Automação Inteligente de Testes de API

Postman e Dynamic Test Runner

A automação de testes de API é uma prática essencial para garantir a qualidade e a robustez de sistemas modernos baseados em microserviços.

Tradicionalmente, o uso do Postman se restringia a testes manuais ou scripts simples. No entanto, este artigo apresenta uma arquitetura avançada que transforma o Postman em um Dynamic Test Runner, ou seja, um executor automatizado de múltiplos cenários de teste a partir de uma única requisição.

Essa abordagem reduz a duplicação, facilita a manutenção e prepara o terreno para integração com pipelines de CI/CD, elevando o nível da automação em projetos profissionais e escaláveis.,

Esta documentação apresenta uma arquitetura inovadora para automação de testes de API usando o Postman, transformando-o em um executor dinâmico de testes (Dynamic Test Runner).

Objetivos:

- Reduzir duplicação de requests estáticos
- Centralizar a lógica de validação em um utilitário (utils)
- Automatizar fluxos sequenciais sem intervenção manual
- Facilitar manutenção e escalabilidade

Casos de Uso Ideais:

- APIs com múltiplos cenários de validação
- Testes de regressão automatizados
- Validação de campos obrigatórios, formatos e regras de negócio
- Integração com CI/CD (Newman, Jenkins, GitHub Actions)

Estrutura da Arquitetura

1. Camada Base: Script da Collection (utils)

Centraliza funções compartilhadas no Pre-request Script da Collection.

```
javascript
const utils = {
    /**
     * Prepara os testes dinâmicos antes da execução
     * @param {object} pm - Postman object
     * @param {array} testCases - Lista de cenários de teste
     */
    prepareTests: function(pm, testCases) {
        pm.collectionVariables.set("test_cases", JSON.stringify(testCases));

        let currentIndex = 0;
        if (!pm.variables.get("current_test_case")) {
            pm.variables.set("current_test_case", JSON.stringify(testCases[0]));
        }

        // Atualiza dinamicamente o body da requisição
        if (pm.request.body?.mode === 'raw') {
            const body = JSON.parse(pm.request.body.raw);
            const currentCase = testCases[currentIndex];

            // Suporte para campos aninhados (ex: "user.address.street")
            if (currentCase.field.includes('.')) {
                const fields = currentCase.field.split('.');
                let temp = body;
                for (let i = 0; i < fields.length - 1; i++) {
                    temp = temp[fields[i]];
                }
                temp[fields[fields.length - 1]] = currentCase.value;
            } else {
                body[currentCase.field] = currentCase.value;
            }
```

```
            pm.request.body.raw = JSON.stringify(body);
        }
    },

    // Outras funções (sendRequest, finishedTests, etc.)
};
```

2. Camada de Execução: Request Script

Cada requisição define seus próprios testCases e validações.

Exemplo Completo: Request + Response

◆ Request (POST /api/users)

```
json
{
    "name": "{{name}}",
    "email": "{{email}}",
    "password": "{{password}}"
}
```

🩹 *Pre-request Script (Define os cenários)*

```
javascript
const testCases = [
    {
        name: "Should return 400 when email is missing",
        field: "email",       // Campo a ser alterado
        value: null,          // Valor de teste
        expectedStatusCode: 400,
        expectedMessage: "Email is required"
    },
    {
        name: "Should return 400 when password is too short",
        field: "password",
        value: "123",         // Senha inválida
        expectedStatusCode: 400,
        expectedMessage: "Password must be at least 8 characters"
    }
];
```

```
utils.prepareTests(pm, testCases);  // Prepara os testes
```

◆ Tests Script (Valida a resposta)

```
javascript
const currentTestCase = utils.getCurrentTest(pm);

// Valida status code
pm.test(`${currentTestCase.name} - Status Code`, () => {
    pm.response.to.have.status(currentTestCase.expectedStatusCode);
});

// Valida mensagem de erro
pm.test(`${currentTestCase.name} - Error Message`, () => {
    const responseJson = pm.response.json();
    pm.expect(responseJson.message).to.eql(currentTestCase.expectedMessage);
});

// Finaliza e prepara próximo teste
utils.finishedTests(pm, postman, "Next_Request_Name");
```

◆ Exemplo de Response (Erro 400)

```
json
{
    "status": "error",
    "message": "Email is required",
    "errors": [
        {
            "field": "email",
            "description": "Email is required"
        }
    ]
```

3. Camada de Orquestração: Controle de Fluxo

Gerencia a execução sequencial dos testes.

```
javascript
finishedTests: function(pm, postman, nextRequest) {
    // Registra teste como concluído
    const finished = JSON.parse(pm.variables.get("test_cases_finished") || "[]");
    finished.push(JSON.parse(pm.variables.get("current_test_case")).name);
    pm.variables.set("test_cases_finished", JSON.stringify(finished));

    // Busca testes pendentes
    const allTests = JSON.parse(pm.collectionVariables.get("test_cases"));
    const pending = allTests.filter(test => !finished.includes(test.name));

    if (pending.length > 0) {
        // Prepara próximo teste
        pm.variables.set("current_test_case", JSON.stringify(pending[0]));
        postman.setNextRequest(postman.request.name);  // Repete a request
    } else {
        // Limpa variáveis e encerra
        pm.variables.unset("current_test_case");
        pm.variables.unset("test_cases_finished");
        postman.setNextRequest(nextRequest || null);  // Próxima request ou fim
    }
}
```

Educa comenta,

A arquitetura proposta transforma o Postman em uma ferramenta de execução automatizada de testes baseada em cenários dinâmicos.

O uso de uma camada de utilitários centralizada permite reaproveitamento de lógica, enquanto o controle de fluxo sequencial garante testes robustos sem a necessidade de múltiplas requisições.

Principais Benefícios

- Redução de redundância e duplicação de código
- Facilidade de manutenção com centralização da lógica
- Execução autônoma sem intervenção humana
- Compatibilidade com CI/CD, como Newman, Jenkins e GitHub Actions

Próximos Passos para a Comunidade EducaCiência FastCode

- Adicionar logs detalhados para rastreamento e auditoria
- Integrar validação de schema (tv4, ajv)
- Gerar relatórios HTML customizados
- Expandir integração com plataformas de DevOps

Essa metodologia permite escalar testes de API com eficiência e inteligência, alinhando-se aos princípios de qualidade contínua e DevOps.

EducaCiência FastCode para a comunidade