**Constrói o imposter passo a passo — exercícios**

**Exercício 0 — Preparação inicial**

1. Objetivo: criar a estrutura base do imposter (porta e protocolo).
2. Tarefa: define a porta em que o imposter vai escutar e que o protocolo é HTTP.
3. Validação: confirma que o imposter pode ser criado no Mountebank (admin API) e que a porta está reservada.
4. Teste: depois de criar o imposter vazio, verifica que uma requisição GET a qualquer caminho retorna 404 (ou a resposta default do imposter).

**Exercício 1 — Stub: lista de marcas (GET /api/marcas)**

1. Objetivo: responder com uma lista de marcas quando for pedido o endpoint de marcas.
2. Predicado: método = GET, path = /api/marcas.
3. Resposta esperada:
   * Código HTTP: 200
   * Header: Content-Type = application/json
   * Corpo: JSON com um array chamado marcas contendo várias strings (ex.: Toyota, Audi, Ford, Volvo, Honda).
4. Testes:
   * Enviar um GET para /api/marcas.
   * Validar: status 200; header Content-Type correcto; corpo JSON contém a chave marcas e pelo menos 1 item.
   * Caso de erro: se o header estiver errado, regista e corrige.

**Exercício 2 — Stub: pesquisa de carros por marca (GET /api/cars?brand=Toyota)**

1. Objetivo: responder com resultados quando existir o parâmetro brand=Toyota.
2. Predicado: método = GET, path = /api/cars, querystring contém brand=Toyota.
3. Resposta esperada:
   * Código HTTP: 200
   * Header: Content-Type = application/json
   * Corpo: JSON com brand = "Toyota" e results contendo pelo menos um objecto com id, modelo, preco.
4. Testes:
   * Enviar GET para /api/cars com query brand=Toyota.
   * Validar: status 200; corpo JSON com brand igual ao parâmetro; results não vazio.
   * Teste negativo: enviar GET para /api/cars sem query (ver Exercício 3) — deve obter resposta diferente.

**Exercício 3 — Stub: GET /api/cars sem parâmetros**

1. Objetivo: responder com lista vazia quando não houver parâmetros específicos.
2. Predicado: método = GET, path = /api/cars (qualquer query que não encaixe no exercício 2).
3. Resposta esperada:
   * Código HTTP: 200
   * Header: application/json
   * Corpo: JSON com results = array vazio.
4. Testes:
   * Enviar GET para /api/cars sem query.
   * Validar: status 200; results é um array e encontra-se vazio.

**Exercício 4 — Stub: login bem-sucedido (POST /api/login com body exacto)**

1. Objetivo: aceitar apenas credenciais específicas e devolver token.
2. Predicados:
   * método = POST, path = /api/login
   * header Content-Type = application/json
   * corpo JSON deepEquals com {"username":"admin","password":"secret"} (estrutura exacta)
3. Resposta esperada:
   * Código HTTP: 200
   * Header: Content-Type = application/json
   * Corpo: JSON com status = "ok" e token = "validtoken"
4. Testes:
   * Enviar POST com corpo exacto (username=admin, password=secret).
   * Validar: status 200; existe token no corpo.
   * Teste de robustez: enviar as mesmas credenciais com ordem de campos diferente ou espaços — se usar deepEquals, pode falhar; documentar o comportamento.

**Exercício 5 — Stub: login com credenciais inválidas (POST /api/login genérico)**

1. Objetivo: tratar todos os outros pedidos de login como inválidos.
2. Predicados:
   * método = POST, path = /api/login
   * header Content-Type = application/json
   * aplica-se quando não corresponder ao predicado de sucesso (Exercício 4).
3. Resposta esperada:
   * Código HTTP: 401
   * Header: Content-Type = application/json
   * Corpo: JSON com error = "invalid\_credentials"
4. Testes:
   * Enviar POST com username diferente ou password errada.
   * Validar: status 401; error com o valor esperado.

**Exercício 6 — Stub: criar utilizador (email duplicado) (POST /api/users com email específico)**

1. Objetivo: simular conflito quando o email já existe.
2. Predicados:
   * método = POST, path = /api/users
   * header Content-Type = application/json
   * corpo contém a substring "email":"exists@example.com" (contains)
3. Resposta esperada:
   * Código HTTP: 409
   * Header: Content-Type = application/json
   * Corpo: JSON com error = "email\_exists" e message explicativa.
4. Testes:
   * Enviar POST com um JSON que contenha "email":"exists@example.com".
   * Validar: status 409; corpo com error e mensagem correta.
   * Teste alternativo: enviar outro email — deve cair no stub seguinte (Exercício 7).

**Exercício 7 — Stub: criar utilizador com sucesso (POST /api/users)**

1. Objetivo: criar utilizador quando não houver conflito de email.
2. Predicados:
   * método = POST, path = /api/users
   * header Content-Type = application/json
   * aplica-se quando o predicado de conflito não corresponder
3. Resposta esperada:
   * Código HTTP: 201
   * Headers: Content-Type = application/json; Location = /api/users/{id}
   * Corpo: JSON com id, status = "created" e mensagem de sucesso.
4. Testes:
   * Enviar POST com nome, email diferente de exists@example.com e idade.
   * Validar: status 201; header Location existe; corpo com id e status.

**Exercício 8 — Stub: temperatura — campo obrigatório (POST /api/temperature sem field)**

1. Objetivo: validar campo obrigatório value.
2. Predicados:
   * método = POST, path = /api/temperature
   * header Content-Type = application/json
   * corpo **não contém** a string "value" (negation of contains)
3. Resposta esperada:
   * Código HTTP: 400
   * Header: Content-Type = application/json
   * Corpo: JSON com error = "missing\_field" e mensagem indicando que value é obrigatório.
4. Testes:
   * Enviar POST sem o campo value.
   * Validar: status 400; mensagem de erro informativa.
   * Teste positivo: enviar com value — deve corresponder ao Exercício 9.

**Exercício 9 — Stub: temperatura — criação com field presente (POST /api/temperature com value)**

1. Objetivo: aceitar medições válidas.
2. Predicados:
   * método = POST, path = /api/temperature
   * header Content-Type = application/json
   * corpo contém a substring "value" (contains)
3. Resposta esperada:
   * Código HTTP: 201
   * Header: Content-Type = application/json
   * Corpo: JSON com status = "created" e mensagem de sucesso.
4. Testes:
   * Enviar POST com sensor\_id e value numérico.
   * Validar: status 201; corpo com status.
   * Teste adicional: enviar value inválido (string não numérica) e especificar o comportamento desejado.

**Exercício 10 — Stub: tarefas duplicadas (POST /api/tasks com title "duplicate")**

1. Objetivo: simular conflito de tarefa duplicada.
2. Predicados:
   * método = POST, path = /api/tasks
   * header Content-Type = application/json
   * corpo contém "title":"duplicate"
3. Resposta esperada:
   * Código HTTP: 409
   * Header: application/json
   * Corpo: JSON com error = "duplicate\_task"
4. Testes:
   * Enviar POST com title exatamente "duplicate".
   * Validar: status 409; mensagem de erro.
   * Teste alternativo: outro título → ver Exercício 11.

**Exercício 11 — Stub: criação de tarefa com sucesso (POST /api/tasks)**

1. Objetivo: criar tarefa normal.
2. Predicados:
   * método = POST, path = /api/tasks
   * header Content-Type = application/json
   * aplica-se quando não corresponder ao predicado de duplicado
3. Resposta esperada:
   * Código HTTP: 201
   * Headers: Content-Type = application/json; Location = /api/tasks/{id}
   * Corpo: JSON com id, status = "created"
4. Testes:
   * Enviar POST com title diferente de "duplicate".
   * Validar: status 201; Location presente; corpo com id.

**Exercício 12 — Stub: autenticação (POST /api/auth)**

1. Objetivo: responder com token apenas a credenciais válidas.
2. Predicados:
   * método = POST, path = /api/auth
   * header Content-Type = application/json
   * corpo deepEquals com {"username":"user1","password":"pass1"} para sucesso
   * predicado genérico para falha (quando não corresponder ao sucesso)
3. Resposta esperada:
   * Sucesso: status 200, corpo com token e expires\_in
   * Falha: status 401, corpo com error
4. Testes:
   * Enviar credenciais certas → verificar token e expires\_in.
   * Enviar credenciais erradas → verificar status 401.

**Exercício 13 — Stub: recurso protegido / perfil (GET /api/profile com Authorization)**

1. Objetivo: exigir header Authorization com token válido.
2. Predicados:
   * método = GET, path = /api/profile
   * header Authorization = "Bearer validtoken" (exact match)
   * predicado genérico para ausência ou token inválido
3. Resposta esperada:
   * Com token válido: 200, corpo JSON com perfil do utilizador (username, email, etc.)
   * Sem token ou inválido: 403, corpo com error = "forbidden"
4. Testes:
   * Fazer GET com header Authorization contendo o token obtido no Exercício 12 → verificar status 200 e conteúdo.
   * Fazer GET sem header ou com token inválido → verificar status 403.

**Exercício 14 — Stub genérico para POST com Content-Type obrigatório**

1. Objetivo: garantir que todos os POST sem header correto recebem erro 415.
2. Predicados:
   * método = POST (genérico)
   * aplica-se quando não existir um predicado anterior correspondente
3. Resposta esperada:
   * Código HTTP: 415
   * Corpo: JSON indicando que Content-Type application/json é obrigatório.
4. Testes:
   * Fazer um POST para qualquer endpoint sem header Content-Type = application/json → verificar 415.

**Exercício 15 — Stub genérico para GET (fallback 404)**

1. Objetivo: resposta padrão para GET não tratados.
2. Predicado:
   * método = GET (genérico)
   * aplica-se se nenhum stub anterior corresponder
3. Resposta esperada:
   * Código HTTP: 404
   * Corpo JSON com error = "not\_found"
4. Testes:
   * Fazer GET para um caminho não definido → verificar retorno 404.

**Validações finais do imposter completo**

1. Ordem dos stubs: importa. As regras mais específicas (deepEquals / contains) devem ser colocadas **antes** das mais genéricas (POST genérico, GET genérico). Confirma essa ordem no ficheiro final.
2. Validação estrutural: valida o JSON do imposter contra um validador JSON (para garantir sintaxe correcta).
3. Validação semântica:
   * Confirma que para cada endpoint existem testes positivos e negativos documentados.
   * Confirma que os headers Content-Type e Authorization são verificados conforme descrito.
4. Testes de integração:
   * Criar uma sequência de operações que simulam um utilizador real: login válido → obter token → aceder perfil → criar tarefa → criar utilizador → verificar duplicado.
   * Documentar o fluxo e capturar as respostas esperadas em cada passo.
5. Testes de robustez:
   * Enviar corpos com campos fora de ordem, com espaços adicionais, ou com tipagem diferente — verificar quais predicados falham (especialmente deepEquals).
   * Testar comportamento quando o servidor Mountebank está com latência simulada (se adicionares comportamento de wait).

**Matriz de Testes (resumo) — o que verificar em cada pedido**

Para cada cenário abaixo, valida:

* Código HTTP esperado
* Header Content-Type
* Presença/valor de campos no corpo JSON
* Header Location quando aplicável

1. GET /api/marcas → 200, marcas array não vazio
2. GET /api/cars?brand=Toyota → 200, brand="Toyota", results não vazio
3. GET /api/cars → 200, results = []
4. POST /api/login (admin/secret) → 200, token presente
5. POST /api/login (outros) → 401, error invalid\_credentials
6. POST /api/users (email exists@example.com) → 409, error email\_exists
7. POST /api/users (novo email) → 201, Location + id
8. POST /api/temperature (sem value) → 400, missing\_field
9. POST /api/temperature (com value) → 201, status created
10. POST /api/tasks (title duplicate) → 409, duplicate\_task
11. POST /api/tasks (nova tarefa) → 201, Location + id
12. POST /api/auth (user1/pass1) → 200, token
13. GET /api/profile (com token) → 200, perfil
14. GET /api/profile (sem token) → 403, forbidden
15. POST qualquer sem Content-Type → 415
16. GET qualquer não definido → 404

**Recomendações finais de trabalho**

1. Constrói o imposter **por etapas**. Depois de adicionar cada stub, recarrega no Mountebank e executa os testes correspondentes. Documenta cada passo com uma captura de ecrã.
2. Mantém os stubs específicos primeiro e os genéricos no fim.
3. Regista todas as respostas esperadas num ficheiro de teste (tabela), para facilitar a demonstração.
4. Se no futuro quiseres respostas dinâmicas (gerar IDs, eco do corpo, timestamps), adiciona *inject* — mas apenas em ambiente controlado e com a opção --allowInjection.