UC:Teste de back-end

Bernardo, Cauã, Rômulo, Victor Hugo

Os conceitos do teste de codificação server-side:

Autenticação

Autenticação é o processo pelo qual o servidor confirma a identidade do usuário que está tentando acessar o sistema. Quando um usuário faz login, ele fornece credenciais — geralmente um e-mail e senha — e o servidor valida essas informações. Se estiverem corretas, o servidor gera um token, como um JWT (JSON Web Token), que funciona como uma “chave” para o usuário provar sua identidade nas próximas requisições. Esse token deve ser enviado junto às chamadas para rotas protegidas para que o servidor saiba quem está acessando os dados ou funções restritas. Autenticação é a base para garantir que o acesso a uma aplicação seja feito apenas por usuários válidos.

Autorização

Depois que a autenticação é feita, entra a autorização, que é diferente, pois está ligada ao controle de acesso. Autorização responde à pergunta: “Esse usuário autenticado tem permissão para fazer essa ação ou acessar esses dados?” Por exemplo, em um sistema de tarefas, um usuário só deve conseguir visualizar, editar ou deletar as suas próprias tarefas, e não as tarefas de outras pessoas. O servidor precisa validar essa permissão antes de permitir a operação. Isso é feito, geralmente, comparando o identificador do usuário no token com o dono do recurso solicitado, bloqueando o acesso caso não seja o mesmo.

API RESTful

APIs RESTful são um padrão para construir serviços web onde os recursos — como usuários, tarefas, produtos — são acessados via URLs específicas e métodos HTTP que indicam a ação a ser realizada. Por exemplo, o método GET é usado para buscar dados, POST para criar novos recursos, PUT para atualizar e DELETE para apagar. Essa arquitetura é stateless, o que significa que cada requisição ao servidor deve conter todas as informações necessárias para ser processada, sem depender de estados salvos no servidor. APIs RESTful são fáceis de entender, escaláveis e amplamente adotadas para comunicação entre front-end e back-end.

Manipulação de Banco de Dados (CRUD)

O servidor é responsável por interagir com o banco de dados para salvar, recuperar, atualizar e deletar dados — as operações conhecidas como CRUD (Create, Read, Update, Delete). Isso pode ser feito tanto em bancos relacionais, usando SQL para manipular tabelas e registros, quanto em bancos NoSQL, que armazenam documentos em formatos mais flexíveis. Por exemplo, para criar uma tarefa, o backend recebe os dados via API, valida, e insere essa informação no banco, garantindo que ela fique associada ao usuário correto. Saber trabalhar com queries, conexões e transações é essencial para a construção da lógica server-side.

Validação de Dados

Antes de qualquer dado ser aceito e salvo, é fundamental validar que ele está correto e completo. Validação previne erros e vulnerabilidades, garantindo que os dados estejam no formato esperado — por exemplo, que um campo de e-mail tenha o símbolo “@” ou que a senha tenha um tamanho mínimo. Caso os dados não estejam válidos, o servidor deve responder com uma mensagem clara explicando o problema, evitando que informações erradas corrompam o banco ou que ataques sejam facilitados. Isso melhora a confiabilidade e segurança da aplicação.

Segurança no Backend

Manter a segurança na camada server-side envolve diversas práticas essenciais. Primeiro, as senhas nunca devem ser armazenadas em texto simples, mas sim “hasheadas” com algoritmos como bcrypt, tornando quase impossível recuperá-las mesmo em caso de vazamento. Também é preciso proteger as rotas com autenticação, garantir que tokens de acesso sejam válidos e expirem, e evitar ataques comuns, como SQL Injection (inserção de comandos maliciosos em consultas), Cross-Site Scripting (XSS), entre outros. Além disso, o uso de HTTPS para criptografar a comunicação entre cliente e servidor é fundamental para proteger dados sensíveis em trânsito.

Tratamento de Erros

Nenhum sistema é perfeito e erros acontecem — pode ser uma requisição com dados faltando, um usuário tentando acessar algo que não pode, ou uma falha interna no servidor. Um backend bem feito sabe capturar essas situações e

responder com códigos HTTP adequados (400 para erro do cliente, 401 para falta de autorização, 404 para recurso não encontrado, 500 para erros internos), além de mensagens claras que orientem o cliente sem expor informações sensíveis. Esse tratamento evita que erros causem falhas graves, melhora a experiência do usuário e ajuda na manutenção do código.

Organização do Código e Arquitetura

A organização do código backend é crucial para facilitar a manutenção, escalabilidade e colaboração. É comum usar arquiteturas que separam as responsabilidades: rotas definem os caminhos da API, controladores implementam a lógica de negócio, modelos representam a estrutura dos dados e interagem com o banco, enquanto middlewares cuidam de funcionalidades transversais como autenticação e tratamento de erros. Esse isolamento torna o código mais limpo e modular, permitindo que cada parte evolua independentemente, e facilitando a identificação e correção de problemas.

Testes Automatizados

Para garantir que o sistema funcione conforme esperado, principalmente ao evoluir ou corrigir bugs, é importante escrever testes automatizados. Eles verificam, por exemplo, se os endpoints retornam os códigos certos, se as regras de negócio são respeitadas, e se o sistema responde corretamente a diferentes situações, como dados inválidos ou acessos não autorizados. Testes unitários focam em funções isoladas, testes de integração verificam a comunicação entre partes do sistema, e testes end-to-end simulam o uso completo da aplicação. Ferramentas como Jest, Mocha, Pytest e outras ajudam a implementar esses testes de forma prática.

Documentação

Por fim, a documentação serve para registrar como a API funciona, quais endpoints estão disponíveis, como se autenticar, exemplos de requisição e resposta, e instruções para rodar o projeto. Uma boa documentação é essencial para que outros desenvolvedores possam utilizar a API corretamente e também para que você ou sua equipe possam manter e evoluir o sistema sem dúvidas. Normalmente, essa documentação fica em um arquivo README.md ou em sistemas dedicados como Swagger.

Planejamento dos Testes: Etapas Fundamentais

Primeiro, é importante entender o **escopo do sistema** que será testado — quais funcionalidades a API oferece, quais regras de negócio precisam ser validadas e quais dados o sistema manipula. A partir disso, você define os objetivos dos testes, ou seja, o que precisa ser verificado para garantir a qualidade.

Depois, é preciso **identificar os tipos de testes que serão realizados**. Geralmente, para backend, dividimos em:

- **Testes unitários**, que focam em funções e métodos isolados, garantindo que cada parte do código funciona corretamente individualmente.

- **Testes de integração**, que verificam se os diferentes componentes da aplicação trabalham juntos, por exemplo, se o endpoint realmente salva os dados no banco.

- **Testes funcionais ou end-to-end**, que simulam o uso real do sistema, validando o fluxo completo — desde o login, passando pela criação de dados, até a exclusão.

Com isso claro, é hora de **elaborar os casos de teste**. Cada caso deve descrever uma situação específica a ser testada, incluindo:

- Dados de entrada (exemplo: corpo da requisição JSON com campos válidos ou inválidos)

- Passos para executar a ação (exemplo: enviar um POST para `/login` com credenciais corretas)

- Resultado esperado (exemplo: receber token JWT e status 200)

É importante considerar não só os cenários positivos (onde tudo funciona) mas também os negativos, como dados incorretos, tentativas de acesso não autorizado, ou erros de sistema.

A etapa seguinte é a **preparação do ambiente de testes**, que deve refletir o ambiente de produção o máximo possível, garantindo que os testes tenham relevância. Isso inclui ter um banco de dados de testes, configurações isoladas, e dados iniciais para serem usados durante os testes.

Depois disso, vem a **execução dos testes**, que pode ser manual — usando ferramentas como Postman ou Curl para enviar requisições e verificar respostas — ou automatizada, com scripts que rodem testes unitários e integrações regularmente, idealmente integrados a um pipeline de CI/CD.

Por fim, ao executar os testes, deve-se fazer a **análise dos resultados**. Todo erro ou comportamento inesperado deve ser registrado, investigado e corrigido. Depois das correções, os testes devem ser rodados novamente para garantir que o problema foi resolvido e que não surgiram novos bugs.

O planejamento dos testes não termina aí: ele deve ser atualizado conforme a aplicação evolui, adicionando novos casos para funcionalidades recém-implementadas ou alteradas.

Implantação dos teste

A implementação dos testes server-side começa identificando quais partes do código devem ser testadas, e para isso, normalmente dividimos em três níveis principais: testes unitários, testes de integração e testes funcionais ou end-to-end. Os testes unitários focam em pequenas funções ou métodos isolados da aplicação, verificando se eles retornam os resultados esperados com diferentes entradas. Por exemplo, uma função que valida se um e-mail tem o formato correto pode ser testada para confirmar que aceita strings válidas e rejeita as inválidas. Essa abordagem ajuda a garantir que a lógica básica da aplicação esteja correta antes de avançar para testes mais complexos.

Depois, temos os testes de integração, que verificam se diferentes partes do sistema interagem corretamente. No contexto de um backend, isso significa testar se, ao chamar um endpoint da API, a aplicação realiza todas as etapas necessárias, como validar os dados, salvar informações no banco de dados e retornar a resposta apropriada. Para isso, o ambiente de teste precisa estar preparado com uma base de dados própria, separada da produção, para evitar qualquer impacto real. Durante esses testes, simulam-se requisições HTTP para os endpoints da aplicação, conferindo se o status retornado, o corpo da resposta e os efeitos colaterais estão corretos. Por exemplo, testar se o endpoint de criação de usuários realmente salva o usuário e responde com os dados corretos, ou se um erro de validação resulta no status adequado.

Além disso, testes funcionais ou end-to-end simulam o uso real do sistema, verificando fluxos completos e garantindo que todas as partes do backend e, eventualmente, o frontend, estejam funcionando em conjunto. Esse tipo de teste valida desde o login até a execução de operações mais complexas, simulando a experiência do usuário final.

Para garantir que os testes sejam confiáveis e independentes uns dos outros, a implementação também deve considerar a preparação e limpeza do ambiente antes e depois de cada execução. Isso é feito com funções que configuram o banco de dados com dados necessários para o teste e limpam esses dados após o término, evitando que um teste interfira no resultado do outro.

Durante a implementação, ferramentas de testes ajudam a escrever e rodar esses testes de forma automatizada. No ambiente Node.js, por exemplo, utilizamos Jest para criar as funções de teste e Supertest para simular requisições HTTP ao servidor. O código dos testes geralmente fica separado do código da aplicação e é executado por comandos no terminal, que mostram se os testes passaram ou falharam, junto com detalhes sobre cada caso.

Por fim, a prática recomendada é integrar esses testes em um pipeline de integração contínua, onde toda vez que um novo código é enviado para o repositório, os testes são executados automaticamente para garantir que nada quebre sem ser percebido. Isso ajuda a manter a qualidade do software alta durante todo o ciclo de desenvolvimento.

A implementação dos testes server-side começa identificando quais partes do código devem ser testadas, e para isso, normalmente dividimos em três níveis principais: testes unitários, testes de integração e testes funcionais ou end-to-end. Os testes unitários focam em pequenas funções ou métodos isolados da aplicação, verificando se eles retornam os resultados esperados com diferentes entradas. Por exemplo, uma função que valida se um e-mail tem o formato correto pode ser testada para confirmar que aceita strings válidas e rejeita as inválidas. Essa abordagem ajuda a garantir que a lógica básica da aplicação esteja correta antes de avançar para testes mais complexos.

Depois, temos os testes de integração, que verificam se diferentes partes do sistema interagem corretamente. No contexto de um backend, isso significa testar se, ao chamar um endpoint da API, a aplicação realiza todas as etapas necessárias, como validar os dados, salvar informações no banco de dados e retornar a resposta apropriada. Para isso, o ambiente de teste precisa estar preparado com uma base de dados própria, separada da produção, para evitar qualquer impacto real. Durante esses testes, simulam-se requisições HTTP para os endpoints da aplicação, conferindo se o status retornado, o corpo da resposta e os efeitos colaterais estão corretos. Por exemplo, testar se o endpoint de criação de usuários realmente salva o usuário e responde com os dados corretos, ou se um erro de validação resulta no status adequado.

Além disso, testes funcionais ou end-to-end simulam o uso real do sistema, verificando fluxos completos e garantindo que todas as partes do backend e,

eventualmente, o frontend, estejam funcionando em conjunto. Esse tipo de teste valida desde o login até a execução de operações mais complexas, simulando a experiência do usuário final.

Execução dos testes server-side

A execução dos testes server-side acontece após escrever os scripts que verificam o funcionamento do backend, como testes unitários e de integração. Usando ferramentas como Jest ou Mocha, os testes são rodados via linha de comando, onde cada caso é executado e comparado com o resultado esperado. Quando o teste passa, significa que o código está correto naquele cenário; se falhar, o framework mostra mensagens que ajudam a identificar o problema.

Durante a execução, o ambiente de teste é preparado, com dados inseridos e limpos para garantir isolamento entre testes. Frequentemente, essa execução é automatizada em pipelines de integração contínua, que verificam a qualidade do código a cada alteração.

No fim, o desenvolvedor recebe um relatório indicando quais testes passaram ou falharam, permitindo corrigir erros e manter a qualidade do backend.

Análise dos resultados dos testes server-side

Primeiro, é preciso revisar quais testes passaram e quais falharam. Quando um teste passa, indica que a funcionalidade testada está funcionando conforme o esperado naquele cenário.

Já quando um teste falha, é necessário investigar a causa. Isso envolve ler as mensagens de erro fornecidas pelo framework de testes, que normalmente indicam qual teste não passou, qual resultado era esperado e qual foi o resultado obtido. Essas informações ajudam a identificar se o problema está no código, nos dados usados no teste ou até mesmo no próprio teste.

Em seguida, o desenvolvedor deve reproduzir o erro no ambiente de desenvolvimento para entender melhor o problema. Pode ser necessário revisar o código testado, corrigir bugs, ajustar regras de validação ou modificar o teste caso ele esteja errado.

Após corrigir o problema, os testes são executados novamente para confirmar que a falha foi resolvida e que não surgiram novos erros em outras partes do sistema — esse processo é chamado de regressão.

Por fim, os resultados e as correções devem ser documentados para que a equipe tenha um histórico das alterações feitas e possa acompanhar a evolução da qualidade do software. Essa análise contínua ajuda a manter o backend estável e confiável ao longo do desenvolvimento.