

**UNIVERSIDADE FEDERAL DE SERGIPE  
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA  
DEPARTAMENTO DE COMPUTAÇÃO**

**GUSTAVO HENRIQUE MARQUES  
HENRIQUE ROCHA VALENTIM BASTOS  
MARIANA SOUZA NUNES  
RAFAEL LAUTON SANTOS DE OLIVEIRA**

**ATIVIDADE 1 – TESTES UNITÁRIOS DISCUTIDOS NO STACK OVERFLOW**

**São Cristóvão - SE  
2025**

## SUMÁRIO

1. Introdução.....	3
2. Desenvolvimento.....	4
2.1. Qual é o problema?.....	4
2.2. Diferenças entre AVA e Jest.....	4
2.3. Aplicando a solução.....	5
3. Soluções Alternativas.....	9
3.1. Solução com Try/Catch.....	9
3.2. Validação Rigorosa com `expect.assertions()` .....	11
3.3. Testando Exceções em Funções Assíncronas.....	12
4. Conclusão.....	13
5. Links do GitHub e do vídeo.....	14
6. Referências.....	14


## 1. Introdução


A escolha da pergunta foi realizada com base nos critérios definidos pela atividade, utilizando a plataforma Stack Overflow como fonte. Primeiramente, acessamos o site e utilizamos o sistema de busca avançada para encontrar perguntas relacionadas especificamente ao tema de testes unitários. Aplicamos o filtro de ordenação por “Highest score” (maior pontuação) para garantir que a pergunta selecionada tivesse alta relevância e aprovação pela comunidade.

Dentre os resultados, analisamos as perguntas que atendiam aos critérios obrigatórios: conter pelo menos uma resposta marcada como aceita, possuir no mínimo 400 votos e estar claramente relacionada a um problema técnico envolvendo testes unitários. Após essa análise, escolhemos a pergunta "How to test the type of a thrown exception in Jest?", que aborda como verificar o tipo de erro lançado em uma função durante a execução de testes com o framework Jest.

---

591 votes

 [How to test the type of a thrown exception in Jest](#)

 23 answers


I'm working with some code where I need to **test** the type of an exception thrown by a function (is it `TypeError`, `ReferenceError`, etc.?). ... `error = t.throws() => { throwError(); }, TypeError); t.is(error.messag...`

811k views

javascript

unit-testing

jestjs

 [bartsmykla](#) 6,081 asked Sep 4, 2017 at 18:43

---

O Jest é um framework de testes em JavaScript amplamente utilizado para garantir a qualidade e o funcionamento correto de aplicações, especialmente aquelas desenvolvidas com Node.js, React e outras tecnologias modernas. Criado pelo time do Facebook, o Jest se destaca por sua simplicidade, desempenho e facilidade de configuração. Ele oferece recursos completos como testes unitários, mocks automáticos, medição de cobertura de código e integração com TypeScript. Com sintaxe intuitiva e suporte nativo a promessas e testes assíncronos, o Jest permite escrever testes claros e confiáveis, tornando-se uma ferramenta essencial no desenvolvimento orientado a testes (TDD).

A dúvida abordada é recorrente entre desenvolvedores que utilizam testes unitários modernos com JavaScript, tornando-a altamente relevante para estudo. A resposta aceita oferece uma solução clara e aplicável. Após a seleção, a equipe registrou a escolha da pergunta e os nomes dos membros na thread disponibilizada no Google Classroom, garantindo a validade da atividade conforme as diretrizes estabelecidas.

## 2. Desenvolvimento

### 2.1. Qual é o problema?

O desafio é como testar corretamente no Jest o tipo de exceção lançada por uma função, por exemplo verificar se ela lança `TypeError`, `ReferenceError` ou erros personalizados. O objetivo é não só saber se um erro ocorreu, mas garantir que o tipo específico do erro esteja correto. Uma abordagem incorreta comum é chamar direto a função dentro do `expect`, o que faz o erro ocorrer antes do Jest capturá-lo, ou usar `try/catch`, que pode passar o teste mesmo se nenhum erro for lançado.

### 2.2. Diferenças entre AVA e Jest

O autor do problema menciona utilizar o framework AVA para realizar testes. No AVA, a verificação de exceções lançadas usa o método `t.throws()`, que retorna diretamente o objeto de erro inconsistente com expectativas, permitindo que você execute assertivas adicionais sobre ele.

Já no Jest, o matcher `expect(...).toThrow()` não retorna o erro em si, ele apenas assegura que a função passada lança a exceção esperada. Nesse contexto, `matcher` refere-se a uma função ou método do Jest usado para fazer uma asserção (verificação) sobre o resultado de um teste.

Apesar de ambos serem frameworks voltados para testes em JavaScript/Typescript, há algumas diferenças notáveis entre estes. O Jest prioriza a simplicidade na configuração (*zero-config*), a compatibilidade com projetos React e uma experiência de desenvolvimento com alta produtividade.

Em contraste, o AVA segue uma proposta minimalista e funcional. Criado com foco em testes assíncronos modernos, AVA adota uma sintaxe concisa e evita o uso de globais, exigindo a importação explícita do método `test` para definição dos casos. Uma de suas principais características é a execução de cada arquivo de teste em um processo isolado, garantindo paralelismo real e evitando interferências entre testes. Embora não ofereça suporte nativo a `mocking` ou cobertura de código, AVA é extremamente eficaz em testes rápidos e modulares, sendo ideal para bibliotecas ou projetos Node.js que priorizam isolamento e desempenho.

Para testar o tipo do erro no Jest, pode-se passar a classe como argumento (`toThrow(CustomError)`); para conferir a mensagem, pode-se usar string ou uma

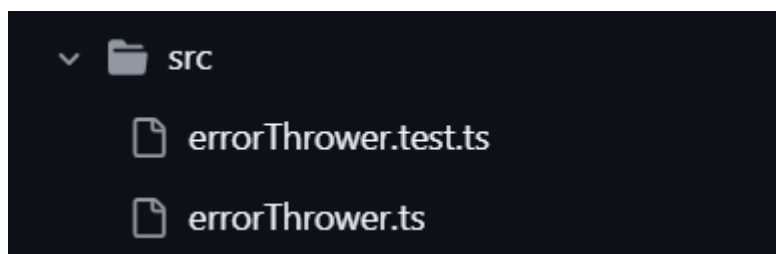
expressão regular (`toThrow('mensagem')`). No entanto, caso seja necessário inspecionar propriedades específicas do erro, precisa usar a abordagem com `try/catch` ou outras funcionalidades, como `expect.assertions()`, porque o `toThrow()` não disponibiliza o objeto de erro para inspeção.

### 2.3. Aplicando a solução

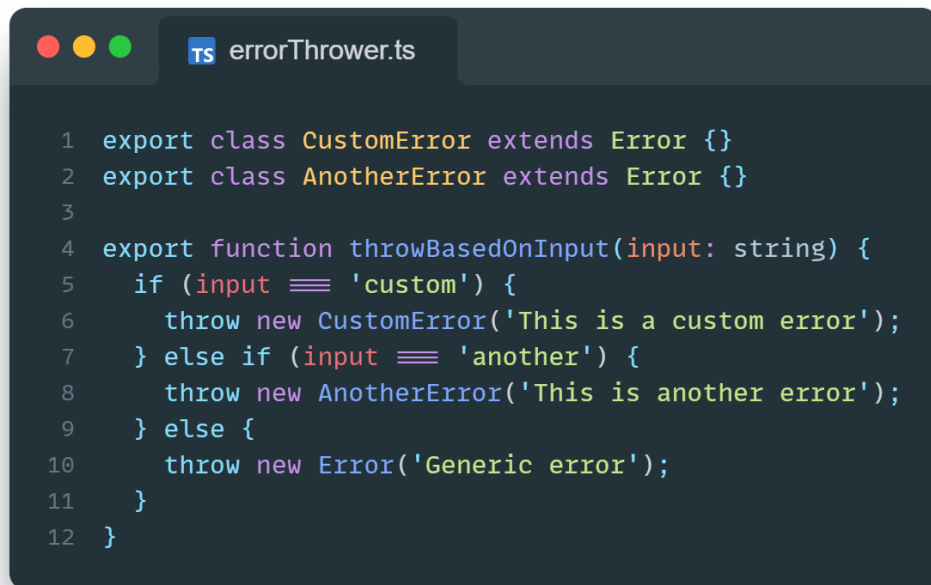
A solução aceita para testar o tipo de exceção lançada com Jest baseia-se no uso do matcher `toThrow` (também conhecido como `toThrowError`) de forma precisa, envolvendo a chamada da função em uma arrow function. Ao receber como argumento uma classe de erro, como `TypeError` ou um erro personalizado, o `toThrow` verifica internamente se o erro lançado é instância dessa classe. Isso resolve o problema de identificar o tipo correto da exceção lançada.

Para reproduzir o problema na IDE, iniciamos criando um projeto Node.js com suporte ao Jest, ferramenta escolhida para a escrita e execução dos testes. O primeiro passo consiste em inicializar o projeto com o comando `npm init -y`, que gera automaticamente o arquivo `package.json`. Em seguida, instalamos o Jest como dependência de desenvolvimento utilizando o comando `npm install jest --save-dev`. Essa instalação prepara o ambiente para que possamos escrever testes automatizados e executá-los facilmente com um simples comando no terminal.

Depois de configurar o ambiente, criamos dois arquivos principais: `errorThrower.ts` e `errorThrower.test.ts`.



No primeiro arquivo, implementamos a lógica da função `throwBasedOnInput`, que lança diferentes tipos de erro de acordo com o valor passado como argumento. Nessa etapa, também definimos duas classes de erro personalizadas, `CustomError` e `AnotherError`, herdando de `Error`, para serem usadas nos testes. O objetivo dessa função é justamente reproduzir um cenário em que diferentes tipos de exceções podem ser lançadas.



```
1 export class CustomError extends Error {}
2 export class AnotherError extends Error {}
3
4 export function throwBasedOnInput(input: string) {
5   if (input === 'custom') {
6     throw new CustomError('This is a custom error');
7   } else if (input === 'another') {
8     throw new AnotherError('This is another error');
9   } else {
10    throw new Error('Generic error');
11  }
12 }
```

Com a função implementada, passamos à criação dos testes no arquivo `errorThrower.test.ts`. No primeiro teste, o valor passado à função é a string `'custom'`. Espera-se que, nesse caso, a função lance uma exceção do tipo `CustomError` com a mensagem `'This is a custom error'`. Para isso, a chamada da função é encapsulada em uma constante `func`, que é então usada com o matcher `toThrow` do Jest, tanto para verificar o tipo quanto a mensagem da exceção lançada.

```

1 import { throwBasedOnInput, CustomError, AnotherError } from './errorThrower';
2
3 describe('throwBasedOnInput', () => {
4   test('should throw CustomError for "custom" input', () => {
5     const func = () => throwBasedOnInput('custom');
6
7     expect(func).toThrow(CustomError);
8
9     expect(func).toThrow('This is a custom error');
10  });

```

[snaplet.com](https://www.snaplet.com)

O segundo teste segue a mesma estrutura, mas utiliza a entrada 'another'. Espera-se que, com esse valor, a função lance uma exceção do tipo AnotherError, com a mensagem 'This is another error'. Novamente, é utilizada a constante func e os matchers toThrow para verificar se a exceção lançada corresponde ao comportamento esperado.

```

1 import { throwBasedOnInput, CustomError, AnotherError } from './errorThrower';
2
3 describe('throwBasedOnInput', () => {
4   test('should throw AnotherError for "another" input', () => {
5     const func = () => throwBasedOnInput('another');
6
7     expect(func).toThrow(AnotherError);
8     expect(func).toThrow('This is another error');
9   });

```

[snaplet.com](https://www.snaplet.com)

No terceiro teste, o valor passado para a função é 'unknown', uma entrada que não corresponde a nenhum dos casos específicos tratados pela função. Nesse

cenário, espera-se que seja lançada uma exceção genérica do tipo `Error`, com a mensagem 'Generic error'. Como o matcher `toThrow` não permite inspecionar o objeto erro diretamente, é usado um bloco `try/catch` para capturar a exceção. Dentro do `catch`, o erro é avaliado com os matchers `toBeInstanceOf` e `toHaveProperty`, assegurando que não seja uma instância de `CustomError` nem de `AnotherError`, mas sim do tipo `Error`, com a mensagem correta.



```
1 import { throwBasedOnInput, CustomError, AnotherError } from './errorThrower';
2
3 describe('throwBasedOnInput', () => {
4   test('should throw generic Error for unknown input', () => {
5     try {
6       throwBasedOnInput('unknown');
7     } catch (err) {
8       expect(err).toBeInstanceOf(Error);
9       expect(err).not.toBeInstanceOf(CustomError);
10      expect(err).not.toBeInstanceOf(AnotherError);
11      expect(err).toHaveProperty('message', 'Generic error');
12    }
13  });
14 });
```



### 3. Soluções Alternativas

#### 3.1. Solução com Try/Catch



It is a little bit weird, but it works and IMHO is good readable:

222



```
it('should throw Error with message \'UNKNOWN ERROR\' when no parameters were passed')
  try {
    throwError();
    // Fail test if above expression doesn't throw anything.
    expect(true).toBe(false);
  } catch (e) {
    expect(e.message).toBe("UNKNOWN ERROR");
  }
});
```

The `catch` block catches your exception, and then you can test on your raised `Error`. Strange `expect(true).toBe(false);` is needed to fail your test if the expected `Error` will be not thrown. Otherwise, this line is never reachable (`Error` should be raised before them).

@Kenny Body suggested a better solution which improve a code quality if you use

`expect.assertions()`:

```
it('should throw Error with message \'UNKNOWN ERROR\' when no parameters were passed')
  expect.assertions(1);
  try {
    throwError();
  } catch (e) {
    expect(e.message).toBe("UNKNOWN ERROR");
  }
});
```

Outra resposta apresentada na discussão do Stack Overflow propõe uma abordagem alternativa para verificar exceções lançadas em testes com Jest, utilizando blocos try/catch e, em versões mais recentes, wrappers auxiliares para contornar limitações de ferramentas como o ESLint. Essa solução visa permitir a inspeção direta do objeto de erro, prática útil quando se deseja verificar propriedades mais específicas da exceção, como um campo `statusCode` ou um `error.code`, além do tipo ou da mensagem padrão.

Na forma mais básica, a estrutura dessa alternativa consiste em envolver a chamada da função que deve lançar o erro dentro de um bloco try. Caso o erro não ocorra, uma asserção forçada como `expect(true).toBe(false)` é utilizada para garantir

a falha do teste. Dentro do catch, o erro capturado pode ser avaliado com qualquer combinação de matchers, como `toBeInstanceOf`, `toHaveProperty` ou `toEqual`. Posteriormente, uma versão mais refinada dessa abordagem foi sugerida, com o uso do método `expect.assertions()` para assegurar que ao menos uma asserção foi realizada, o que evita falsos positivos em casos onde o erro não é lançado e o catch nunca é executado.

## EDIT 2022:

To use this approach and not trigger `no-conditional-expect` rule (if you're using `eslint-plugin-jest`), documentation of this rule suggest to use error wrapper:

```
class NoErrorThrownError extends Error {}

const getError = async <TError>(call: () => unknown): Promise<TError> => {
  try {
    await call();

    throw new NoErrorThrownError();
  } catch (error: unknown) {
    return error as TError;
  }
};

describe('when the http request fails', () => {
  it('includes the status code in the error', async () => {
    const error = await getError(async () => makeRequest(url));

    // check that the returned error wasn't that no error was thrown
    expect(error).not.toBeInstanceOf(NoErrorThrownError);
    expect(error).toHaveProperty('statusCode', 404);
  });
});
```

See: [no-conditional-expect docs](#)

Além disso, uma técnica mais moderna foi apresentada utilizando uma função auxiliar chamada `getError()`, que encapsula a lógica de captura da exceção e evita violações às regras do `eslint-plugin-jest`, em especial a `no-conditional-expect`.

Apesar de funcional, essa abordagem alternativa não foi a solução aceita na discussão. A principal razão reside na complexidade desnecessária que ela introduz, especialmente diante da existência de uma solução nativa e mais idiomática do Jest.

### 3.2. Validação Rigorosa com `expect.assertions()`

[The answer by bodolsog](#) which suggests using a try/catch is close, but rather than expecting true to be false to ensure the expect assertions in the catch are hit, you can instead use `expect.assertions(2)` at the start of your test where `2` is the number of expected assertions. I feel this more accurately describes the intention of the test.

A full example of testing the type *and* message of an error:

```
describe('functionUnderTest', () => {
  it('should throw a specific type of error.', () => {
    expect.assertions(2);

    try {
      functionUnderTest();
    } catch (error) {
      expect(error).toBeInstanceOf(TypeError);
      expect(error).toHaveProperty('message', 'Something bad happened!');
    }
  });
});
```

If `functionUnderTest()` does *not* throw an error, the assertions will be hit, but the `expect.assertions(2)` will fail and the test will fail.

Além das soluções previamente analisadas, uma outra proposta relevante foi apresentada na discussão do Stack Overflow. Essa alternativa se baseia na combinação de um bloco try/catch com o uso explícito do método `expect.assertions(n)`, onde `n` representa a quantidade de asserções esperadas no teste. Tal prática impede a ocorrência de falsos positivos decorrentes de exceções inesperadas ou da ausência de erro.

O autor sugere encapsular a chamada à função em teste dentro de um bloco try, capturar o erro com catch e, nesse escopo, realizar duas asserções explícitas: uma para confirmar o tipo da exceção com `toBeInstanceOf()` e outra para verificar sua mensagem com `toHaveProperty('message', ...)`. O uso de `expect.assertions(2)` no início do teste tem um papel fundamental: ele assegura que ambas as asserções devem obrigatoriamente ser executadas para que o teste seja validado. Caso a função não lance erro algum, o catch não será acionado, nenhuma asserção será registrada e o teste falhará, evitando assim falsos positivos silenciosos.

Essa abordagem, embora mais verbosa do que o uso direto de `toThrow()`, destaca-se pela expressividade e controle sobre o objeto de exceção. Ela é particularmente útil em contextos mais sofisticados, como testes de APIs ou validações que envolvem erros personalizados com múltiplas propriedades. Portanto,

trata-se de uma alternativa metodologicamente sólida e que pode complementar o uso convencional dos matchers do Jest, agregando precisão e clareza na validação de comportamentos excepcionais.

### 3.3. Testando Exceções em Funções Assíncronas

I manage to combine some answers and end up with this:

```
it('should throw', async () => {  
  await expect(service.methodName('some@email.com', 'unknown')).rejects.toThrow(  
    HttpException,  
  );  
});
```

Essa estratégia adotada visa testar se um determinado método pertencente a um serviço, lança corretamente uma exceção do tipo `HttpException` ao receber parâmetros inválidos ou desconhecidos, utilizando encadeamento `await expect(...).rejects.toThrow(...)`. O teste, portanto, visa não apenas assegurar que uma exceção foi lançada, mas também garantir que essa exceção seja de um tipo específico, adequado ao comportamento esperado da aplicação.

Essa construção é particularmente apropriada para funções que operam de maneira assíncrona e que, em certos contextos, podem rejeitar com uma exceção. Ao utilizar o modificador `await`, o teste aguarda a conclusão da `Promise` retornada pela função. O uso da cadeia de matchers `expect(...).rejects.toThrow(...)` instrui o Jest a observar se a `Promise` é rejeitada com uma exceção, e se essa exceção corresponde ao tipo especificado, no caso `HttpException`.

Essa abordagem elimina a necessidade de um bloco explícito de `try/catch`, pois a captura da exceção é feita internamente pelo próprio Jest. Isso contribui para uma escrita mais limpa, legível e menos propensa a erros manuais, como esquecer de adicionar `expect.assertions()` ou não capturar corretamente o erro. Em segundo lugar, essa técnica é altamente expressiva: em uma única linha, o teste deixa claro o comportamento esperado.

Entretanto, há também limitações a serem consideradas. A principal delas reside na impossibilidade de inspecionar diretamente o conteúdo do objeto de erro lançado. Isto é, ao utilizar `toThrow` com `rejects`, pode-se verificar o tipo da exceção ou

a presença de uma determinada mensagem, mas não é possível acessar diretamente propriedades adicionais do erro, como o código de status HTTP, um identificador personalizado ou um campo de metadados. Para esses casos, a abordagem clássica com try/catch ainda é recomendada, pois oferece maior controle e detalhamento da exceção capturada.

## **4. Conclusão**

A análise das estratégias para testar o tipo de exceção lançada com o framework Jest revelou nuances importantes entre soluções sintaticamente elegantes e abordagens mais detalhadas, voltadas à inspeção aprofundada dos erros. A primeira solução apresentada foi escolhida como a mais adequada pela comunidade por combinar clareza, concisão e total alinhamento com a sintaxe e filosofia do Jest. Essa abordagem destaca-se por sua simplicidade e eficiência, permitindo a verificação direta do tipo de exceção com poucas linhas de código e alta legibilidade.

Por encapsular a função testada em uma arrow function e utilizar a classe do erro como argumento do matcher, ela elimina a necessidade de estruturas adicionais, reduzindo a complexidade e o risco de erros manuais. Outras soluções tendem a ser mais verbosas e suscetíveis a omissões, como esquecer `expect.assertions()` ou capturar incorretamente a exceção. Assim, a resposta aceita reflete um equilíbrio eficaz entre praticidade e cobertura funcional, sendo especialmente indicada para a maioria dos casos de uso em testes unitários.

Portanto, conclui-se que, ao escrever testes com Jest, é essencial compreender as características de cada abordagem, optando por soluções idiomáticas quando possível, mas recorrendo a técnicas alternativas quando o contexto exigir uma inspeção mais rigorosa da exceção.

## 5. Links do GitHub e do vídeo

[Link do GitHub](#)

[Link do vídeo](#)

## 6. Referências

JEST, IN. Stack Overflow. Disponível em: <<https://stackoverflow.com/questions/46042613>>. Acesso em: 16 jun. 2025.

RAFAEL-LAUTON-UFS. GitHub - raphael-lauton-ufs/Teste\_Software\_2025\_Gustavo\_Marques\_Henrique\_Bastos\_Mariana\_Nunes\_Rafael\_Oliveira. Disponível em: <[https://github.com/rafael-lauton-ufs/Teste\\_Software\\_2025\\_Gustavo\\_Marques\\_Henrique\\_Bastos\\_Mariana\\_Nunes\\_Rafael\\_Oliveira/tree/main](https://github.com/rafael-lauton-ufs/Teste_Software_2025_Gustavo_Marques_Henrique_Bastos_Mariana_Nunes_Rafael_Oliveira/tree/main)>. Acesso em: 16 jun. 2025.

JEST. Jest · Delightful JavaScript Testing. Disponível em: <<https://jestjs.io/>>.

Using Matchers · Jest. Disponível em: <<https://jestjs.io/docs/using-matchers>>. Acesso em: 16 jun. 2025.

AVAJS. ava/docs/01-writing-tests.md at main · avajs/ava. Disponível em: <<https://github.com/avajs/ava/blob/main/docs/01-writing-tests.md>>. Acesso em: 16 jun. 2025.

AVA vs Jest comparison of testing frameworks. Disponível em: <[https://knapsackpro.com/testing\\_frameworks/difference\\_between/ava/vs/jest](https://knapsackpro.com/testing_frameworks/difference_between/ava/vs/jest)>. Acesso em: 16 jun. 2025.

RAYGUN. JavaScript unit testing frameworks: Comparing Jasmine, Mocha, AVA, Tape and Jest. Disponível em: <<https://medium.com/@raygunio/javascript-unit-testing-frameworks-comparing-jasmine-mocha-ava-tape-and-jest-b989e95ed334>>. Acesso em: 16 jun. 2025.