoken`: usada pelos commands e
  tools da uazapi; obtida no painel da uazapi.

## Tasks

- [ ] Setting `uazapi.baseUrl` e credencial `uazapi.admintoken` `#termfy-config`
- [ ] Client da uazapi com os headers `admintoken` e `token` `#termfy-api`
- [ ] Família `instances` e tool `instances_list` `#termfy-command-set`
- [ ] Command `messages send` e tool `messages_send` `#termfy-command-set`

## Tests

- [ ] `mycli instances list --json` devolve `{ count, instances[] }`
- [ ] `messages send` sem `uazapi.admintoken` sai com exit code 4
- [ ] `instances status --instance inexistente` sai com exit code 6

## Notes

- 2026-11-12: fonte `https://docs.uazapi.com/openapi-bundled.json`.
- 2026-11-12: o server tem a variável `{subdomain}`; a pessoa grava a
  URL real com `mycli config set uazapi.baseUrl`.
- 2026-11-12: conferir se `GET /instance/all` devolve o `token` de cada
  instância.
```

O trecho omite as entradas de `instances status` e `instances connect`, que seguem o mesmo formato.

Regras que a skill segue

- **Ler tudo antes de propor.** Com OpenAPI, o arquivo é a fonte. Com texto, a skill extrai a mesma lista e marca o incerto em `## Notes`, para a dúvida não virar código errado.
- **Agrupar por recurso, não por endpoint.** `GET /instance/all` e `GET /instance/status` viram `instances list` e `instances status`. Commands agrupados ficam juntos no `--help` e na família `recurso verbo`.
- **Começar pequeno.** Leituras (`list`, `show`) de todos os recursos relevantes e escritas só dos recursos que você citou. O resto fica no `## Goal` como fora desta evolução, uma linha por recurso.
- **Parâmetros viram options e arguments.** Query param vira option em kebab-case (`page_size` vira `--page-size`); path param vira argument obrigatório. Body com até cinco campos vira uma option por campo, como `--number` e `--text`; corpo maior vira `--input <file>`.

- **Autenticação vira credencial, base URL vira setting.** A credencial aponta a seção da documentação em "Como obter". A base URL vira `service.baseUrl` com `servers[0].url` como padrão. Exemplos de token da documentação ficam fora da spec, porque são valores de terceiros.
- **Exit codes por família.** Cada command recebe 4 (credencial) e 5 (rede ou API), mais 6 quando o endpoint devolve 404. Um 401 ou 403 causado por credencial também sai com 4.
- **Tools só de leitura por padrão.** Cada command de leitura vira uma tool em `snake_case` com efeito `network`. Escritas viram tool só quando você pede, com efeito `write` e descrição que avisa o efeito, porque um agente pode chamar a tool sem você olhar.
- **Paginação e rate limit no Design.** O esquema da API (cursor, página, cabeçalhos) fica registrado para a `termfy-api` implementar.
- **Proposta antes de gravar.** Você vê a tabela e escolhe o que entra. Nada vira código nesta etapa.
- **`CLISPEC.md` só no fechamento.** As entradas vivem na Surface da spec até ela ser fechada pelo ciclo da `termfy-spec`. Assim, a Surface do `CLISPEC.md` descreve só o que existe no código.

O que ela entrega

A skill relata:

- a fonte lida;
- o número de recursos e endpoints encontrados;
- o ID e o caminho da spec;
- o que entrou na Surface e o que ficou fora;
- as dúvidas em `## Notes`;
- as skills seguintes.

Arquivos que mudam: `specs/NNNN-slug.md` e o índice `## Specs` do `CLISPEC.md`. A cópia da documentação fica em `tmp/`, fora do Git.

Próxima skill

Siga a sequência indicada pela skill: a `termfy-config` para a setting e a credencial, a `termfy-api` para o client e a `termfy-command-set` para cada recurso. Com todos os itens marcados, feche a spec com a termfy-spec.

termfy-cli

A skill `termfy-cli` cria, altera ou remove commands do CLI do seu binário. Ela parte do registry declarativo de `src/cli/registry.ts`: você descreve o command uma vez e o mesmo objeto alimenta o parser, o `--help`, o `--json`, o `--skill` e o `--opencli`. No fim, você tem o command registrado, testado em subprocesso, com saída em texto e em JSON, exit codes fixos e `docs/opencli.json` regenerado.

Quando usar

- Você quer um command novo, como `mycli orders show <id>`.
- Um command existente ganha uma option, um argument ou um subcommand.
- Você vai mudar o formato da saída ou os exit codes de um command.
- Um command saiu da spec e precisa sair do registry.

Para entregar o mesmo recurso também como tool e screen, use termfy-command-set. Para o client HTTP que o command consome, use termfy-api. Para a setting ou a credencial que ele lê, use termfy-config.

Como pedir

Para um command novo:

> Use a skill termfy-cli para criar `mycli orders show <id>`, que mostra um pedido da loja em texto e em `--json`.

Para mudar um command que já existe:

> Use a skill termfy-cli para acrescentar `--status <status>` ao `orders list`, com os valores `open`, `paid` e `shipped`.

Para um command com interação:

> Use a skill termfy-cli para criar `orders cancel <id>` com confirmação no terminal e `--yes` para scripts.

O que a skill faz

1. Roda `git status --short` e lê a entrada da spec aberta em `specs/`: Surface, Behavior, Design e os itens `#termfy-cli`.
2. Copia `template/command/example.ts` para `src/cli/commands/<nome>.ts` e ajusta os imports para caminhos relativos ao novo local.
3. Preenche a definição com os nomes da Surface e implementa `run`.

4. Registra o command em `src/cli/commands.ts`.
5. Escreve ou ajusta os testes em `tests/cli.test.ts`, com o binário rodando em subprocesso real.
6. Roda `bun run opencli` e revisa o diff de `docs/opencli.json`.
7. Atualiza `docs/cli.md` quando o command traz conceito novo. Esse texto entra no binário pelo `--skill` no próximo build.
8. Roda `bun run validate` e marca `[x]` nos itens testados da spec.

Exemplo

O command `orders show` recebe o id e chama `getOrder(id)`, do client da loja. Um subcommand novo entra nos `subcommands` do agrupador `orders`:

```
import { manifest } from "../../app/manifest.ts";
import { ExitCode } from "../../app/exit-codes.ts";
import { error } from "../../app/errors.ts";
import { table } from "../format.ts";
import type { Command } from "../registry.ts";
import { getOrder } from "../../api/shop.ts";

export const showOrderCommand: Command = {
  name: "show",
  description: "Show one shop order.",
  longDescription: "Reads shop.baseUrl from settings.json and the shop.token
credential.",
  arguments: [{ name: "id", description: "Order identifier, as listed by orders
list.", required: true }],
  exitCodes: [ExitCode.CONFIG, ExitCode.CREDENTIALS, ExitCode.NETWORK_OR_API],
  examples: [`${manifest.bin} orders show o1`, `${manifest.bin} orders show o1 --
json`],
  run: async (ctx) => {
    const id = ctx.args[0] as string;
    if (id.trim() === "") throw error.usage("The order id cannot be empty.");
    ctx.stderr(`Fetching order ${id}...`);
    const order = await getOrder(id);
    return {
      data: { order },
      text: () => table([["id", order.id], ["status", order.status]]),
    };
  },
};
```

O resultado em texto vai para stdout. A linha de progresso vai para stderr, por isso não aparece num pipe:

```
$ mycli orders show o1 2>/dev/null
  id      o1
  status  open
$ mycli orders show o1 --json 2>/dev/null
{
  "ok": true,
  "command": "mycli orders show",
  "data": {
    "order": { "id": "o1", "status": "open" }
  }
}
```

Sem a credencial, o command encerra com o código 4. Em `--json`, o erro sai no envelope com `ok: false`:

```
$ mycli orders show o1 --json 2>/dev/null; echo "exit $?"
{
  "ok": false,
  "command": "mycli orders show",
  "error": {
    "code": 4,
    "name": "CREDENTIALS",
    "message": "Credential \"shop.token\" not found. Save it with: mycli credentials
set shop.token",
    "details": { "name": "shop.token" }
  }
}
exit 4
```

O teste roda o binário com uma pasta de configuração temporária, como o helper `run` de `tests/cli.test.ts`:

```
test("orders show without the token exits with 4", async () => {
  const r = await run(["orders", "show", "o1", "--json"]);
  expect(r.code).toBe(4);
  expect(JSON.parse(r.stdout).error.name).toBe("CREDENTIALS");
});
```

Regras que a skill segue

- **Toda option e todo argument têm `description`.** Ela diz o efeito, os valores aceitos e o padrão. O mesmo texto aparece no `--help`, no `--skill` e no `--opencli`.
- **`run` devolve `{ data, text }`.** `data` é serializável e estável para scripts. `text` formata o resultado para leitura no terminal.
- **Falha prevista lança `AppError`.** `error.usage`, `error.config`, `error.credentials`, `error.network`, `error.notFound` e `error.permission` levam ao código certo. O command não chama `process.exit`, porque `src/cli/index.ts` imprime o erro e devolve o código.
- **Segredo não entra por argument.** Valor digitado na linha de comando fica no histórico do shell. Use `readSecret(label)` ou `--stdin` para gravar com `setCredential()` e `getCredential(name)` para ler.
- **`interactive: true` só descreve.** O campo não confere o TTY. O `run` verifica `ctx.interactive` e oferece `--stdin` ou `--yes` quando couber.
- **Stdout só recebe a saída do command.** Progresso e avisos vão para `ctx.stderr`, para o JSON de stdout continuar válido num pipe.
- **Default resolvido em `run`.** `OptionDefinition.default` é metadado. O parser não o coloca em `ctx.options`, então use `Number(ctx.options.limit ?? 10)`.
- **Nomes em inglês.** Commands, options e chaves de `data` ficam em inglês, com chaves em camelCase.
- **Campo publicado não muda.** Depois de publicado, um campo de `data` mantém nome e tipo. Para trocar, acrescente um campo novo e remova o antigo só em versão major.
- **Completion e OpenCLI saem do registry.** `completion <shell>` e `--opencli` mostram o command novo sem trabalho extra. Se ele depende de algo do ambiente, a verificação entra em `src/cli/commands/doctor.ts`.

O que ela entrega

A skill relata os arquivos criados ou alterados, os comandos de validação com o resultado e exemplos de uso do command em texto e em `--json`.

Arquivo	Mudança
`src/cli/commands/<nome>.ts`	command novo ou alterado
`src/cli/commands.ts`	registro na raiz, quando o command é novo
`tests/cli.test.ts`	casos em subprocesso real
`docs/opencli.json`	regenerado por `bun run opencli`
`docs/cli.md`	conceito novo explicado, quando houver
`specs/<spec>.md`	itens `#termfy-cli` marcados com `[x]`

Próxima skill

Se o mesmo recurso também vai para agentes, exponha a tool com termfy-mcp. Para uma tela interativa, siga com termfy-tui.

termfy-tui

A skill `termfy-tui` cria e revisa as screens da TUI do seu binário com a API imperativa do OpenTUI Core. Ela monta o layout em células e trata a navegação por teclado e por mouse, mantendo o foco no item selecionado. Nas telas que consultam serviços, também trata o carregamento assíncrono e libera os recursos ao sair da aba. No fim, você tem uma screen registrada como aba, com número e ícone, com estados de carregando, vazio, erro e pronto, testada por teclas, por clique e por frames nas três faixas de largura. A conferência visual cobre os temas `terminal`, `night` e `day`.

Quando usar

- Você quer uma aba nova, como a screen `orders` com a lista de pedidos.
- Uma screen existente precisa de navegação por teclado, mouse, rolagem ou um campo de formulário.
- A tela precisa de ícones de papel e de uma barra de ações clicável.
- A tela consulta uma API e você precisa tratar carregamento, erro e respostas que chegam fora de ordem.
- Uma screen trava, vaza timers ou deixa o terminal sujo ao sair.

Para entregar a screen junto com o command e a tool do mesmo recurso, use termfy-command-set. A chamada HTTP fica no client da termfy-api: a screen apresenta o resultado.

Como pedir

Para uma aba nova:

> Use a skill termfy-tui para criar a screen `orders`, que lista os pedidos da loja e mostra o pedido selecionado ao apertar Enter.

Para uma tela que consulta um serviço:

> Use a skill termfy-tui para a screen `orders` carregar ao entrar na aba, com `r` para recarregar e mensagem própria quando a API falhar.

Para revisar uma tela existente:

> Use a skill termfy-tui para revisar a screen `orders`: a lista some quando troco de aba e volto.

O que a skill faz

1. Confere o Git e lê a spec aberta em `specs/`: a screen na Surface, o Behavior com teclas e estados, o Design e os itens `#termfy-tui`.
2. Confere a versão instalada de `@opentui/core` no lock antes de usar qualquer API.
3. Lê `src/tui/index.ts`, `src/tui/screen.ts`, `src/tui/components.ts`, `src/tui/theme.ts` e `src/tui/screens/index.ts`. Mostra um plano curto com arquivos, comportamentos e verificações.
4. Escolhe um dos cinco padrões de `references/screen-patterns.md` e adapta só o necessário: lista e detalhe, painel de status, formulário por diálogos, visualizador de documento e tela de progresso.
5. Cria `src/tui/screens/<id>.ts`, compõe para `wide`, `medium` e `compact` com `ctx.layout()`, declara `icon` e as `actions` da tela e registra a screen em `src/tui/screens/index.ts`. Para abrir direto na aba, acrescenta o id em `tui.initialTab` de `src/config/settings.ts` e em `--tab` de `src/cli/commands/tui.ts`.
6. Liga cada comando a uma ação, com tecla, clique e entrada na paleta na mesma função. Testa a lógica isolada, depois a screen com `createTestRenderer` e por fim a navegação com `mountTui`, com tecla, clique e diálogos.
7. Confere `terminal`, `night` e `day` em 60×16, 80×24 e 120×32.
8. Passa a `references/quality-checklist.md` linha a linha, atualiza `docs/tui.md`, roda `bun run validate` e marca `[x]` nos itens.

Padrões de tela e checklist

`references/screen-patterns.md` traz cinco composições. Cada uma descreve quando usar, os componentes, as teclas, a composição em `wide`, `medium` e `compact` e como testar.

Padrão	Uso
lista e detalhe	coleção com seleção e painel de leitura, com `table` e `keyValueTable`
painel de status	resumo de saúde, versão ou configuração atual
formulário por diálogos	poucos campos, editados com `confirm`, `prompt` e `choose`
visualizador de documento	aba de documentação com índice e `Markdown`
tela de progresso	operação longa com `spinner` e `progressBar`

`references/quality-checklist.md` é a conferência antes de declarar a tela pronta: foco visível, estados vazio, carregando e erro, layout a partir de 60×16, ação com tecla, clique e paleta, confirmação antes de apagar, toast no resultado, resumo em `ctx.status`, contraste nos três temas, glifo só em `icons.ts`, segredo fora do frame, `watch` para arquivo lido e timers parados em `onLeave` e `onPause`.

Exemplo

A screen `orders` recebe a função de carga pela factory. Assim cada montagem tem o próprio estado e o teste injeta uma carga local:

```
import { BoxRenderable, TextRenderable } from "@opentui/core";
import { listPanel } from "../components.ts";
import type { Screen, ScreenAction, ScreenContext } from "../screen.ts";
import { listOrders, type Order } from "../../api/shop.ts";

type Load = () => Promise<{ items: Order[] }>;

export function createOrdersScreen(load: Load = () => listOrders({ limit: 50 })):
Screen {
  let root: BoxRenderable | undefined;
  let status: TextRenderable | undefined;
  let list: ReturnType<typeof listPanel> | undefined;
  let orders: Order[] = [];
  let generation = 0;

  const show = (ctx: ScreenContext, items: Order[]) => {
    if (!root) return;
    if (list) { root.remove(list.box); list.box.destroyRecursively(); }
    orders = items;
    list = listPanel(ctx.renderer, ctx.theme, {
      title: "Orders",
      items: items.map((o) => `${o.id}  ${o.status}`),
      onSelect: (i) => { if (status) status.content = `Order ${orders[i]?.id}
selected`; },
    });
    root.add(list.box, 0);
  };

  const reload = async (ctx: ScreenContext) => {
    const current = ++generation;
    if (status) status.content = "Loading orders...";
    try {
      const page = await load();
      if (current !== generation) return;
      show(ctx, page.items);
      if (status) status.content = page.items.length ? "Enter selects an order" :
"No orders";
    } catch {
      if (current === generation && status) status.content =
"Unable to load orders. Press r to retry.";
    }
  };
```

```
  const actions: ScreenAction[] = [
    { keys: ["up", "k"], label: "↑/k", description: "up", run: () =>
list?.move(-1) },
    { keys: ["down", "j"], label: "↓/j", description: "down", run: () =>
list?.move(1) },
    {
      keys: ["return", "enter"],
      label: "Enter",
      description: "select",
      icon: "select",
      run: () => {
        const index = list?.selected() ?? -1;
        if (index >= 0) list?.select(index);
      },
    },
    { keys: ["r"], description: "reload", icon: "refresh", run: (ctx) => { void
reload(ctx); } },
  ];

  return {
    id: "orders",
    title: "Orders",
    icon: "list",
    description: "Shop orders",
    actions,
    mount: (ctx) => {
      root = new BoxRenderable(ctx.renderer, { flexDirection: "column", flexGrow:
1 });
      status = new TextRenderable(ctx.renderer, { content: "", height: 1,
flexShrink: 0, fg: ctx.theme.textDim });
      root.add(status);
      ctx.renderer.once("destroy", () => { generation++; });
      return root;
    },
    onEnter: (ctx) => { void reload(ctx); },
    onLeave: () => { generation++; },
  };
}
```

O registro entra em `src/tui/screens/index.ts` como `[overviewScreen, createOrdersScreen(), aboutScreen]`. Com `orders` em `tui.initialTab` e em `--tab`, a aba abre direto:

```
mycli tui --tab orders
mycli config set tui.initialTab orders
```

Trecho do frame em 80×24, depois da carga e de uma tecla `j`:

```
┌ Orders ──────────────────────────────────────────┐
│   o1  open                                       │
│ > o7  paid                                       │
└──────────────────────────────────────────────────┘
Enter selects an order
```

O teste espera a carga controlada terminar, sem pausa fixa:

```
const setup = await createTestRenderer({ width: 80, height: 24 });
(screens as Screen[]).push(createOrdersScreen(async () => ({
  items: [{ id: "o1", status: "open" }, { id: "o7", status: "paid" }],
})));
const closed = mountTui(setup.renderer, { initialTab: "orders" }, themes.night);
try {
  await setup.waitForFrame((frame) => frame.includes("o7  paid"));
  setup.mockInput.pressKey("j");
  await setup.renderOnce();
  expect(setup.captureCharFrame()).toContain("> o7  paid");
  setup.mockInput.pressKey("q");
  await closed;
} finally { setup.renderer.destroy(); }
```

Regras que a skill segue

- **OpenTUI imperativo.** A aba usa `@opentui/core` direto, sem React ou Solid, porque o host já entrega renderer e tema prontos.
- **Uma screen por arquivo, com factory.** Estado no escopo do módulo vaza seleção entre duas montagens ou dois testes.
- **Componentes compartilhados.** `panel`, `table`, `listPanel`, `stateView`, `spinner`, `progressBar`, `link`, `keyValueTable`, `heading`, `dim` e `spacer` mantêm o visual igual entre abas. `spacer(renderer, height)` não recebe tema.
- **Ação única para tecla, clique e paleta.** Cada comando é uma `ScreenAction` com `keys`, `description`, `icon` e `run`. O botão, a linha de tabela ou de lista e a entrada da paleta chamam a mesma função.

- **Layout por faixa.** A tela lê `ctx.layout()` e compõe para `wide`, `medium` e `compact`. O shell remonta a tela quando a faixa muda.
- **Diálogo no lugar de formulário.** Um ou dois campos usam `ctx.prompt`, `ctx.choose` e `ctx.confirm`. Efeito destrutivo usa `danger` e valor oculto usa `secret`.
- **Resumo e toast.** `ctx.status` mostra o resumo da tela e `ctx.notify` mostra o resultado de cada ação.
- **Ícones de `ctx.icons`.** Nenhum glifo de papel fixo na screen; a key `tui.icons` troca o conjunto `nerd`, `unicode` ou `ascii`.
- **Mouse e teclado no mesmo efeito.** O clique, o hover e a roda passam pela mesma ação da tecla. O teste usa `setup.mockMouse`.
- **Cores de `ctx.theme`.** Cor fixa no código quebra uma das paletas. No tema `terminal`, cada cor é uma referência à paleta do terminal.
- **`mount` síncrono.** `mount` cria os nós. O trabalho ativo começa em `onEnter` e para em `onLeave`.
- **Arquivo observado.** `watch()` lista os arquivos que a tela lê. O shell recarrega a tela quando outro processo muda um deles e avisa no toast.
- **Pausa no foco.** `onPause` para timers e `onResume` retoma. `ctx.motion` governa a animação curta de diálogos e toasts.
- **Cleanup em `onLeave`.** O shell chama `onLeave` na troca de aba e no encerramento, então pare timers, assinaturas e consultas ali.
- **A aplicação é dona do renderer.** A screen sai com `ctx.quit()`, sem `process.exit()`, sem outro renderer e sem escrever em stdout.
- **`onKey` devolve `true` só para tecla consumida.** Assim `q`, `?`, `Tab`, as setas laterais e os dígitos continuam trocando de aba e encerrando.
- **Respostas antigas descartadas.** O contador de geração impede que uma carga lenta substitua uma mais nova ou altere uma aba que já saiu.
- **Mensagem própria no erro.** A screen não mostra `String(error)` da API, que pode trazer detalhes internos do serviço.
- **Serviço compartilhado.** A screen chama o mesmo client do CLI e do MCP, sem duplicar autenticação nem HTTP.

O que ela entrega

A skill relata os arquivos alterados, as versões conferidas, as teclas, as ações, os estados cobertos, as dimensões ensaiadas e um frame real capturado. Separa testes executados, exemplos só compilados e comportamentos não exercitados.

Arquivo	Mudança
`src/tui/screens/<id>.ts`	screen nova ou revisada

Arquivo	Mudança
`src/tui/screens/index.ts`	screen na lista de abas
`src/config/settings.ts`	id em `tui.initialTab`, quando abre direto
`src/cli/commands/tui.ts`	id nos valores de `--tab`, idem
`tests/`	testes com `createTestRenderer` e `mountTui`
`docs/tui.md`	teclas e uso da aba

Próxima skill

Se a screen trouxe uma preferência nova, declare a key com termfy-config. Para expor o mesmo recurso a agentes, siga com termfy-mcp.

termfy-mcp

A skill `termfy-mcp` expõe funções do seu binário a agentes de IA pelo Model Context Protocol. O servidor roda em `mycli mcp`, sobre stdio, no mesmo executável do CLI e da TUI. A skill cria tools, resources e prompts com schema Zod, paginação, tratamento de erro e testes pelo `Client` do SDK. No fim, você tem a tool registrada, visível em `mcp --list` e na descoberta real do protocolo, com resposta em texto e em `structuredContent`.

Quando usar

- Você quer que um agente consulte pedidos pela tool `orders_list`.
- Uma tool existente precisa de paginação por `limit` e `cursor`, de `outputSchema` ou de uma mensagem de erro melhor.
- Você vai expor documentação como resource `<bin>://` ou um prompt.
- Você quer revisar o que já está exposto antes de ligar o servidor num agente.

Para entregar a tool junto com o command e a screen do mesmo recurso, use termfy-command-set. Para configurar Claude Code, Codex ou outro host para usar o servidor, use a skill `termfy-mcp-client`.

Como pedir

Para uma tool de consulta:

> Use a skill termfy-mcp para criar a tool `orders_list`, com `limit`, `status` e `cursor`, que devolve os pedidos da loja.

Para revisar a resposta de uma tool:

> Use a skill termfy-mcp para acrescentar `outputSchema` e anotações à tool `orders_list` e garantir que o erro de credencial vira `isError`.

Para uma tool de gravação:

> Use a skill termfy-mcp para criar `orders_cancel` com `dryRun`, como a spec define, sem confirmação de terminal.

O que a skill faz

1. Roda `git status --short` e lê a spec aberta: tools, resources e prompts da Surface, com efeito, entrada e saída, mais os itens `#termfy-mcp`.
2. Confere a versão real do SDK e apresenta um plano com interfaces, serviços reutilizados, campos, efeitos e verificações.

3. Copia `template/tool/example.ts` para `src/mcp/tools/<grupo>.ts`, ajusta os imports e escolhe uma receita em `references/examples/`.
4. Implementa a tool chamando o serviço compartilhado, sem copiar a lógica do command.
5. Registra a função em `registerAll` (`src/mcp/server.ts`). Depois compara o catálogo local com a descoberta real.
6. Testa com o `Client` do SDK em `tests/mcp.test.ts` e, à mão, com `npx @modelcontextprotocol/inspector bun run src/main.ts mcp`.
7. Atualiza a tabela de tools em `docs/mcp.md` e a `longDescription` do command `mcp` em `src/cli/commands/mcp.ts`.
8. Roda `bun run validate` e marca `[x]` nos itens testados da spec.

Exemplo

A tool `orders_list` em `src/mcp/tools/orders.ts` projeta só os campos públicos e declara o schema de saída:

```
import { z } from "zod";
import { AppError } from "../../app/errors.ts";
import { listOrders } from "../../api/shop.ts";
import type { Registrar } from "../server.ts";

const description =
  "Use when the user asks about shop orders. Returns up to limit orders " +
  "(id, status) and nextCursor for the next page. Reads the shop API only.";

export function registerOrdersTools({ server, catalog }: Registrar): void {
  catalog.tools.push({ name: "orders_list", description });
  server.registerTool("orders_list", {
    title: "List orders",
    description,
    inputSchema: {
      limit: z.number().int().min(1).max(50).optional().describe("Maximum number of
orders. Default 10."),
      status: z.enum(["open", "paid", "shipped"]).optional().describe("Only orders
with this status."),
      cursor: z.string().min(1).optional().describe("nextCursor returned by the
previous call."),
    },
    outputSchema: {
      count: z.number().int(),
      orders: z.array(z.object({ id: z.string(), status: z.string() })),
      nextCursor: z.string().nullable(),
    },
    annotations: { readOnlyHint: true, openWorldHint: true },
  }, async ({ limit, status, cursor }) => {
    try {
      const page = await listOrders({ limit: limit ?? 10, status, cursor });
      const orders = page.items.map((o) => ({ id: o.id, status: o.status }));
      const d = { count: orders.length, orders, nextCursor: page.nextCursor };
      return { content: [{ type: "text", text: JSON.stringify(d, null, 2) }],
structuredContent: d };
    } catch (e) {
      const message = e instanceof AppError ? e.message : "The shop API request
failed.";
      return { content: [{ type: "text", text: message }], isError: true };
    }
  });
}
```

Em `src/mcp/server.ts`, a função entra em `registerAll` com a linha `registerOrdersTools(r);`. O `mcp --list` mostra o catálogo sem abrir o transporte:

```
$ mycli mcp --list
MCP server mycli v0.1.0

Tools:
  about  Version, executable, runtime, platform and credits of the binary.
  ...
  orders_list  Use when the user asks about shop orders. Returns up to ...
```

Uma chamada com `{ "limit": 2 }` devolve o mesmo objeto nos dois campos:

```
{
  "content": [{ "type": "text", "text": "{\n  \"count\": 2, ..." }],
  "structuredContent": {
    "count": 2,
    "orders": [{ "id": "o1", "status": "open" }, { "id": "o7", "status": "paid" }],
    "nextCursor": "c2"
  }
}
```

Sem a credencial `shop.token`, a falha volta como resultado, não como exceção. O agente lê a orientação e solicita que você grave o token:

```
{
  "content": [{ "type": "text",
    "text": "Credential \"shop.token\" not found. Save it with: mycli credentials
set shop.token" }],
  "isError": true
}
```

O teste conversa com o servidor real por stdio, como em `tests/mcp.test.ts`:

```
test("orders_list reports a missing token as isError", async () => {
  const { tools } = await client.listTools();
  expect(tools.map((t) => t.name)).toContain("orders_list");
  const r = await client.callTool({ name: "orders_list", arguments: { limit: 2 } });
  expect(r.isError).toBe(true);
});
```

Regras que a skill segue

- **Stdout pertence ao JSON-RPC.** Enquanto `mcp` roda, log vai para stderr. Um `console.log` no client HTTP corrompe a conversa.
- **Tool bem descrita.** Nome em `snake_case`, `title`, descrição que diz quando usar e o que devolve, Zod com `.describe()` em cada campo. O agente escolhe a tool só por esse texto.
- **Mesmo objeto em texto e estrutura.** `content` e `structuredContent` saem do mesmo valor, com `outputSchema` correspondente. Os campos são projetados antes do retorno para não vazar propriedades extras.
- **Falha prevista vira `isError: true`.** A mensagem é pública e fixa. O SDK pode devolver o texto original de uma exceção, que pode trazer detalhes do serviço.
- **Credencial lida por dentro.** A tool chama `getCredential` no serviço e não aceita token como argumento, porque o argumento fica no log do host. `credentials_status` mostra nomes e metadados, sem valores.
- **Efeito declarado.** Tool que grava diz isso na descrição e aceita `dryRun` quando couber. Um `confirm: true` vindo do modelo não comprova sua aprovação.
- **Anotação não é permissão.** `readOnlyHint` e `openWorldHint` informam o cliente. A verificação de acesso fica no código.
- **Catálogo e descoberta conferidos.** A tool entra em `catalog.tools`, senão `mcp --list` não a enxerga. O SDK não mantém esse catálogo.
- **Respostas curtas.** A tool devolve o que o agente precisa para a próxima ação, com `limit` e `cursor` quando a lista pode crescer.
- **Registro sem efeito.** Registrar a tool não consulta rede, não exige credencial e não inicia timer, porque `mcp --list` monta o catálogo sem transporte.
- **Capacidade só com teste.** HTTP, subscriptions, sampling ou elicitation só entram com integração e teste próprios.

O que ela entrega

A skill relata tools, resources e prompts adicionados ou alterados, com entrada, saída, efeitos, versões conferidas e testes. Separa receita isolada, integração no produto, binário compilado e cliente real.

Arquivo	Mudança
`src/mcp/tools/<grupo>.ts`	tools novas ou alteradas
`src/mcp/server.ts`	chamada em `registerAll`
`src/mcp/resources/` e `src/mcp/prompts.ts`	quando a spec pede
`tests/mcp.test.ts`	descoberta e chamadas pelo `Client`

Arquivo	Mudança
`docs/mcp.md`	tabela de tools
`src/cli/commands/mcp.ts`	`longDescription` com a lista de tools

Próxima skill

Para ligar o servidor num agente, siga com a skill `termfy-mcp-client`. Se a tool lê uma credencial nova, declare o nome com termfy-config.

termfy-config

A skill `termfy-config` mantém o que o seu binário guarda em `~/.<bin>/`: as preferências em `settings.json`, os segredos no cofre cifrado `credentials.json`, a master key e as variáveis de ambiente que mudam esse comportamento. Ela acrescenta keys ao `SETTINGS_SCHEMA` e liga credenciais aos commands e às tools que as consomem. No fim, você tem a key visível em `config schema`, a credencial lida com segurança e testes isolados da sua pasta pessoal.

Quando usar

- Você precisa de uma preferência nova, como `shop.baseUrl` ou o tamanho de página de `orders list`.
- Um command ou uma tool vai ler a credencial `shop.token`.
- Você quer mudar a política da master key, por exemplo para CI.
- O cofre não abre e você quer entender a pasta `~/.mycli/`.

Para o client HTTP que usa a URL e o token, use termfy-api: ela chama esta skill para declarar a key. Para um command que só lê uma setting já existente, use termfy-cli.

Como pedir

Para uma key nova:

> Use a skill termfy-config para acrescentar `shop.baseUrl` e `shop.pageSize` ao schema, com tipo, padrão e descrição.

Para consumir uma credencial:

> Use a skill termfy-config para o command `orders list` e a tool `orders_list` lerem a credencial `shop.token` do cofre.

Para preparar o CI:

> Use a skill termfy-config para documentar como rodar o `mycli` no CI com `MYCLI_MASTER_KEY`, sem gravar `master.key` em disco.

O que a skill faz

1. Lê a spec aberta em `specs/`: as settings e credenciais da Surface, o Behavior, o Design e os itens `#termfy-config`. Sem spec que cubra o pedido, chama a skill `termfy-spec` antes.
2. Lê `docs/configuration.md` e os arquivos de `src/config/`: `paths.ts`, `settings.ts`, `crypto.ts` e `credentials.ts`.

3. Para uma key: acrescenta a definição em `SETTINGS_SCHEMA`, lê o valor com `getPath(readSettings(), "group.key")` e testa em `tests/config.test.ts` a conversão e o padrão.
4. Para uma credencial: chama `getCredential(name)` no início do `run` ou do callback da tool, antes de qualquer efeito. Acrescenta `ExitCode.CREDENTIALS` em `exitCodes` do command.
5. Documenta a key ou a credencial em `docs/configuration.md` e na `longDescription` do command que a usa.
6. Roda `bun run validate` ao concluir o escopo e marca `[x]` nos itens.

Exemplo

Duas keys novas entram em `SETTINGS_SCHEMA`, no `src/config/settings.ts`:

```
{
  key: "shop.baseUrl",
  type: "string",
  default: "https://api.example.com",
  description: "Base URL for the shop API.",
},
{
  key: "shop.pageSize",
  type: "number",
  default: 10,
  description: "Default number of orders returned by orders list.",
},
```

O command lê os dois valores e a credencial em tempo de chamada:

```
import { getPath, readSettings } from "../../config/settings.ts";
import { getCredential } from "../../config/credentials.ts";

const settings = readSettings();
const baseUrl = String(getPath(settings, "shop.baseUrl"));
const pageSize = Number(getPath(settings, "shop.pageSize") ?? 10);
const token = getCredential("shop.token");
```

O `config schema` já mostra as keys. O `config set` converte o texto para o tipo declarado:

```
$ mycli config schema
Known keys

  output.format      Default CLI output format when --json is not given. [enum:
text|json; default "text"]
  ...
  shop.baseUrl       Base URL for the shop API. [string; default "https://
api.example.com"]
  shop.pageSize      Default number of orders returned by orders list. [number;
default 10]
$ mycli config set shop.pageSize ten
error "shop.pageSize" expects a number; got "ten". [3 CONFIG]
$ mycli config set shop.pageSize 25
✓ shop.pageSize = 25
```

A credencial entra pelo prompt ou por `--stdin`, sem argument. A leitura mascara o valor por padrão:

```
$ printf '%s\n' "$SHOP_TOKEN" | mycli credentials set shop.token --stdin
✓ credential "shop.token" saved to /home/ana/.mycli/credentials.json
$ mycli credentials get shop.token
shp_************op
$ mycli credentials get shop.token --json
{
  "ok": true,
  "command": "mycli credentials get",
  "data": { "name": "shop.token", "masked": "shp_************op", "length": 20 }
}
```

No CI, a master key vem da variável de ambiente, em 64 caracteres hex. O nome da variável sai do nome do binário em maiúsculas:

```
export MYCLI_MASTER_KEY="$(cat /run/secrets/mycli-master-key)"
mycli orders list --json
```

Regras que a skill segue

- **Preferência em settings, segredo no cofre.** O que dá acesso a um serviço vai para `credentials.json`, porque `settings.json` fica em texto claro.

- **Key nova no schema.** Com `key`, `type`, `default` e `description`, a key ganha conversão, padrão e linha em `config schema`. Fora do schema ela é aceita como texto livre, sem validação.
- **Nome de credencial em `service.purpose`.** Minúsculas, como `shop.token` ou `openai.api_key`. O cofre rejeita outro formato com código 2.
- **`getCredential` sem desvio.** A função lança `AppError` com código 4 quando o valor falta. Capturar esse erro para seguir sem o valor esconde da pessoa o passo que falta.
- **Segredo fora de saída.** O valor não aparece em argument, log, mensagem de erro, `--json` sem `--reveal`, resposta de tool ou frame da TUI. `mask()` cobre qualquer exibição.
- **Permissões 0600 e 0700.** Arquivos e pasta mantêm esses modos para outros usuários da máquina não lerem o cofre.
- **Variável antes do arquivo.** `<BIN>_MASTER_KEY` tem prioridade sobre `master.key`. Em CI e servidores, a variável evita gravar a chave em disco.
- **Rotação sem apagar o cofre.** `credentials rotate-key` decifra com a chave atual e grava uma nova em `master.key`. Com `--from-env`, a chave antiga vem de `master.key` e a nova vem da variável.
- **Testes isolados.** `<BIN>_HOME` aponta para uma pasta temporária, como em `tests/config.test.ts`, para o teste não ler o cofre da pessoa.

O cofre protege contra leitura casual, backup e varredura de disco. Ele não protege contra outro processo do mesmo usuário do sistema, que lê `master.key`. Para esse cenário, injete a variável a partir de um gerenciador de segredos.

O que ela entrega

A skill lista as keys ou credenciais adicionadas, os arquivos alterados e os testes executados. Valores de credenciais ficam fora da resposta.

Arquivo	Mudança
`src/config/settings.ts`	keys novas em `SETTINGS_SCHEMA`
`tests/config.test.ts`	conversão e padrão das keys
`docs/configuration.md`	tabela de keys e nomes de credenciais
`src/cli/commands/<nome>.ts`	`getCredential` e `ExitCode.CREDENTIALS`
`src/mcp/tools/<grupo>.ts`	leitura da credencial e `isError`
`specs/<spec>.md`	itens `#termfy-config` marcados com `[x]`

Próxima skill

Com a key e a credencial no lugar, implemente o acesso ao serviço com termfy-api. Para expor a leitura a agentes, siga com termfy-mcp.

termfy-api

A skill `termfy-api` liga o seu binário a um serviço HTTP externo. Ela cria um client único em `src/api/<service>.ts`, usado pelo CLI, pela TUI e pelo servidor MCP, com a credencial lida do cofre, a URL lida de `settings.json`, timeout por tentativa e retries limitados. No fim, você tem funções de domínio tipadas e um arquivo `tests/api-<service>.test.ts` que testa o client contra um servidor local feito com `Bun.serve`.

Quando usar

- Você vai consumir uma API pela primeira vez, como a de WhatsApp da uazapi. Ela pede autenticação, URL configurável e tratamento de erro.
- Um command, uma tool ou uma screen precisa de uma operação remota nova.

Não use esta skill para criar a família de commands que consome o client: para isso use termfy-command-set. Para declarar só uma setting ou credencial nova, sem HTTP, use termfy-config. Se o contrato do serviço ainda não está na spec, comece pela skill `termfy-spec`.

Como pedir

Para uma integração nova:

> Use a skill termfy-api para integrar a API da uazapi: listar instâncias com o admintoken e enviar texto pela instância escolhida em `--instance`.

Para o client do domínio padrão do guia:

> Use a skill termfy-api para criar o client da loja com `listOrders`, lendo `shop.baseUrl` e a credencial `shop.token`.

Para revisar um client que já existe:

> Use a skill termfy-api para revisar o client da loja: o POST de criação de pedido está com retry e pode duplicar pedidos.

O que a skill faz

1. Roda `git status --short` e lê a spec aberta em `specs/` (status `in_progress`): Surface, Behavior, Design e os itens `#termfy-api`.
2. Lê a documentação oficial do serviço para autenticação, endpoints, paginação, limites e idempotência. Contrato ausente volta para a spec.
3. Cria `src/api/<service>.ts`, separando montagem da request, execução com timeout, retries, validação e normalização.

4. Declara a key `<service>.baseUrl` em `SETTINGS_SCHEMA`, no arquivo `src/config/settings.ts`, com `key`, `type`, `default` e `description`.
5. Lê a credencial com `getCredential("<service>.token")` a cada chamada.
6. Cria `tests/api-<service>.test.ts` com um dublê em `Bun.serve` e uma pasta temporária no lugar de `~/.<bin>/`.
7. Liga o client aos consumidores previstos e atualiza `docs/configuration.md`. Roda `bun test` e `bun run typecheck`. Se a CLI mudou, roda `bun run opencli` antes de `bun run validate`. Marca `[x]` nos itens implementados e testados da spec.

Exemplo

A uazapi atende em `https://{subdomain}.uazapi.com`. O header `admintoken` autoriza `GET /instance/all`, que lista as instâncias com `name`, `token` e `status`. O header `token` opera uma instância: `GET /instance/status`, `POST /instance/connect` e `POST /send/text` com `number` e `text`. Só o admintoken vai para o cofre. O token da instância é resolvido em memória pelo nome recebido em `--instance <nome>`.

A setting entra em `SETTINGS_SCHEMA`. Você grava a URL real com `mycli config set uazapi.baseUrl https://minhaloja.uazapi.com`.

```
{
  key: "uazapi.baseUrl",
  type: "string",
  default: "https://subdomain.uazapi.com",
  description: "Base URL of the uazapi server.",
},
```

O client fica em `src/api/uazapi.ts`:

```
import { error } from "../app/errors.ts";
import { getCredential } from "../config/credentials.ts";
import { getPath, readSettings } from "../config/settings.ts";

export interface Instance { name: string; token: string; status: string }

function baseUrl(): URL {
  const raw = getPath(readSettings(), "uazapi.baseUrl");
  let url: URL; try { url = new URL(String(raw)); } catch {
    throw error.config("uazapi.baseUrl is not a valid URL.", { key:
"uazapi.baseUrl" });
  }
  const localTest = process.env.NODE_ENV === "test" && url.hostname === "127.0.0.1";
  if (url.protocol !== "https:" && !localTest) {
    throw error.config("uazapi.baseUrl must use HTTPS.", { key: "uazapi.baseUrl" });
  }
  return url;
}

async function call(method: "GET" | "POST", path: string, headers: Record<string,
string>,
  body: unknown, retry: boolean): Promise<unknown> {
  const url = new URL(path, baseUrl());
  const attempts = retry ? 3 : 1;
  for (let attempt = 1; attempt <= attempts; attempt++) {
    let response: Response;
    try {
      response = await fetch(url, {
        method, redirect: "error", signal: AbortSignal.timeout(5000),
        headers: { ...headers, "content-type": "application/json" },
        body: body === undefined ? undefined : JSON.stringify(body),
      });
    } catch {
      if (attempt < attempts) { await Bun.sleep(250 * 2 ** attempt); continue; }
      throw error.network("uazapi request failed.",
        { service: "uazapi", method, status: null, body: null, attempts: attempt });
    }
    if (response.status >= 500 && attempt < attempts) {
      await response.body?.cancel(); await Bun.sleep(250 * 2 ** attempt); continue;
    }
    if (!response.ok) {
      await response.body?.cancel();
```

```
      throw error.network(`uazapi returned HTTP ${response.status}.`,
        { service: "uazapi", method, status: response.status, body: "[omitted]",
attempts: attempt });
    }
    return response.status === 204 ? null : await response.json();
  }
  throw error.network("uazapi request failed.", { service: "uazapi", method,
attempts });
}

export async function listInstances(): Promise<Instance[]> {
  const admintoken = getCredential("uazapi.admintoken");
  const raw = await call("GET", "/instance/all", { admintoken }, undefined, true);
  if (!Array.isArray(raw)) throw error.network("Unexpected uazapi response.",
{ service: "uazapi" });
  return raw.map((item) => ({
    name: String(item?.name ?? ""), token: String(item?.token ?? ""), status:
String(item?.status ?? ""),
  }));
}

/** Sends one text message. No retry: repeating the POST sends it twice. */
export async function sendText(instance: string, number: string, text: string) {
  const found = (await listInstances()).find((i) => i.name === instance);
  if (!found) throw error.notFound(`Instance "${instance}" not found.`,
{ instance });
  const token = found.token;
  await call("POST", "/send/text", { token }, { number, text }, false);
  return { instance, number, sent: true };
}
```

O command `instances list` projeta só `name` e `status`, sem o token:

```
$ mycli credentials set uazapi.admintoken
$ mycli instances list
  loja     connected
  suporte  disconnected
$ mycli instances list --json
{
  "ok": true,
  "command": "mycli instances list",
  "data": {
    "count": 2,
    "instances": [
      { "name": "loja", "status": "connected" },
      { "name": "suporte", "status": "disconnected" }
    ]
  }
}
```

Sem o admintoken no cofre, a chamada para antes de qualquer request:

```
$ mycli send text --instance loja --number 5511999999999 --text "Pedido enviado"
error Credential "uazapi.admintoken" not found. Save it with: mycli credentials set
uazapi.admintoken [4 CREDENTIALS]
```

Regras que a skill segue

- **Um client por serviço.** O `fetch` fica no client, que devolve objetos tipados, sem o envelope `{ ok, command }`. Assim CLI, TUI e MCP recebem o mesmo erro e o mesmo retry. Cada camada monta a própria apresentação.
- **Token só pelo cofre.** O client chama `getCredential` com o nome literal. Token por argument, option ou campo de tool fica no histórico do shell e no log do agente.
- **Erro de credencial preservado.** O `AppError` de código 4 sobe intacto para que você receba a orientação de gravar a credencial ausente.
- **URL validada antes do token.** O protocolo e a origem são conferidos antes do envio. HTTP fica restrito ao loopback dos testes, para o token não trafegar em texto claro.
- **Timeout novo a cada tentativa.** `AbortSignal.timeout` é recriado por tentativa, porque um sinal já vencido abortaria a tentativa seguinte.
- **Retry só no que pode repetir.** GET ganha até três tentativas em falha de rede e 5xx. POST de envio fica com uma: repetir enviaria a mensagem duas vezes. 401, 403 e 404 viram código 5 sem retry, pois repetir não muda o resultado. O 429 segue `Retry-After` dentro do orçamento.

- **Detalhes sanitizados.** `details` traz serviço, método, status e corpo resumido. Token, cookies e headers de autenticação ficam de fora porque `details` aparece no `--json`.
- **Teste sem rede real.** O dublê roda em `127.0.0.1` com porta 0 e a pasta de configuração é temporária. O teste não depende da API real nem do cofre da pessoa.

O que ela entrega

A skill relata arquivos, operações, política de retry e timeout, key, nome da credencial e testes. Pendências aparecem sem valores de credenciais.

Arquivo	Mudança
`src/api/<service>.ts`	client novo com as funções de domínio
`src/config/settings.ts`	key `<service>.baseUrl` em `SETTINGS_SCHEMA`
`tests/api-<service>.test.ts`	testes com `Bun.serve` em loopback
`docs/configuration.md`	key e credencial documentadas
`specs/<spec>.md`	itens `#termfy-api` marcados com `[x]`

Próxima skill

Entregue commands, tools e screen que usam o client com termfy-command-set, ou um command isolado com termfy-cli.

termfy-database

A `termfy-database` dá ao binário um banco SQLite quando a spec pede que ele guarde registros entre execuções. Ela copia os stubs de `template/db/`, cria as tabelas por migration, escreve um repositório por tabela e liga esse repositório aos commands, tools e screens. No fim, você tem o arquivo `~/.<bin>/data.db`, migrations embutidas no executável, testes com banco em memória verdes e o binário compilado aplicando o schema fora do repositório.

Quando usar

Use esta skill quando a spec aberta descrever informações que precisam sobreviver ao fim do processo:

- um histórico de consultas que você quer rever depois;
- um cache local de pedidos baixados da API da loja;
- uma fila de tarefas que o binário processa aos poucos;
- notas ou registros próprios, criados pelo CLI e lidos pela TUI.

Não use para preferências nem para segredos. Preferência é key de `settings.json`, declarada pela termfy-config. Token e senha vão para o cofre, também pela `termfy-config`. Se a necessidade de guardar registros aparecer no meio de outra tarefa, volte à termfy-spec para registrar a mudança antes de implementar. O esqueleto recebe o banco apenas quando você solicita esse armazenamento.

Como pedir

Para criar o banco pela primeira vez:

> Use a skill termfy-database para implementar a spec 0003, que guarda notas locais na tabela `notes` e expõe os commands `db status` e `db migrate`.

Para evoluir uma tabela que já saiu em release:

> Use a skill termfy-database para acrescentar a coluna `pinned` à tabela `notes`, conforme a spec 0004, sem editar a migration já publicada.

Para levar o repositório aos três modos:

> Use a skill termfy-database para ligar o repositório de notas ao command `notes list`, a uma tool MCP e a uma screen da TUI.

O que a skill faz

1. Roda `git status --short` e lê a spec aberta em `specs/`. Ela confere que a seção Surface lista as tabelas com colunas e que o Design descreve a migration. Se faltar, ela devolve o trabalho à `termfy-spec`.
2. Na primeira vez no projeto, copia `template/db/database.ts` para `src/db/database.ts` e `template/db/migrations/index.ts` para `src/db/migrations/index.ts`, trocando os imports `../../src/` por `../`.
3. Troca o import da migration de notes no `index.ts` pela primeira migration da spec. A tabela de exemplo só fica quando a spec pede.
4. Cria a migration em `src/db/migrations/NNNN-<name>.ts`, com a próxima versão. Em seguida, registra a migration no `index.ts`.
5. Cria o repositório de cada tabela em `src/db/<resource>.ts`, a partir de `template/db/repository.ts`.
6. Quando a spec pede `db status` e `db migrate`, copia `template/db/command.ts` para `src/cli/commands/db.ts` e registra em `src/cli/commands.ts`.
7. Liga o repositório aos commands, tools e screens: abre o banco no `run` e fecha em `finally`.
8. Escreve os testes em `tests/<resource>.test.ts`, a partir de `template/db/example-test.ts`.
9. Documenta o banco em `docs/configuration.md` e os commands novos em `docs/cli.md`.
10. Roda `bun run build` e executa o binário de `dist/` numa pasta vazia, com `<BIN>_HOME` temporário, para provar que as migrations estão embutidas.
11. Confere à mão cada tabela de Surface contra os `CREATE TABLE` e `ALTER TABLE` das migrations.
12. Roda `bun run validate` e marca `[x]` nas tarefas `#termfy-database` concluídas na spec.

Exemplo

A spec pede uma tabela de notas. A migration é o stub `0001-create-notes.ts`, copiado para `src/db/migrations/`:

```
import type { Migration } from "./index.ts";

export const createNotes: Migration = {
  version: 1,
  name: "create_notes",
  up: `
    CREATE TABLE notes (
      id INTEGER PRIMARY KEY,
      title TEXT NOT NULL,
      body TEXT NOT NULL DEFAULT '',
      created_at TEXT NOT NULL
    ) STRICT;
    CREATE INDEX notes_created_at ON notes (created_at);
  `,
};
```

O registro importa cada migration de forma estática. `checkMigrations()` roda na importação e recusa lacuna de versão ou SQL vazio:

```
import { createNotes } from "./0001-create-notes.ts";

export const migrations: readonly Migration[] = checkMigrations([createNotes]);
```

O repositório em `src/db/notes.ts` concentra o SQL da tabela. Cada função recebe o `Database` aberto e usa parâmetros nomeados:

```
export function getNote(db: Database, id: number): Note {
  const row = db.query<NoteRow, { id: number }>(
    "SELECT id, title, body, created_at FROM notes WHERE id = $id",
  ).get({ id });
  if (!row) throw new AppError(ExitCode.NOT_FOUND, `Note ${id} not found.`);
  return toNote(row);
}
```

O teste abre um banco `:memory:` por caso. Assim os testes não usam `~/.<bin>/`:

```
describe("notes repository", () => {
  let db: Database;
  beforeEach(() => { db = openDatabase(":memory:"); });
  afterEach(() => db.close());

  test("adds, lists, reads and removes", () => {
    const note = addNote(db, { title: "First" }, new
Date("2026-11-10T12:00:00.000Z"));
    expect(listNotes(db)).toEqual([note]);
    expect(getNote(db, note.id).title).toBe("First");
    removeNote(db, note.id);
    expect(listNotes(db)).toEqual([]);
  });

  test("a database newer than the binary is refused", () => {
    db.run("PRAGMA user_version = 999");
    expect(() => migrate(db)).toThrow(AppError);
  });
});
```

Meses depois, a spec 0004 pede notas fixadas. A migration 0001 já saiu em release, então a skill cria `0002-add-notes-pinned.ts` com `ALTER TABLE notes ADD COLUMN pinned INTEGER NOT NULL DEFAULT 0` e acrescenta `addNotesPinned` à lista do `index.ts`. Ao abrir um banco na versão 1, `migrate()` copia o arquivo para `data.db.v1.bak` antes de aplicar a 2.

A seção Surface da spec lista a tabela. A linha vai para `### Storage` do `CLISPEC.md` quando a spec fecha:

```
| Tabela | Coluna | Tipo | Chave primária | Not null |
| --- | --- | --- | --- | --- |
| `notes` | `id` | `INTEGER` | sim | sim |
| `notes` | `title` | `TEXT` | não | sim |
| `notes` | `body` | `TEXT` | não | sim |
| `notes` | `created_at` | `TEXT` | não | sim |
```

Regras que a skill segue

- **Banco previsto na spec.** A pasta `src/db/` só aparece quando a spec descreve os registros a preservar, mantendo SQLite opcional no projeto.
- **Um arquivo por binário.** O banco fica em `~/.<bin>/data.db`, obtido por `databasePath()`. Com `<BIN>_HOME`, testes e CI usam uma pasta descartável.

- **Schema muda só por migration.** Uma migration por arquivo, versões a partir de 1, em sequência. Migration publicada não é editada, porque bancos já instalados aplicaram a versão antiga.
- **Import estático.** O registro importa cada arquivo para o build embutir o SQL. O binário compilado roda onde o repositório não existe, então ler `.sql` do disco quebraria a instalação.
- **`openDatabase()` aplica as pendentes.** Ele liga foreign keys, usa WAL e aplica cada migration na sua transação. Um banco mais novo que o binário é recusado com exit code 3, para que uma versão antiga não grave sobre um schema que ela não conhece.
- **`bun:sqlite`, sem ORM.** O driver incluído no Bun entra no executável, dispensando uma dependência adicional no `package.json`.
- **SQL só no repositório.** Commands, tools e screens chamam as mesmas funções. Os três modos leem e gravam do mesmo jeito.
- **Segredo fica no cofre.** Token lido por `getCredential()` não vai para tabela, porque `data.db` não é cifrado.
- **Erros viram `AppError`.** CONFIG (3) para banco ilegível ou schema incompatível, PERMISSION (7) para pasta sem acesso, NOT_FOUND (6) para linha ausente. Você recebe uma mensagem correspondente à falha.
- **Backup no lugar de rollback.** Não há migration de volta. Por isso `migrate()` copia o arquivo com `VACUUM INTO` antes de mudar um schema existente. Para voltar, você fecha o binário e troca `data.db` pelo backup.
- **Tabelas na superfície.** Nenhum script compara o schema com a spec. A skill confere lendo as migrations. A `termfy-review` repete a leitura.

O que ela entrega

No relatório final, a skill lista:

- as tabelas criadas e a versão do schema;
- os arquivos novos, em geral `src/db/database.ts`, `src/db/migrations/index.ts`, `src/db/migrations/NNNN-<name>.ts`, `src/db/<resource>.ts` e `tests/<resource>.test.ts`;
- `src/cli/commands/db.ts` e o registro em `src/cli/commands.ts`, quando a spec pede `db status` e `db migrate`;
- os commands, tools e screens que usam o banco;
- o resultado dos testes e do binário compilado numa pasta vazia;
- as alterações em `docs/configuration.md` e `docs/cli.md`;
- os itens `#termfy-database` marcados na spec.

Próxima skill

Com o repositório pronto, siga para a termfy-cli ou a termfy-command-set, que criam os commands que chamam as funções. Depois vêm a termfy-mcp, a termfy-tui e a termfy-test.

termfy-command-set

A skill `termfy-command-set` entrega uma família de commands de uma vez: `orders list`, `show`, `create`, `update` e `delete`, ou só os verbos que a spec aberta prevê. Ela extrai as operações para funções compartilhadas e cria os três adaptadores: o command no CLI, a tool espelhada no MCP e a screen de lista e detalhe na TUI. No fim, você tem um recurso disponível nos três modos, com testes em cada um e os itens da spec marcados.

Quando usar

- A spec aberta descreve um recurso inteiro, como pedidos, com verbos, tools e screen.
- Você quer que `mycli orders list`, a tool `orders_list` e a screen `orders` mostrem o mesmo resultado.
- Um recurso existe só no CLI e precisa ganhar tool e screen.

Para um command isolado, sem tool nem screen, use termfy-cli. Para só uma tool, use termfy-mcp. Para só uma tela, use termfy-tui. O client HTTP do serviço sai antes, pela termfy-api.

Como pedir

Para a família completa nos três modos:

> Use a skill termfy-command-set para entregar `orders list` e `orders show` com as tools `orders_list` e `orders_show` e a screen `orders`, conforme a spec aberta.

Para um recurso que só tem leitura no CLI:

> Use a skill termfy-command-set para acrescentar a tool `orders_list` e a screen `orders` à família `orders` que já existe no CLI.

Para verbos de gravação:

> Use a skill termfy-command-set para entregar `orders create` e `orders_create` com `--yes` e `dryRun`, como a spec define.

O que a skill faz

1. Roda `git status --short` e relaciona cada verbo e cada modo aos itens de Tasks e Tests da spec aberta em `specs/`.
2. Implementa as operações compartilhadas em módulo próprio sob `src/`, separado dos adaptadores. Se houver HTTP, usa o client da `termfy-api`.
3. Cria `src/cli/commands/orders.ts` a partir de `template/command/example.ts` e acrescenta o agrupador aos `subcommands` da raiz em `src/cli/commands.ts`.

4. Cria `src/mcp/tools/orders.ts` a partir de `template/tool/example.ts` e chama `registerOrdersTools` em `registerAll`, no `src/mcp/server.ts`.
5. Cria `src/tui/screens/orders.ts` a partir de `template/screen/example.ts` e registra a screen em `src/tui/screens/index.ts`. A tela segue o padrão lista e detalhe, com `table`, `stateView`, os diálogos da edição e as ações `n`, `Enter`, `d` e `r`. Se ela puder abrir primeiro, acrescenta `orders` em `tui.initialTab` e em `--tab`.
6. Escreve os testes: CLI em subprocesso, MCP com `Client` e TUI com `createTestRenderer` e `mountTui`.
7. Atualiza `docs/cli.md`, `docs/mcp.md` e `docs/tui.md`. Roda `bun run opencli`, `bun test`, `bun run typecheck` e `bun run docs:check`.
8. Marca `[x]` nos itens testados e roda `bun run validate` no fim.

Exemplo

A função `listOrders({ limit, status })` vem do client da loja. O command `orders list` valida as options e chama a função:

```
import { manifest } from "../../app/manifest.ts";
import { ExitCode } from "../../app/exit-codes.ts";
import { error } from "../../app/errors.ts";
import { table } from "../format.ts";
import type { Command } from "../registry.ts";
import { listOrders } from "../../api/shop.ts";

const listCommand: Command = {
  name: "list",
  description: "List shop orders.",
  options: [
    { name: "--limit", argument: { name: "n" }, default: "10",
      description: "Maximum number of orders, from 1 to 50. Default 10." },
    { name: "--status", argument: { name: "status", values: ["open", "paid",
"shipped"] },
      description: "Only orders with this status. Absent means every status." },
  ],
  exitCodes: [ExitCode.CONFIG, ExitCode.CREDENTIALS, ExitCode.NETWORK_OR_API],
  examples: [`${manifest.bin} orders list`, `${manifest.bin} orders list --status
open --json`],
  run: async (ctx) => {
    const limit = Number(ctx.options.limit ?? 10);
    if (!Number.isInteger(limit) || limit < 1 || limit > 50) {
      throw error.usage("--limit must be an integer from 1 to 50.");
    }
    const status = typeof ctx.options.status === "string" ? ctx.options.status :
undefined;
    const page = await listOrders({ limit, status });
    const data = { count: page.items.length, orders: page.items, nextCursor:
page.nextCursor };
    return { data, text: () => table(page.items.map((o) => [o.id, o.status])) };
  },
};

export const ordersCommand: Command = {
  name: "orders",
  description: "Query shop orders.",
  subcommands: [listCommand],
};
```

A tool `orders_list` chama a mesma função e devolve o mesmo objeto:

```
import { z } from "zod";
import { AppError } from "../../app/errors.ts";
import { listOrders } from "../../api/shop.ts";
import type { Registrar } from "../server.ts";

const description =
"Use when the user asks about shop orders. Returns count, orders and nextCursor.";

export function registerOrdersTools({ server, catalog }: Registrar): void {
  catalog.tools.push({ name: "orders_list", description });
  server.registerTool("orders_list", {
    title: "List orders",
    description,
    inputSchema: {
      limit: z.number().int().min(1).max(50).optional().describe("Maximum number of
orders. Default 10."),
      status: z.enum(["open", "paid", "shipped"]).optional().describe("Only orders
with this status."),
    },
  }, async ({ limit, status }) => {
    try {
      const page = await listOrders({ limit: limit ?? 10, status });
      const d = { count: page.items.length, orders: page.items, nextCursor:
page.nextCursor };
      return { content: [{ type: "text", text: JSON.stringify(d, null, 2) }],
structuredContent: d };
    } catch (e) {
      if (!(e instanceof AppError)) throw e;
      return { content: [{ type: "text", text: e.message }], isError: true };
    }
  });
}
```

Na screen `orders`, `mount` devolve um container síncrono com a `table` e o painel de detalhe. A carga roda em `onEnter` e atualiza a tabela com `setRows`. A geração descarta respostas que chegam depois de `onLeave`. As ações são `n` criar pelo `ctx.prompt`, `Enter` abrir o detalhe, `d` apagar com `ctx.confirm` e `danger`, `r` recarregar. Em `compact` a tela mostra uma visão por vez.

A saída nos dois formatos do CLI:

```
$ mycli orders list --status open
  o1  open
  o7  open
$ mycli orders list --status open --json
{
  "ok": true,
  "command": "mycli orders list",
  "data": {
    "count": 2,
    "orders": [{ "id": "o1", "status": "open" }, { "id": "o7", "status": "open" }],
    "nextCursor": null
  }
}
$ mycli orders list --limit 0
error --limit must be an integer from 1 to 50. [2 USAGE]
Use "mycli orders list --help".
```

Regras que a skill segue

- **Só os verbos da spec.** A skill gera o subconjunto combinado na Surface. Verbo criado por simetria vira código sem teste nem dono.
- **Uma função, três adaptadores.** A tool não chama `Command.run` e a screen não executa a CLI em subprocesso. Formatação e protocolo ficam nos adaptadores para o domínio responder igual nos três modos.
- **Defaults aplicados em `run`.** `OptionDefinition.default` só documenta. O parser não preenche `ctx.options`, então `run` resolve o padrão.
- **`--json` não se repete.** Ela é global e chega a todas as folhas.
- **Exit codes por folha.** Cada verbo declara os seus em `exitCodes`. O agrupador não repassa códigos para as folhas.
- **Tool com Zod descrito.** Cada campo tem `.describe()` e a tool entra em `catalog.tools`, senão `mcp --list` não a mostra.
- **Gravação sem prompt na tool.** Confirmação ou `dryRun` são implementados na própria tool, porque um agente não responde a prompt de terminal.
- **Falha não vira lista vazia.** Erro aparece como erro no CLI, como `isError: true` na tool e como estado de erro na screen.
- **Tela em lista e detalhe.** A screen usa `table` para a coleção e `keyValueTable` no detalhe, com `stateView` para vazio, carregando e erro. `n` cria pelo `ctx.prompt`, `Enter` abre o detalhe, `d` apaga com `ctx.confirm` e `danger`, `r` recarrega.
- **Mesma expectativa nos testes.** Os três modos usam a mesma fonte local controlada, sem rede real nem o cofre da pessoa.

O que ela entrega

A skill relata verbos e modos entregues, arquivos, contratos de `data`, itens marcados e o resultado da validação. Inclui exemplos em texto e JSON, os nomes das tools e um frame da screen.

Arquivo	Mudança
`src/cli/commands/orders.ts`	agrupador e folhas novos
`src/cli/commands.ts`	`ordersCommand` nos `subcommands` da raiz
`src/mcp/tools/orders.ts`	`registerOrdersTools` com as tools
`src/mcp/server.ts`	chamada em `registerAll`
`src/tui/screens/orders.ts`	screen de lista e detalhe
`src/tui/screens/index.ts`	screen na lista de abas
`tests/` e `docs/`	testes dos três modos e capítulos atualizados
`docs/opencli.json`	regenerado por `bun run opencli`

Próxima skill

Para ajustar detalhes de cada modo, siga com termfy-cli, termfy-mcp ou termfy-tui. Se a família trouxe settings ou credenciais novas, confira com termfy-config.

termfy-test

A `termfy-test` escreve e mantém os testes Bun do binário a partir dos itens de `## Tests` da spec aberta. Ela cobre os três modos (CLI em subprocesso, MCP por stdio com o SDK real, TUI por frame) e a configuração. No fim, você tem cada item de aceite ligado a pelo menos um teste verde, `bun run typecheck` e `bun test` passando e os itens cobertos marcados com `[x]` na spec.

Quando usar

Use esta skill nestas situações:

- a spec 0002 criou o command `orders list` e os itens de `## Tests` ainda estão `[ ]`;
- um bug apareceu na tool `orders_list` e você quer reproduzi-lo com uma assertion antes de corrigir;
- uma mudança de contrato (campo novo em `data`, exit code diferente) deixou assertions antigas desatualizadas;
- a screen `orders` ganhou uma tecla nova e o frame precisa de teste.

Não use para criar funcionalidade. Um comportamento que não está na spec passa primeiro pela termfy-spec. Para auditar se os testes existentes provam a entrega, use a termfy-review, que relata sem editar. Os testes das tabelas SQLite saem da termfy-database, que já traz o stub `template/db/example-test.ts`.

Como pedir

Para cobrir uma entrega nova:

> Use a skill termfy-test para escrever os testes de aceite da spec 0002: `orders list` em texto e em `--json`, a tool `orders_list` e a screen `orders`.

Para reproduzir uma falha:

> Use a skill termfy-test para reproduzir o erro: `orders list` sem `shop.token` sai com exit code 1 em vez de 4.

Para atualizar depois de uma mudança de contrato:

> Use a skill termfy-test para ajustar os testes de `orders list` ao campo `count` que a spec 0005 acrescentou ao `data`.

O que a skill faz

1. Roda `git status --short` e delimita a entrega pela tarefa da spec com a tag `#termfy-test` ou pela tarefa de implementação que pede o teste.
2. Relaciona cada item de `## Tests` a pelo menos um teste. Para cada comportamento alterado, escolhe os casos de sucesso, falha e limite que se aplicam (command, tool, screen ou configuração).
3. Escreve os testes no arquivo da área: `tests/cli.test.ts`, `tests/mcp.test.ts`, `tests/tui.test.ts` ou `tests/config.test.ts`. Para uma correção, reproduz a falha antes com uma assertion observável.
4. Roda o arquivo alterado, por exemplo `bun test tests/cli.test.ts`. Depois roda `bun run typecheck` e `bun test`.
5. Roda `bun run validate` e separa falha da entrega de problema que já existia antes.
6. Com a entrega concluída e os testes verdes, troca `[ ]` por `[x]` na tarefa e nos itens de `## Tests` cobertos, no arquivo da spec.

Exemplo

A spec 0002 tem este item de aceite:

```
## Tests

- [ ] `mycli orders list --json` sem `shop.token` sai com 4 e envelope de erro.
```

A skill escreve o teste em `tests/cli.test.ts`, no modelo de subprocesso. `HOME_VARIABLE` aponta para uma pasta temporária, então o cofre real da máquina fica fora do teste:

```
/** Runs orders list without a stored token in an isolated subprocess. */
import { test, expect } from "bun:test";
import { mkdtempSync, rmSync } from "node:fs";
import { tmpdir } from "node:os";
import { join } from "node:path";
import { HOME_VARIABLE, MASTER_KEY_VARIABLE } from "../src/config/paths.ts";
import { ExitCode } from "../src/app/exit-codes.ts";
import { manifest } from "../src/app/manifest.ts";

test("orders list without shop.token exits with CREDENTIALS", async () => {
  const directory = mkdtempSync(join(tmpdir(), "mycli-cli-"));
  const env = { ...process.env, [HOME_VARIABLE]: directory, NO_COLOR: "1" };
  delete env[MASTER_KEY_VARIABLE];
  const child = Bun.spawn([process.execPath, "run", "src/main.ts", "orders",
"list", "--json"], {
    cwd: join(import.meta.dir, ".."), env,
    stdin: "ignore", stdout: "pipe", stderr: "pipe",
  });
  const timeout = setTimeout(() => child.kill(), 4000);
  try {
    const [stdout, code] = await Promise.all([new Response(child.stdout).text(),
child.exited]);
    expect(code).toBe(ExitCode.CREDENTIALS);
    const result = JSON.parse(stdout);
    expect(result.ok).toBe(false);
    expect(result.command).toBe(`${manifest.bin} orders list`);
    expect(result.error.name).toBe("CREDENTIALS");
  } finally {
    clearTimeout(timeout);
    child.kill();
    await child.exited;
    rmSync(directory, { recursive: true, force: true });
  }
});
```

Para a tool, o teste usa `Client` e `StdioClientTransport` do SDK. Ele conversa com o servidor real por stdio e confere a descoberta:

```
await client.connect(transport);
const catalog = await client.listTools();
expect(catalog.tools.some((tool) => tool.name === "orders_list")).toBe(true);
```

Para a screen, monta a TUI com `createTestRenderer` e `mountTui`, envia a tecla com `mockInput.pressKey`, chama `renderOnce()` e confere um trecho de `captureCharFrame()`. O teste não espera a Promise de `mountTui` antes de interagir, porque ela só termina quando o renderer é destruído.

Com os testes verdes, a skill marca o item:

```
- [x] `mycli orders list --json` sem `shop.token` sai com 4 e envelope de erro.
```

Regras que a skill segue

- **Um arquivo por área, em inglês.** Nomes de arquivo, testes, variáveis e mensagens ficam em inglês, como o resto do código do kit.
- **CLI em subprocesso.** `Bun.spawn` com `src/main.ts`, `HOME_VARIABLE` numa pasta temporária e `NO_COLOR=1`. O teste vê o binário como a pessoa vê, sem cores que atrapalham a comparação.
- **Exit code antes da saída.** Cada chamada, inclusive as de preparo, tem o code conferido com `ExitCode`, pois um retorno JSON válido também precisa informar o código previsto pelo contrato.
- **Contrato público, não implementação.** As assertions conferem `data`, `text` e o envelope. Copiar a lógica do command no teste só prova que o código é igual a ele mesmo.
- **MCP sem rede.** O teste usa o SDK real, mas tools de rede recebem respostas locais controladas. Assim a suíte roda no CI sem cadastro externo.
- **TUI por trecho.** Um trecho do frame e uma transição bastam. Snapshot do frame inteiro quebra a cada ajuste de layout.
- **Fixtures sintéticas.** `MASTER_KEY_VARIABLE` sai do ambiente do subprocesso. Testes de config e TUI restauram as variáveis no `finally`, para que um teste não contamine o próximo.
- **Recursos fechados.** Subprocessos, clientes e renderers fecham mesmo quando uma assertion falha. Casos que mudam `process.env` não rodam com `test.concurrent`.
- **Assertion não enfraquece.** Se o contrato falha, a skill descreve a falha. Ela só mexe no código de produção quando o pedido inclui a correção.
- **Marca na spec, não no `CLISPEC.md`.** Fechar a spec e mudar o status para `done` é trabalho da `termfy-spec`. Um item sem teste fica `[ ]`.

O que ela entrega

No relatório final, a skill lista:

- os arquivos de `tests/` alterados;
- os comportamentos cobertos, com o texto da tarefa e dos itens de `## Tests` relacionados;

- os comandos executados (`bun test tests/<área>.test.ts`, `bun run typecheck`, `bun test`, `bun run validate`);
- a quantidade de testes que passaram e falharam;
- as limitações restantes e os casos que não se aplicam, com o motivo.

A spec aberta recebe `[x]` nos itens cobertos. Valores reais de credenciais não aparecem no relatório. Quando o pedido é só de testes, a skill relata ao responsável pela implementação o item apto a ser marcado.

Próxima skill

Com a entrega coberta, siga para a termfy-review, que confronta o código e os testes com o Behavior e o Design da spec.

termfy-review

A `termfy-review` audita uma entrega contra a spec aberta (Behavior, Design, Surface, Tasks e Tests) e faz a conferência manual do `CLISPEC.md` e das specs contra o código. Ela não edita arquivos. No fim, você tem um relatório com os achados ordenados de P0 a P3, cada um com `arquivo:linha` e a correção sugerida, a lista das verificações executadas e a resposta sobre a spec poder ou não ser fechada.

Quando usar

Use esta skill nestas situações:

- a implementação da spec 0002 (`orders list`, `orders_list` e a screen `orders`) terminou e os testes passaram;
- você vai fazer merge de uma branch e quer uma leitura independente antes;
- a spec tem todos os itens `[x]` e você precisa saber se ela pode fechar;
- você suspeita que um segredo, como `shop.token`, aparece em alguma saída.

Use a revisão para obter o relatório das falhas e encaminhe as correções à skill da área: termfy-cli, termfy-mcp, termfy-tui ou termfy-config. Para escrever testes que faltam, use a termfy-test. Para fechar a spec e gravar o `CLISPEC.md`, use a termfy-spec.

Como pedir

Para auditar uma entrega completa:

> Use a skill termfy-review para auditar a spec 0002 e dizer se ela pode ser fechada.

Para uma checagem dirigida:

> Use a skill termfy-review para conferir se `shop.token` vaza em `orders list --json`, na tool `orders_list` ou no frame da screen `orders`.

Para revisar antes de um merge:

> Use a skill termfy-review para revisar o diff desta branch contra a spec aberta e o `CLISPEC.md`, sem alterar nada.

O que a skill faz

1. Roda `git status --short` e lê o diff com `git diff --stat`, `git diff` e `git diff --cached`. Fixa os arquivos e os itens sob revisão. Lê também os arquivos novos, porque `git diff` não mostra o conteúdo deles.

2. Lê a spec inteira e confere a linha `Status:` com `rg -n '^Status:|^- \[[xX ]\]' specs/0002-*.md`.
3. Roda `bun run typecheck` e `bun test`. Para cada item `[x]` de Tasks e Tests, localiza um teste que exercita a entrega. Procura testes desligados com `rg -n '\b(test|it|describe)\.(skip|todo|only)|expect\(' tests`.
4. Confronta Behavior e Design com o código: saídas em texto e em `--json`, mensagens de erro, exit codes, teclas, tools, arquivos criados e funções compartilhadas entre CLI, MCP e TUI.
5. Faz a conferência manual do `CLISPEC.md`. As buscas `rg -n 'name: "' src/cli/commands.ts src/cli/commands`, `rg -n 'registerTool|registerResource|registerPrompt' src/mcp`, `rg -no 'getCredential\("[^"]+"\)' src` e `rg -n 'CREATE TABLE|ALTER TABLE' src` listam o lado do código. A skill compara cada nome com as tabelas de Surface e com `### Storage`.
6. Verifica descriptions de commands, options e arguments, roda `bun run opencli:check` e `bun run docs:check`, segue cada `getCredential` até os consumidores, procura `console.log` alcançado pelo MCP e cores literais fora de `src/tui/theme.ts`.
7. Classifica os achados pela tabela de gravidade e propõe a menor correção suficiente, sem executá-la.
8. Roda `git status --short` de novo e relata qualquer arquivo que os testes tenham gerado, sem reverter nada.

Exemplo

A revisão da spec 0002 do `mycli` termina com este relatório. Os achados são ilustrativos e mostram o formato, com uma linha por achado:

```
1. [P1] src/mcp/tools/orders.ts:31 | Um console.log no callback de orders_list
escreve em stdout e quebra a conversa JSON-RPC. | Remover o log ou enviar o
diagnóstico, sem segredo, para stderr.
2. [P2] src/cli/commands/orders.ts:18 | A option --status não tem description; o --
help e o opencli.json saem incompletos. | Acrescentar a description e rodar bun run
opencli.
3. [P2] specs/0002-orders.md:41 | O item "orders list --json sem shop.token sai com
4" está [x], mas nenhum teste confere o exit code. | Desmarcar o item ou pedir o
teste à termfy-test.
4. [P3] src/tui/screens/orders.ts:22 | Cor literal "#ff5555" fora de src/tui/
theme.ts. | Trocar por uma cor do tema.

Verificações
- bun run typecheck: exit 0
- bun test: 48 pass, 0 fail
- bun run opencli:check: exit 1 (ver achado 2)
- bun run docs:check: exit 0
- Conferência manual: Surface da spec bate com o registry; shop.token aparece em
Credentials.

Spec 0002: não pode fechar. Itens pendentes: achados 1 a 3.
```

Repare que o achado 3 aponta um item marcado sem teste. A suíte passou, mas um `[x]` precisa de teste verde que exercite aquela entrega.

Regras que a skill segue

- **Revisão não edita.** Sem build, commit, instalação ou `bun run opencli`. Ela também não roda `bun run validate`, porque ele inclui build. Assim o repositório chega ao fim da revisão como estava no começo.
- **Behavior e Design são a régua.** Desvio sem registro em `## Notes` da spec é achado. Preferência pessoal de estilo não é defeito.
- **`[x]` exige teste.** Um item marcado precisa de teste verde no código atual. Resultado antigo ou teste sem assertion pertinente não aprova.
- **Busca aponta candidato.** A skill lê o código e segue as chamadas antes de relatar um vazamento, porque um match de `rg` sozinho não prova nada.
- **Segredo não vai para o relatório.** Se encontrar um token gravado, ela descreve o lugar sem copiar o valor.
- **Sem serviço externo.** Falhas são reproduzidas com testes locais e fixtures sintéticas, fora da pasta real de configuração da pessoa.

- **Limitação declarada.** Se uma checagem não pôde rodar, o relatório diz qual e o impacto. A entrega não é aprovada enquanto faltar confirmação essencial.
- **Conferência manual é daqui.** Nenhum script lê `CLISPEC.md` nem `specs/`. A revisão é o momento de comparar Surface, Credentials, `### Storage` e o índice `## Specs` com o código.
- **Gravidade pelo efeito.** P0 para segredo exposto ou perda de registros ampla, P1 para fluxo principal inutilizável, P2 para contrato descumprido localizado, P3 para apresentação e manutenção. Ocorrências da mesma causa viram um achado só.

O que ela entrega

O relatório traz, nesta ordem:

- os achados de P0 a P3, cada um com gravidade, `arquivo:linha`, comportamento observado, consequência e correção sugerida;
- as verificações executadas, com comando e exit code;
- os itens da spec ainda pendentes;
- a resposta sobre a spec poder ser fechada pelo fluxo da `termfy-spec`;
- os pontos que precisam de resposta da pessoa.

Sem achados, a skill diz isso e cita o que conferiu, sem prometer ausência de falhas fora do escopo. Nenhum arquivo do repositório muda. Fixtures temporárias criadas pelos testes são conferidas e removidas.

Próxima skill

Corrigidos os achados pelas skills da área, siga para a termfy-docs, que relê a documentação embutida antes do fechamento da spec.

termfy-docs

A `termfy-docs` melhora a documentação embutida do binário, a pasta `docs/` que entra no executável e sai por `--skill`, pela tool MCP `skill` e pela resource `<bin>://skill`. Ela confere a cobertura contra o registry com `bun run docs:check`, relê tudo como um agente que só tem o binário e regenera `docs/opencli.json` quando o registry muda. No fim, você tem `docs:check` verde, exemplos que rodam de verdade e uma leitura completa do `--skill` sem ponto onde o agente trave.

Quando usar

Use esta skill nestas situações:

- `bun run docs:check` falhou depois que o command `orders list` entrou no registry;
- a entrega da spec 0002 criou a tool `orders_list` e a screen `orders`. Mas `docs/` ainda não fala delas;
- um agente chamou `mycli --skill` e não descobriu que `orders list` exige `shop.token`;
- você quer reescrever uma seção confusa de `docs/configuration.md`.

Não use para o guia do framework nem para explicar as skills: `docs/` descreve o binário. Para mudar o nome, o título ou os créditos que aparecem nos textos, use a termfy-brand. Para criar o command que falta, use a termfy-cli; a documentação vem depois que ele existe no registry.

Como pedir

Para corrigir uma falha de cobertura:

> Use a skill termfy-docs para fazer o `docs:check` passar depois do command `orders list` e da tool `orders_list`.

Para revisar a leitura como agente:

> Use a skill termfy-docs para reler o `--skill` do mycli inteiro e completar o que falta para um agente usar `orders list` sem perguntar.

Para cumprir as tarefas da spec:

> Use a skill termfy-docs para concluir os itens `#termfy-docs` da spec 0002 e documentar a screen `orders` em `docs/tui.md`.

O que a skill faz

1. Roda `git status --short`, `bun run docs:check` e `bun run opencli:check` para saber o ponto de partida.

2. Lê `bun run dev -- --skill` do começo ao fim e anota onde um agente travaria: conceito sem explicação, command sem exemplo, exit code sem causa, key sem padrão.
3. Corrige primeiro as faltas do `docs:check`. Cada entrada do registry ganha linha na tabela da seção certa, com um parágrafo de explicação quando introduzir um conceito novo.
4. Reescreve as seções fracas seguindo a forma de cada seção (o que é, exemplo, tabela, falhas, como estender). Prefere cortar a inflar.
5. Roda `bun run opencli` se o registry mudou e repete `bun run docs:check`.
6. Relê `bun run dev -- --skill <topic>` de cada topic alterado, por exemplo `bun run dev -- --skill cli`.
7. Roda `bun run validate`.
8. Marca `[x]` nos itens `#termfy-docs` concluídos da spec aberta.

Exemplo

Depois da spec 0002, o `docs:check` aponta o que falta:

```
✗ docs/cli.md does not mention command "orders"
✗ docs/cli.md does not mention command "orders list"
✗ docs/mcp.md does not mention tool "orders_list"
✗ docs/tui.md does not mention screen "orders"
```

A skill acrescenta uma seção a `docs/cli.md`. Ela abre com a pergunta que responde, mostra o exemplo antes da tabela e diz o que falha:

````
## Pedidos da loja

`orders list` lista os pedidos da loja. Exige `shop.token` gravado com
`{{bin}} credentials set shop.token`.

```bash
{{bin}} orders list --json
```

| Command | Função |
| --- | --- |
| `orders list` | Lista os pedidos. Chama a API da loja. |

Sem `shop.token`, sai com 4 (`CREDENTIALS`). Falha de rede ou resposta
inesperada da API sai com 5 (`NETWORK_OR_API`).
````

O texto usa `{{bin}}` no lugar de `mycli`. O binário e o `docs:check` trocam o placeholder pelo nome real, então `bun run init` não precisa editar `docs/`. A lista exata de options não entra à mão: ela vem de `--skill reference`, gerada do registry.

Na mesma passada, `docs/mcp.md` ganha `orders_list` na tabela de tools e `docs/tui.md` ganha a screen `orders` com suas teclas. Como o registry mudou, a skill roda `bun run opencli`. O `docs:check` termina com:

```
✓ docs/ covers every command, screen, tool, resource, prompt, setting and exit code
```

Regras que a skill segue

- **Referência vem do registry.** A lista de options sai de `--skill reference`. Copiá-la à mão em `docs/cli.md` cria uma segunda fonte que envelhece a cada mudança.
- **Toda entrada tem menção.** Command, screen, tool, resource, prompt e key de configuração aparecem na tabela da seção certa. É o que o `docs:check` confere.
- **Cada seção responde a uma pergunta.** O que é, como entrar, o que sai, o que pode falhar e com qual exit code. Essas informações orientam você e o agente durante o uso do binário.
- **Exemplos executáveis.** Cada exemplo começa pelo nome do binário, cabe em uma linha e precisa funcionar para orientar a execução pelo agente.
- **Placeholders no lugar de nomes.** `{{bin}}`, `{{title}}`, `{{version}}` e `{{BIN}}` mantêm `docs/` válido depois do `bun run init`.
- **Código em inglês, prosa no idioma do projeto.** Nomes de commands, keys e campos aparecem como no código, para o agente copiar sem traduzir.
- **`opencli.json` não se edita à mão.** Ele é regenerado com `bun run opencli` a cada mudança do registry.
- **`docs/` fala do binário.** O framework e as skills ficam no guia do Termfy, para o executável carregar apenas as instruções de uso do binário.
- **Texto por tópico.** Um topic acima de 200 linhas deve ser dividido ou revisto para remover repetições. Repetir a referência gerada aumenta o consumo de tokens sem acrescentar instruções ao agente.

O que ela entrega

No relatório final, a skill lista:

- os arquivos de `docs/` alterados (`docs/README.md`, `docs/cli.md`, `docs/tui.md`, `docs/mcp.md`, `docs/configuration.md` ou `docs/exit-codes.md`);
- as faltas de cobertura corrigidas;
- as seções reescritas;

- `docs/opencli.json` regenerado, quando o registry mudou;
- os comandos de validação executados e o resultado de cada um.

A spec aberta recebe `[x]` nos itens `#termfy-docs` concluídos. O fechamento da spec fica com a `termfy-spec`.

Próxima skill

Com a documentação em dia e todos os itens da spec marcados, feche a spec pela termfy-spec. Com as specs da versão fechadas, siga para a termfy-release.

termfy-release

A `termfy-release` publica uma versão do binário derivado. Ela escolhe o número pelo SemVer, move as notas de `Unreleased` para a seção da versão no `CHANGELOG.md`, roda `bun run validate`, cria o commit e a tag e acompanha o workflow `.github/workflows/release.yml`. No fim, você tem uma GitHub Release com os cinco executáveis e o `SHA256SUMS`, mais a prova de que um asset baixado responde `--version` com o número publicado.

Quando usar

Use esta skill nestas situações:

- as specs 0002 e 0003 do `mycli` estão `done` e você quer publicar o `orders list` para os operadores da loja;
- um bug corrigido precisa chegar às instalações existentes do binário;
- o workflow de uma versão anterior falhou e a correção precisa de uma versão nova;
- você quer só preparar a versão (número, changelog, validação) sem enviar nada ainda.

Não use com spec aberta sem antes falar com a pessoa: feche a spec pela termfy-spec primeiro. Para publicar o próprio framework Termfy ou o starter kit, siga o fluxo de cada repositório; esta skill roda no repositório do binário derivado. Para ligar o binário publicado aos agentes, use a termfy-mcp-client depois.

Como pedir

Para publicar de ponta a ponta:

> Use a skill termfy-release para publicar o mycli com as specs 0002 e 0003 fechadas.

Para preparar sem enviar:

> Use a skill termfy-release para preparar a versão do mycli: escolha o número, atualize o changelog e rode a validação, sem commit nem tag.

Para refazer depois de uma falha:

> Use a skill termfy-release para publicar uma correção do mycli depois que o workflow da v0.6.2 falhou na compilação.

O que a skill faz

1. Confere `git status --short`, `git branch --show-current`, `git remote -v` e `gh auth status`. Trabalha na `main` atualizada com `git fetch origin`, sem descartar mudanças locais.

2. Lê o changelog, o diff desde a última tag e o índice `## Specs` do `CLISPEC.md`. Escolhe X.Y.Z e confere que a tag não existe com `git tag --list "$release_tag"` e `git ls-remote --tags origin "refs/tags/$release_tag"`.
3. Atualiza `version` no `package.json`. Move as notas de `Unreleased` para `## [X.Y.Z] - YYYY-MM-DD` no `CHANGELOG.md`.
4. Roda `bun run opencli` se o registry mudou. Depois roda `bun run validate` e corrige as falhas antes de seguir.
5. Revisa o diff e faz o commit `release: <bin> vX.Y.Z` só com os arquivos da versão.
6. Cria a tag `vX.Y.Z` nesse commit e faz push de `main` e da tag específica, com `git push origin main` e `git push origin "$release_tag"`.
7. Localiza a execução do workflow com `gh run list --workflow release.yml --commit "$release_commit"` e acompanha com `gh run watch "$release_run_id" --exit-status`.
8. Confere a release com `gh release view "$release_tag"`, baixa o asset da máquina com `gh release download`, valida o SHA-256 e roda `--version`.
9. Se o workflow falhar, lê o log com `gh run view "$release_run_id" --log-failed`, corrige a causa e publica outra versão.

Exemplo

O `mycli` estava na 0.6.1. Desde então, a spec 0002 acrescentou o command `orders list` e a tool `orders_list`, sem quebrar nada. Capacidade nova compatível antes da 1.0.0 pede minor, então a versão é 0.7.0.

O `CHANGELOG.md` fica assim, em português e no formato Keep a Changelog:

```
## [Unreleased]

## [0.7.0] - 2026-11-12

### Adicionado

- Command `orders list` e tool MCP `orders_list`, que listam os pedidos
  da loja com a credencial `shop.token` (spec 0002).

## [0.6.1] - 2026-11-03
```

O workflow procura o cabeçalho `## [0.7.0]` e usa o texto até a próxima versão como notas da release. Por isso a seção não pode ficar vazia.

Os comandos de envio, com as variáveis da tarefa:

```
release_bin='mycli'
release_version='0.7.0'
release_tag="v$release_version"
git commit -m "release: $release_bin $release_tag"
release_commit=$(git rev-parse HEAD)
git tag "$release_tag" "$release_commit"
git push origin main
git push origin "$release_tag"
```

A tag `v0.7.0` dispara o workflow pelo gatilho `v*`. Ele instala as dependências, valida, compila com `bun run build:all` e publica estes assets, com os nomes definidos em `scripts/build.ts`:

Alvo	Asset
Linux x64	`mycli-linux-x64`
Linux arm64	`mycli-linux-arm64`
macOS x64	`mycli-darwin-x64`
macOS arm64	`mycli-darwin-arm64`
Windows x64	`mycli-windows-x64.exe`
Checksums	`SHA256SUMS`

Numa máquina Linux x64, a conferência final é esta:

```
release_download=$(mktemp -d)
release_asset="$release_bin-linux-x64"
gh release download "$release_tag" --pattern "$release_asset" --pattern SHA256SUMS
--dir "$release_download"
(cd "$release_download" && sha256sum --check --ignore-missing SHA256SUMS)
chmod u+x "$release_download/$release_asset"
"$release_download/$release_asset" --version
```

A última linha imprime `mycli 0.7.0`.

Regras que a skill segue

- **SemVer pelo efeito.** Contrato incompatível pede major, capacidade nova compatível pede minor, correção pede patch. Vale o maior incremento do conjunto, porque um script antigo quebra pelo item mais grave.

- **Antes da 1.0.0, a política do projeto.** Sem política declarada, minor para incompatibilidade ou capacidade nova e patch para correção. A faixa 0.x não esconde migração: o changelog descreve a chamada antiga e a nova.
- **Changelog em português, Keep a Changelog.** Versões anteriores ficam como estão. A data é a da publicação real.
- **Validação antes do envio.** Sem `bun run validate` com exit code 0, não há commit de release. O workflow repete essa validação e uma falha após a publicação da tag exige outra versão para a correção.
- **Specs fechadas primeiro.** Uma spec `in_progress` com código no repositório indica entrega incompleta. Se ela não puder fechar, a skill pergunta se a versão sai parcial e registra isso no changelog.
- **Commit só da versão.** Nada de `git add .` sem revisão. Mudanças de outras tarefas ficam fora.
- **Tag publicada não se mexe.** Sem mover, apagar ou recriar tag e sem force push. Cada download da v0.7.0 precisa entregar o mesmo arquivo. Correção ganha outra versão, mesmo depois de workflow falho.
- **Push da tag específica.** `git push origin "$release_tag"` envia só a tag da versão, sem levar tags de outras entregas.
- **Concluída só com o asset testado.** A skill confere o workflow, os cinco executáveis, os checksums e o `--version` de um asset baixado. O build local não substitui o que foi publicado.
- **Sem release manual para esconder falha.** Se o workflow falhar, a correção passa pelo projeto e por uma versão nova. Se a mesma causa persistir, a skill para e relata o diagnóstico.

O que ela entrega

No relatório final, a skill informa:

- a versão anterior e a nova, com o motivo do incremento;
- o commit, a tag e a URL da release;
- o resultado do `bun run validate` e do workflow;
- os nomes dos assets publicados;
- o resultado da verificação do SHA-256 e do `--version`.

Os arquivos que mudam no repositório são `package.json`, `CHANGELOG.md` e, quando o registry mudou, `docs/opencli.json`. Num pedido só de preparação, a skill para com esses arquivos prontos e validados, sem commit nem tag. Se algo falhar, o relatório diz a etapa e traz o trecho útil do log.

Próxima skill

Com a release publicada, siga para a termfy-mcp-client, que configura o executável nos clientes de agente e valida a conexão MCP.

termfy-mcp-client

A `termfy-mcp-client` configura um binário Termfy, publicado ou em desenvolvimento, nos clientes de agente Claude Code, Codex, Cursor, OpenCode e Gemini CLI. Ela não cria tools nem muda o servidor: o modo `mcp` já existe no executável. No fim, você tem a entrada do servidor no arquivo do cliente, no escopo pedido, sem nenhum segredo gravado nela. Você também tem a prova de que a conexão funciona: o Inspector e o próprio cliente chamam a tool `skill` e recebem a seção CLI da documentação.

Quando usar

Use esta skill nestas situações:

- a release v0.7.0 do `mycli` saiu e você quer usar a tool `orders_list` no Claude Code;
- os desenvolvedores precisam da mesma entrada no projeto, num `.mcp.json` compartilhado;
- você está desenvolvendo e quer que o Cursor rode `src/main.ts` direto, sem compilar a cada mudança;
- o cliente mostra o servidor `mycli` com erro, timeout ou sem tools.

Não use para criar ou alterar tools, resources e prompts: isso é da termfy-mcp. Para gravar `shop.token` no cofre ou declarar uma credencial nova, use a termfy-config. Para publicar o executável que o cliente vai rodar, use antes a termfy-release.

Como pedir

Para um binário publicado:

> Use a skill termfy-mcp-client para configurar o mycli publicado no Claude Code, no escopo de projeto, e validar a tool `orders_list`.

Para desenvolvimento local:

> Use a skill termfy-mcp-client para ligar o mycli em desenvolvimento ao Cursor, rodando `src/main.ts` com Bun.

Para diagnosticar uma falha:

> Use a skill termfy-mcp-client para descobrir por que o Codex não lista as tools do mycli.

O que a skill faz

1. Identifica o cliente, a versão instalada e o escopo (usuário ou projeto). Lê o arquivo de configuração existente antes de escrever.
2. Localiza o executável e roda `mycli --version`. Em desenvolvimento, roda `bun run src/main.ts --version` na raiz do projeto e anota o caminho absoluto do Bun e do entry point.
3. Roda `mycli mcp --list` e confere a presença de `skill`. Roda também `mycli doctor` para conferir a configuração e o acesso ao cofre. Avisos de stdin e stdout sem TTY são esperados nesse cenário.
4. Adiciona a entrada do cliente. No Claude Code, o formato básico é `claude mcp add mycli -- mycli mcp`. Nos demais, aplica o bloco JSON ou TOML do cliente com os caminhos reais.
5. Valida o JSON ou o TOML, revisa o diff e recarrega a conexão ou reinicia o cliente.
6. Roda `npx @modelcontextprotocol/inspector mycli mcp` (ou `npx @modelcontextprotocol/inspector bun run src/main.ts mcp` em desenvolvimento). Seleciona stdio, conecta, lista as tools e chama `skill` com `{"topic":"cli"}`.
7. No cliente configurado, envia o primeiro prompt de teste e confere no histórico que a tool `skill` foi chamada.

Exemplo

O `mycli` 0.7.0 está instalado em `/usr/local/bin/mycli`. Você quer a entrada no projeto, então a skill usa o escopo de projeto do Claude Code com o caminho absoluto:

```
claude mcp add --scope project mycli -- /usr/local/bin/mycli mcp
```

O comando grava esta entrada em `.mcp.json`, na raiz do projeto:

```
{
  "mcpServers": {
    "mycli": {
      "type": "stdio",
      "command": "/usr/local/bin/mycli",
      "args": ["mcp"]
    }
  }
}
```

Repare no que não está ali: nenhum `shop.token`, nenhuma master key, nenhum `env` com segredo. O servidor lê o token do cofre com `getCredential()`. Antes, você grava a credencial pela CLI, com o mesmo usuário que roda o cliente:

```
mycli credentials set shop.token
mycli credentials status
```

No Codex, a mesma entrada vai para `~/.codex/config.toml`:

```
[mcp_servers.mycli]
command = "/usr/local/bin/mycli"
args = ["mcp"]
```

Em desenvolvimento, sem compilar, o Cursor recebe o Bun e o entry point como caminhos absolutos em `.cursor/mcp.json`:

```
{
  "mcpServers": {
    "mycli": {
      "command": "/home/ana/.bun/bin/bun",
      "args": ["run", "/home/ana/mycli/src/main.ts", "mcp"]
    }
  }
}
```

Com a entrada carregada, a skill envia ao agente o prompt de teste:

> No servidor MCP mycli, chame a tool skill com topic cli. Leia o resultado e resuma como usar os commands e a saída JSON. Não altere arquivos nem execute operações de escrita.

A conexão está confirmada quando o histórico de tools mostra a chamada a `skill` com `{"topic":"cli"}` e o retorno em Markdown. Uma resposta em texto do agente, sem chamada de tool, não confirma nada.

Regras que a skill segue

- **Só o cliente e o escopo pedidos.** Entradas existentes, comentários e outras preferências do arquivo ficam intactos.
- **Caminho absoluto quando o `PATH` falha.** Aplicativos gráficos podem herdar um `PATH` diferente do shell. Command e arguments vão em campos separados, não numa string só como `bun run src/main.ts mcp`.
- **`mcp` como argument, sem `--json` nem `--list`.** No processo persistente, stdout pertence ao JSON-RPC. `mcp --list` imprime o catálogo e termina, então o cliente perderia o servidor.

- **Segredo fora da configuração.** Token, credencial e master key não entram em `args`, `env` nem no arquivo do cliente. Esses arquivos costumam ir para o Git. O cofre existe para guardar o token cifrado.
- **Mesmo usuário e mesma pasta de dados.** Um cliente com outro ambiente lê outro cofre. A correção é ajustar o ambiente de lançamento, sem copiar o cofre para o cliente.
- **Catálogo não é conexão.** `mcp --list` mostra o que está registrado, mas não negocia o protocolo. Por isso a skill faz uma chamada real pelo Inspector e pelo cliente.
- **Inspector é ferramenta externa.** O `npx` roda só no diagnóstico. Ele não acrescenta Node nem npm ao executável Termfy.
- **Uma entrada por nome e escopo.** Para alternar entre publicado e local, a skill substitui a entrada existente em vez de criar outra.
- **Diagnóstico por etapa.** Diante de falha, a skill separa inicialização do processo, negociação, descoberta de tools e execução. Ela não aumenta timeout antes de achar a etapa lenta nem troca stdio por HTTP.

O que ela entrega

No relatório final, a skill informa:

- o cliente e a versão;
- o arquivo e o escopo alterados, como `.mcp.json`, `~/.codex/config.toml`, `.cursor/mcp.json`, `opencode.json` ou `~/.gemini/settings.json`;
- o command e os arguments configurados, sem segredos;
- o resultado de `--version`, `mcp --list`, do Inspector e da primeira chamada no cliente.

Ela separa o que testou do que ficou como instrução pendente, por exemplo quando o cliente não está instalado na máquina. Nenhum arquivo do repositório do binário muda, só a configuração do cliente.

Próxima skill

Com o binário ligado aos agentes, o ciclo recomeça na próxima evolução: volte à termfy-spec para especificar o que vem depois.