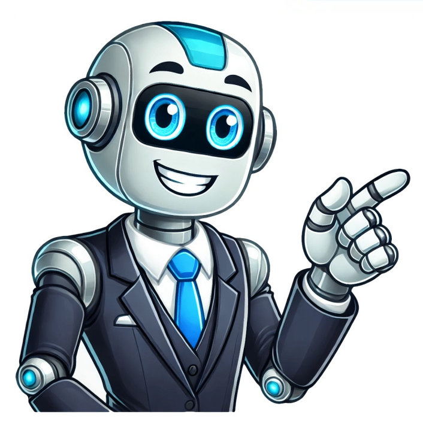
Continue



Vitest config exclude test

Hello there, I am trying to exclude the release.config.cjs and files like index.js which are sitting in my root folder of the project. I tried few different ways to add the entries, for example: export default defineConfig({ test: { exclude:[...configDefaults.exclude, '**/release.config.cjs', //or 'release.config.cjs' doesn't matter 'index.js' // or even **/index.*], coverage: { reporter: ['text', 'json', 'html'], }, }, }) Reproduction in terminal run npm run coverage both the release.config.js and src/basic.ts are counted in the coverage System Info System: OS: macOS 14.5 CPU: (10) arm64 Apple M1 Max Memory: 97.78 MB / 32.00 GB Shell: 5.9 - /bin/zsh Binaries: Node: 20.15.1 - ~/.volta/tools/image/node/20.15.1/bin/node Yarn: 1.22.19 - ~/.volta/tools/image/yarn/1.22.19/bin/yarn npm: 10.7.0 - ~/.volta/tools/image/node/20.15.1/bin/npm pnpm: 9.7.0 - ~/Library/pnpm/pnpm Browsers: Chrome: 127.0.6533.100 Safari: 17.5 npmPackages: @vitest/coverage-v8: ^2.0.5 => 2.0.5 vitest: ^2.0.5 => 2.0.5 Used Package Manager pnpm Validations If you are using Vite and have a vite.config file, Vitest will read it to match with the plugins and setup as your Vite app. If you want to have a different configuration for testing or your main app doesn't rely on Vite specifically, you could either:Create vitest.config.ts, which will have the higher priority and will override the configuration from vite.config.ts (Vitest supports all conventional JS and TS extensions, but doesn't support json) - it means all options in your vite.config will be ignoredPass --config option to CLI, e.g. vitest --config ./path/to/vitest.config.tsUse process.env.VITEST or mode property on defineConfig (will be set to test/benchmark if not overridden with --mode) to conditionally apply different configuration in vite.config.tsTo configure vitest itself, add test property in your Vite config. You'll also need to add a reference to Vitest types using a triple slash command at the top of your config file, if you are importing defineConfig from vite itself.Open Config ExamplesUsing defineConfig from vite you should follow this:ts/// import { defineConfig } from 'vite' export default defineConfig({ test: { // ... Specify options here. }, })The will stop working in Vitest 4, but you can already start migrating to vitest/config:ts/// import { defineConfig } from 'vite' export default defineConfig({ test: { // ... Specify options here. }, })Using defineConfig from vitest/config you should follow this:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { // ... Specify options here. }, })You can retrieve Vitest's default options to expand them if needed:tsimport { configDefaults, defineConfig } from 'vitest/config' export default defineConfig({ test: { exclude: [...configDefaults.exclude, 'packages/template/*'], }, })When using a separate vitest.config.js, you can also extend Vite's options from another config file if needed:tsimport { defineConfig, mergeConfig } from 'vitest/config' import viteConfig from './vite.config' export default mergeConfig(viteConfig, defineConfig({ test: { exclude: ['packages/template/*'], }, }))If your Vite config is defined as a function, you can define the config like this:tsimport { defineConfig, mergeConfig } from 'vitest/config' import viteConfig from './vite.config' export default defineConfig(configEnv => mergeConfig(viteConfig(configEnv), defineConfig({ test: { exclude: ['packages/template/*'], }, })))WARNINGAll listed options on this page are located within a test property inside the configuration:tsexport default defineConfig({ test: { exclude: [], }, })Since Vitest uses Vite config, you can also use any configuration option from Vite. For example, define to define global variables, or resolve.alias to define aliases - these options should be defined on the top level, not within a test property.Configuration options that are not supported inside a workspace project config have * sign next to them. This means they can only be set in the root Vitest config.include Type: string[]Default: ['**/*.{test,spec}.?(c|m)[jt]s?(x)']CLI: vitest [...include], vitest **/*.test.jsA list of glob patterns that match your test files.NOTEWhen using coverage, Vitest automatically adds test files include patterns to coverage's default exclude patterns. See coverage.exclude.exclude Type: string[]Default: ['**/node_modules/**', '**/dist/**', '**/cypress/**', '**/.{idea,git,cache,output,temp}/**', '**/{karma,rollup,webpack,vite,vitest,jest,ava,babel,nyc,cypress,tsup,build,eslint,prettier}.config.*']CLI: vitest --exclude "**/excluded-file"A list of glob patterns that should be excluded from your test files.WARNINGThis option does not affect coverage. If you need to remove certain files from the coverage report, use coverage.exclude.This is the only option that doesn't override your configuration if you provide it with a CLI flag. All glob patterns added via --exclude flag will be added to the config's exclude.includeSource Type: string[]Default: []Include globs for in-source test files.When defined, Vitest will run all matched files with import.meta.vitest inside.name Assign a custom name to the test project or Vitest process. The name will be visible in the CLI and available in the Node.js API via project.name.server Type: { sourcemap?, deps?, ... }Vite-Node server options.server.sourcemap Type: 'inline' | booleanDefault: 'inline'Inject inline source map to modules.server.debug Type: { dumpModules?, loadDumppedModules? }Vite-Node debugger options.server.debug.dumpModules Dump the transformed module to filesystem. Passing a string will dump to the specified path.server.debug.loadDumppedModules Read dumped module from filesystem whenever exists. Useful for debugging by modifying the dump result from the filesystem.server.deps Type: { external?, inline?, ... }Handling for dependencies resolution.server.deps.external Type: (string | RegExp)[]Default: [/\/node_modules\//]Externalize means that Vite will bypass the package to the native Node. Externalized dependencies will not be applied to Vite's transformers and resolvers, so they do not support HMR on reload. By default, all packages inside node_modules are externalized.These options support package names as they are written in node_modules or specified inside deps.moduleDirectories. For example, package @company/some-name located inside packages/some-name should be specified as some-name, and packages should be included in deps.moduleDirectories. Basically, Vitest always checks the file path, not the actual package name.If regexp is used, Vitest calls it on the file path, not the package name.server.deps.inline Type: (string | RegExp)[] | trueDefault: []Vite will process inlined modules. This could be helpful to handle packages that ship .js in ESM format (that Node can't handle).If true, every dependency will be inlined. All dependencies, specified in ssr.noExternal will be inlined by default.server.deps.fallbackCJS Type booleanDefault: falseWhen a dependency is a valid ESM package, try to guess the cjs version based on the path. This might be helpful, if a dependency has the wrong ESM file.This might potentially cause some misalignment if a package has different logic in ESM and CJS mode.server.deps.cacheDir Type stringDefault: 'node_modules/.vite'Directory to save cache files.deps Type: { optimizer?, ... }Handling for dependencies resolution.deps.optimizer Type: { ssr?, web? }See also: Dep Optimization OptionsEnable dependency optimization. If you have a lot of tests, this might improve their performance.When Vitest encounters the external library listed in include, it will be bundled into a single file using esbuild and imported as a whole module. This is good for several reasons:Importing packages with a lot of imports is expensive. By bundling them into one file we can save a lot of timeImporting UI libraries is expensive because they are not meant to run inside Node.jsYour alias configuration is now respected inside bundled packagesCode in your tests is running closer to how it's running in the browserBe aware that only packages in deps.optimizer?.[mode].include option are bundled (some plugins populate this automatically, like Svelte). You can read more about available options in Vite docs (Vitest doesn't support disable and noDiscovery options). By default, Vitest uses optimizer.web for jsdom and happy-dom environments, and optimizer.ssr for node and edge environments, but it is configurable by transformMode.This options also inherits your optimizeDeps configuration (for web Vitest will extend optimizeDeps, for ssr - ssr.optimizeDeps). If you redefine include/exclude option in deps.optimizer it will extend your optimizeDeps when running tests. Vitest automatically removes the same options from include, if they are listed in exclude.TIPYou will not be able to edit your node_modules code for debugging, since the code is actually located in your cacheDir or test.cache.dir directory. If you want to debug with console.log statements, edit it directly or force rebundling with deps.optimizer?.[mode].force option.deps.optimizer.{mode}.enabled Type: booleanDefault: falseEnable dependency optimization.deps.web Type: { transformAssets?, ... }Options that are applied to external files when transform mode is set to web. By default, jsdom and happy-dom use web mode, while node and edge environments use ssr transform mode, so these options will have no affect on files inside those environments.Usually, files inside node_modules are externalized, but these options also affect files in server.deps.external.deps.web.transformAssets Type: booleanDefault: trueShould Vitest process assets (.png, .svg, .jpg, etc) files and resolve them like Vite does in the browser.This module will have a default export equal to the path to the asset, if no query is specified.deps.web.transformCss Type: booleanDefault: trueShould Vitest process CSS (.css, .scss, .sass, etc) files and resolve them like Vite does in the browser.If CSS files are disabled with css options, this option will just silence ERR_UNKNOWN_FILE_EXTENSION errors.deps.web.transformGlobPattern Type: RegExp | RegExp[]Default: []Regexp pattern to match external files that should be transformed.By default, files inside node_modules are externalized and not transformed, unless it's CSS or an asset, and corresponding option is not disabled.deps.interopDefault Type: booleanDefault: trueInterpret CJS module's default as named exports. Some dependencies only bundle CJS modules and don't use named exports that Node.js can statically analyze when a package is imported using import syntax instead of require. When importing such dependencies in Node environment using named exports, you will see this error:import { read } from 'fs-jetpack'; ^^^^ SyntaxError: Named export 'read' not found. The requested module 'fs-jetpack' is a CommonJS module, which may not support all module.exports as named exports. CommonJS modules can always be imported via the default export.Vitest doesn't do static analysis, and cannot fail before your running code, so you will most likely see this error when running tests, if this feature is disabled:TypeError: createAsyncThunk is not a function TypeError: default is not a functionBy default, Vitest assumes you are using a bundler to bypass this and will not fail, but you can disable this behaviour manually, if you code is not processed.deps.moduleDirectories Type: string[]Default: ['node_modules']A list of directories that should be treated as module directories. This config option affects the behavior of vi.mock: when no factory is provided and the path of what you are mocking matches one of the moduleDirectories values, Vitest will try to resolve the mock by looking for a __mocks__ folder in the root of the project.This option will also affect if a file should be treated as a module when externalizing dependencies. By default, Vitest imports external modules with native Node.js bypassing Vite transformation step.Setting this option will override the default, if you wish to still search node_modules for packages include it along with any other options:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { deps: { moduleDirectories: ['node_modules', path.resolve('../../packages')], } }, })runner Type: VitestRunnerConstructorDefault: node, when running tests, or benchmark, when running benchmarksPath to a custom test runner. This is an advanced feature and should be used with custom library runners. You can read more about it in the documentation.benchmark Type: { include?, exclude?, ... }Options used when running vitest bench.benchmark.include Type: string[]Default: ['**/*.{bench,benchmark}.?(c|m)[jt]s?(x)']Include globs for benchmark test filesbenchmark.exclude Type: string[]Default: ['node_modules', 'dist', '.idea', '.git', '.cache']Exclude globs for benchmark test filesbenchmark.includeSource Type: string[]Default: []Include globs for in-source benchmark test files. This option is similar to includeSource.When defined, Vitest will run all matched files with import.meta.vitest inside.benchmark.reporters Type: ArrayableDefault: 'default'Custom reporter for output. Can contain one or more built-in report names, reporter instances, and/or paths to custom reporters.benchmark.outputFile Deprecated in favor of benchmark.outputJson.benchmark.outputJson Type: string | undefinedDefault: undefinedA file path to store the benchmark result, which can be used for --compare option later.For example:sh# save main branch's result git checkout main vitest bench --outputJson main.json # change a branch and compare against main git checkout feature vitest bench --compare main.jsonbenchmark.compare Type: string | undefinedDefault: undefinedA file path to a previous benchmark result to compare against current runs.alias Type: Record | ArrayDefine custom aliases when running inside tests. They will be merged with aliases from resolve.alias.WARNINGVitest uses Vite SSR primitives to run tests which has certain pitfalls.Aliases affect only modules imported directly with an import keyword by an inlined module (all source code is inlined by default).Vitest does not support aliasing require calls.If you are aliasing an external dependency (e.g., react -> preact), you may want to alias the actual node_modules packages instead to make it work for externalized dependencies. Both Yarn and pnpm support aliasing via the npm: prefix.globals Type: booleanDefault: falseCLI: --globals, --globals=falseBy default, vitest does not provide global APIs for explicitness. If you prefer to use the APIs globally like Jest, you can pass the --globals option to CLI or add globals: true in the config.tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { globals: true, }, })To get TypeScript working with the global APIs, add vitest/globals to the types field in your tsconfig.jsonjson{ "compilerOptions": { "types": ["vitest/globals"] } }If you are already using unplugin-auto-import in your project, you can also use it directly for auto importing those APIs.tsimport { defineConfig } from 'vitest/config' import AutoImport from 'unplugin-auto-import/vite' export default defineConfig({ plugins: [AutoImport({ imports: ['vitest'], dts: true, // generate TypeScript declaration }),], })environment Type: 'node' | 'jsdom' | 'happy-dom' | 'edge-runtime' | stringDefault: 'node'CLI: --environment=The environment that will be used for testing. The default environment in Vitest is a Node.js environment. If you are building a web application, you can use browser-like environment through either jsdom or happy-dom instead. If you are building edge functions, you can use edge-runtime environmentTIPYou can also use Browser Mode to run integration or unit tests in the browser without mocking the environment.By adding a @vitest-environment docblock or comment at the top of the file, you can specify another environment to be used for all tests in that file:Docblock style:js/** * @vitest-environment jsdom */ test('use jsdom in this test file', () => { const element = document.createElement('div') expect(element).not.toBeNull() })Comment style:js// @vitest-environment happy-dom test('use happy-dom in this test file', () => { const element = document.createElement('div') expect(element).not.toBeNull() })For compatibility with Jest, there is also a @jest-environment:js/** * @jest-environment jsdom */ test('use jsdom in this test file', () => { const element = document.createElement('div') expect(element).not.toBeNull() })If you are running Vitest with --isolate=false flag, your tests will be run in this order: node, jsdom, happy-dom, edge-runtime, custom environments. Meaning, that every test with the same environment is grouped, but is still running sequentially.Starting from 0.23.0, you can also define custom environment. When non-builtin environment is used, Vitest will try to load package vitest-environment-${name}. That package should export an object with the shape of Environment:tsimport type { Environment } from 'vitest' export default { name: 'custom', transformMode: 'ssr', setup() { // custom setup return { teardown() { // called after all tests with this env have been run } } } }Vitest also exposes builtinEnvironments through vitest/environments entry, in case you just want to extend it. You can read more about extending environments in our guide.TIPjsdom environment exposes jsdom global variable equal to the current JSDOM instance. If you want TypeScript to recognize it, you can add vitest/jsdom to your tsconfig.json when you use this environment:json{ "compilerOptions": { "types": ["vitest/jsdom"] } }environmentOptions Type: RecordDefault: {}These options are passed down to setup method of current environment. By default, you can configure only JSDOM options, if you are using it as your test environment.environmentMatchGlobs Type: [string, EnvironmentName][]Default: []DEPRECATEDThis API was deprecated in Vitest 3. Use workspace to define different configurations instead.tsexport default defineConfig({ test: { environmentMatchGlobs: [['./*.jsdom.test.ts', 'jsdom'],], workspace: [{ extends: true, test: { environment: 'jsdom', }, },], }, })Automatically assign environment based on globs. The first match will be used.For example:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { environmentMatchGlobs: [// all tests in tests/dom will run in jsdom ['tests/dom/**', 'jsdom'], // all tests in tests/ with .edge.test.ts will run in edge-runtime ['**\/*.edge.test.ts', 'edge-runtime'], // ...] } })poolMatchGlobs Type: [string, 'threads' | 'forks' | 'vmThreads' | 'vmForks' | 'typescript'][]Default: []DEPRECATEDThis API was deprecated in Vitest 3. Use workspace to define different configurations instead:tsexport default defineConfig({ test: { poolMatchGlobs: [['./*.threads.test.ts', 'threads'],], workspace: [{ test: { extends: true, pool: 'threads', }, },], }, })Automatically assign pool in which tests will run based on globs. The first match will be used.For example:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { poolMatchGlobs: [// all tests in "worker-specific" directory will run inside a worker as if you enabled `--pool=threads` for them, ['**/tests/worker-specific/**', 'threads'], // run all tests in "browser" directory in an actual browser ['**/tests/browser/**', 'browser'], // all other tests will run based on "browser.enabled" and "threads" options, if you didn't specify other globs // ...] } })Type: booleanDefault: falseCLI: -u, --update, --update=falseUpdate snapshot files. This will update all changed snapshots and delete obsolete ones.watch * Type: booleanDefault: !process.env.CI && process.stdin.isTTYCLI: -w, --watch, --watch=falseEnable watch modeIn interactive environments, this is the default, unless --run is specified explicitly.In CI, or when run from a non-interactive shell, "watch" mode is not the default, but can be enabled explicitly with this flag.root Type: stringCLI: -r , --root=Project rootdir Type: stringCLI: --dir=Default: same as rootBase directory to scan for the test files. You can specify this option to speed up test discovery if your root covers the whole projectreporters * Type: Reporter | Reporter[]Default: 'default'CLI: --reporter=, --reporter= --reporter=Custom reporters for output. Reporters can be a Reporter instance, a string to select built-in reporters, or a path to a custom implementation (e.g. './path/to/reporter.ts', '@scope/reporter').outputFile * Type: string | RecordCLI: --outputFile=, --outputFile.json=./pathWrite test results to a file when the --reporter=json, --reporter=html or --reporter=junit option is also specified. By providing an object instead of a string you can define individual outputs when using multiple reporters.pool * Type: 'threads' | 'forks' | 'vmThreads' | 'vmForks'Default: 'forks'CLI: --pool=threadsPool used to run tests in.threads * Enable multi-threading using tinypool (a lightweight fork of Piscina). When using threads you are unable to use process related APIs such as process.chdir(). Some libraries written in native languages, such as Prisma, bcrypt and canvas, have problems when running in multiple threads and run into segfaults. In these cases it is advised to use forks pool instead.forks * Similar as threads pool but uses child_process instead of worker_threads via tinypool. Communication between tests and main process is not as fast as with threads pool. Process related APIs such as process.chdir() are available in forks pool.vmThreads * Run tests using VM context (inside a sandboxed environment) in a threads pool.This makes tests run faster, but the VM module is unstable when running ESM code. Your tests will leak memory - to battle that, consider manually editing poolOptions.vmThreads.memoryLimit value.WARNINGRunning code in a sandbox has some advantages (faster tests), but also comes with a number of disadvantages.The globals within native modules, such as (fs, path, etc), differ from the globals present in your test environment. As a result, any error thrown by these native modules will reference a different Error constructor compared to the one used in your code:tstry { fs.writeFileSync('/doesnt exist') } catch (err) { console.log(err instanceof Error) // false }Importing ES modules caches them indefinitely which introduces memory leaks if you have a lot of contexts (test files). There is no API in Node.js that clears that cache.Accessing globals takes longer in a sandbox environment.Please, be aware of these issues when using this option. Vitest team cannot fix any of the issues on our side.vmForks * Similar as vmThreads pool but uses child_process instead of worker_threads via tinypool. Communication between tests and the main process is not as fast as with vmThreads pool. Process related APIs such as process.chdir() are available in vmForks pool. Please be aware that this pool has the same pitfalls listed in vmThreads.poolOptions * Type: RecordDefault: {}poolOptions.threads Options for threads pool.tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { poolOptions: { threads: { // Threads related options here } } } })poolOptions.threads.maxThreads * Type: number | stringDefault: available CPUsMaximum number or percentage of threads. You can also use VITEST_MAX_THREADS environment variable.poolOptions.threads.minThreads * Type: number | stringDefault: available CPUsMinimum number or percentage of threads. You can also use VITEST_MIN_THREADS environment variable.poolOptions.threads.singleThread Type: booleanDefault: falseRun all tests with the same environment inside a single worker thread. This will disable built-in module isolation (your source code or inlined code will still be reevaluated for each test), but can improve test performance.WARNINGEven though this option will force tests to run one after another, this option is different from Jest's --runInBand. Vitest uses workers not only for running tests in parallel, but also to provide isolation. By disabling this option, your tests will run sequentially, but in the same global context, so you must provide isolation yourself.This might cause all sorts of issues, if you are relying on global state (frontend frameworks usually do) or your code relies on environment to be defined separately for each test. But can be a speed boost for your tests (up to 3 times faster), that don't necessarily rely on global state or can easily bypass that.poolOptions.threads.useAtomics * Type: booleanDefault: falseUse Atomics to synchronize threads.This can improve performance in some cases, but might cause segfault in older Node versions.poolOptions.threads.isolate Type: booleanDefault: trueIsolate environment for each test file.poolOptions.threads.execArgv * Type: string[]Default: []Pass additional arguments to node in the threads. See Command-line API | Node.js for more information.poolOptions.forks Options for forks pool.tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { poolOptions: { forks: { // Forks related options here } } } })poolOptions.forks.maxForks * Type: number | stringDefault: available CPUsMaximum number or percentage of forks. You can also use VITEST_MAX_FORKS environment variable.Type: number | stringDefault: available CPUsMinimum number or percentage of forks. You can also use VITEST_MIN_FORKS environment variable.poolOptions.forks.isolate Type: booleanDefault: trueIsolate environment for each test file.poolOptions.forks.singleFork Type: booleanDefault: falseRun all tests with the same environment inside a single child process. This will disable built-in module isolation (your source code or inlined code will still be reevaluated for each test), but can improve test performance.WARNINGEven though this option will force tests to run one after another, this option is different from Jest's --runInBand. Vitest uses child processes not only for running tests in parallel, but also to provide isolation. By disabling this option, your tests will run sequentially, but in the same global context, so you must provide isolation yourself.This might cause all sorts of issues, if you are relying on global state (frontend frameworks usually do) or your code relies on environment to be defined separately for each test. But can be a speed boost for your tests (up to 3 times faster), that don't necessarily rely on global state or can easily bypass that.poolOptions.forks.execArgv * Type: string[]Default: []Pass additional arguments to node process in the child processes. See Command-line API | Node.js for more information.poolOptions.vmThreads Options for vmThreads pool.tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { poolOptions: { vmThreads: { // VM threads related options here } } } })poolOptions.vmThreads.maxThreads * Type: number | stringDefault: available CPUsMaximum number or percentage of threads. You can also use VITEST_MAX_THREADS environment variable.poolOptions.vmThreads.minThreads * Type: number | stringDefault: available CPUsMinimum number or percentage of threads. You can also use VITEST_MIN_THREADS environment variable.poolOptions.vmThreads.memoryLimit * Type: string | numberDefault: 1 / CPU CoresSpecifies the memory limit for workers before they are recycled. This value heavily depends on your environment, so it's better to specify it manually instead of relying on the default.TIPThe implementation is based on Jest's workerIdleMemoryLimit.The limit can be specified in a number of different ways and whatever the result is Math.floor is used to turn it into an integer value: 1 - Assumed to be a fixed byte value. Because of the previous rule if you wanted a value of 1 byte (I don't know why) you could use 1.1.With units 50% - As above, a percentage of total system memory100KB, 65MB, etc - With units to denote a fixed memory limit. K / KB - Kilobytes (x1000)KiB - Kibibytes (x1024)M / MB - MegabytesMiB - MebibytesG / GB - GigabytesGiB - GibibytespoolOptions.vmThreads.useAtomics * Type: booleanDefault: falseUse Atomics to synchronize threads.This can improve performance in some cases, but might cause segfault in older Node versions.poolOptions.vmThreads.execArgv * Type: string[]Default: []Pass additional arguments to node process in the VM context. See Command-line API | Node.js for more information.poolOptions.vmForks * Options for vmForks pool.tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { poolOptions: { vmForks: { // VM forks related options here } } } })poolOptions.vmForks.maxForks * Type: number | stringDefault: available CPUsMaximum number or percentage of forks. You can also use VITEST_MAX_FORKS environment variable.Type: number | stringDefault: available CPUsMinimum number or percentage of forks. You can also use VITEST_MIN_FORKS environment variable.poolOptions.vmForks.memoryLimit * Type: string | numberDefault: 1 / CPU CoresSpecifies the memory limit for workers before they are recycled. This value heavily depends on your environment, so it's better to specify it manually instead of relying on the default. How the value is calculated is described in poolOptions.vmThreads.memoryLimitpoolOptions.vmForks.execArgv * Type: string[]Default: []Pass additional arguments to node process in the VM context. See Command-line API | Node.js for more information.fileParallelism * Type: booleanDefault: trueCLI: --no-file-parallelism, --fileParallelism=falseShould all test files run in parallel. Setting this to false will override maxWorkers and minWorkers options to 1.TIPThis option doesn't affect tests running in the same file. If you want to run those in parallel, use concurrent option on describe or via a config.maxWorkers * Maximum number or percentage of workers to run tests in. poolOptions.{threads,vmThreads}.maxThreads/poolOptions.forks.maxForks has higher priority.minWorkers * Minimum number or percentage of workers to run tests in. poolOptions.{threads,vmThreads}.minThreads/poolOptions.forks.minForks has higher priority.testTimeout Type: numberDefault: 5_000 in Node.js, 15_000 if browser.enabled is trueCLI: --test-timeout=5000, --testTimeout=5000Default timeout of a test in milliseconds. Use 0 to disable timeout completely.hookTimeout Type: numberDefault: 10_000 in Node.js, 30_000 if browser.enabled is trueCLI: --hook-timeout=10000, --hookTimeout=10000Default timeout of a hook in milliseconds. Use 0 to disable timeout completely.teardownTimeout * Type: numberDefault: 10000CLI: --teardown-timeout=5000, --teardownTimeout=5000Default timeout to wait for close when Vitest shuts down, in millisecondssilent * Type: boolean | 'passed-only'Default: falseCLI: --silent, --silent=falseSilent console output from tests.Use 'passed-only' to see logs from failing tests only. Logs from failing tests are printed after a test has finished.setupFiles Path to setup files. They will be run before each test file.You can use process.env.VITEST_POOL_ID (integer-like string) inside to distinguish between threads.TIPNote, that if you are running --isolate=false, this setup file will be run in the same global scope multiple times. Meaning, that you are accessing the same global object before each test, so make sure you are not doing the same thing more than you need.For example, you may rely on a global variable:tsimport { config } from '@some-testing-lib' if (!globalThis.defined) { config.plugins = [myCoolPlugin] computeHeavyThing() globalThis.defined = true } // hooks are reset before each suite afterEach(() => { cleanup() }) globalThis.resetBeforeEachTest = trueprovide 2.1.0+ Type: PartialDefine values that can be accessed inside your tests using inject method.WARNINGProperties have to be strings and values need to be serializable because this object will be transferred between different processes.TIPIf you are using TypeScript, you will need to augment ProvidedContext type for type safe access:tsdeclare module 'vitest' { export interface ProvidedContext { API_KEY: string } } // mark this file as a module so augmentation works correctly export {}globalSetup Path to global setup files, relative to project root.A global setup file can either export named functions setup and teardown or a default function that returns a teardown function (example).Since Vitest 3, you can define a custom callback function to be called when Vitest reruns tests. If the function is asynchronous, the runner will wait for it to complete before executing tests. Note that you cannot destruct the project like { onTestsRerun } because it relies on the context.tsimport type { TestProject } from 'vitest/node' export default function setup(project: TestProject) { project.onTestsRerun(async () => { await restartDb() }) }forceRerunTriggers * Type: string[]Default: ['**/package.json/**', '**/vitest.config.*/**', '**/vite.config.*/**']Glob pattern of file paths that will trigger the whole suite rerun. When paired with the --changed argument will run the whole test suite if the trigger is found in the git diff.Useful if you are testing calling CLI commands, because Vite cannot construct a module graph:tstest('execute a script', async () => { // Vitest cannot rerun this test, if content of `dist/index.js` changes await execa('node', ['dist/index.js']) })TIPMake sure that your files are not excluded by server.watch.ignored.coverage * You can use v8, istanbul or a custom coverage solution for coverage collection.You can provide coverage options to CLI with dot notation:shnpx vitest --coverage.enabled --coverage.provider=istanbul --coverage.allWARNINGIf you are using coverage options with dot notation, don't forget to specify --coverage.enabled. Do not provide a single --coverage option in that case.coverage.provider Type: 'v8' | 'istanbul' | 'custom'Default: 'v8'CLI: --coverage.provider=Use provider to select the tool for coverage collection.coverage.enabled Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.enabled, --coverage.enabled=falseEnables coverage collection. Can be overridden using --coverage CLI option.coverage.include Type: string[]Default: ['**']Available for providers: 'v8' | 'istanbul'CLI: --coverage.include=, --coverage.include= --coverage.include=List of files included in coverage as glob patternscoverage.extension Type: string | string[]Default: ['.js', '.cjs', '.mjs', '.ts', '.mts', '.tsx', '.jsx', '.vue', '.svelte', '.marko', '.astro']Available for providers: 'v8' | 'istanbul'CLI: --coverage.extension=, --coverage.extension= --coverage.extension=coverage.exclude js['coverage/**', 'dist/**', '**/node_modules/**', '**/[.]**', 'packages/*/test?(s)/**', '**/*.d.ts', '**/virtual:*', '**/__x00__*', '**/\x00*', 'cypress/**', 'test?(s)/**', 'test?(-*).?(c|m)[jt]s?(x)', '**/*{.,-}{test,spec,bench,benchmark}?(-d).?(c|m)[jt]s?(x)', '**/__tests__/**', '**/{karma,rollup,webpack,vite,vitest,jest,ava,babel,nyc,cypress,tsup,build,eslint,prettier}.config.*', '**/vitest.{workspace,projects}.[jt]s?(on)', '**/.{eslint,mocha,prettier}rc.{?(c|m)js,yml}',]Available for providers: 'v8' | 'istanbul'CLI: --coverage.exclude=, --coverage.exclude= --coverage.exclude=List of files excluded from coverage as glob patterns.This option overrides all default options. Extend the default options when adding new patterns to ignore:tsimport { coverageConfigDefaults, defineConfig } from 'vitest/config' export default defineConfig({ test: { coverage: { exclude: ['**/custom-pattern/**', ...coverageConfigDefaults.exclude] }, }, })NOTEVitest automatically adds test files include patterns to the coverage.exclude. It's not possible to show coverage of test files.coverage.all Type: booleanDefault: trueAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.all, --coverage.all=falseWhether to include all files, including the untested ones into report.coverage.clean Type: booleanDefault: trueAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.clean, --coverage.clean=falseClean coverage results before running testscoverage.cleanOnRerun Type: booleanDefault: trueAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.cleanOnRerun, --coverage.cleanOnRerun=falseClean coverage report on watch rerun. Set to false to preserve coverage results from previous run in watch mode.coverage.reportsDirectory Type: stringDefault: './coverage'Available for providers: 'v8' | 'istanbul'CLI: --coverage.reportsDirectory=WARNINGVitest will delete this directory before running tests if coverage.clean is enabled (default value).Directory to write coverage report to.To preview the coverage report in the output of HTML reporter, this option must be set as a sub-directory of the html report directory (for example ./html/coverage).coverage.reporter Type: string | string[] | [string, {}][]Default: ['text', 'html', 'clover', 'json']Available for providers: 'v8' | 'istanbul'CLI: --coverage.reporter=, --coverage.reporter= --coverage.reporter=Coverage reporters to use. See istanbul documentation for detailed list of all reporters. See @types/istanbul-reporter for details about reporter specific options.The reporter has three different types:A single reporter: { reporter: 'html' }Multiple reporters without options: { reporter: ['html', 'json'] }A single or multiple reporters with reporter options: ts{ reporter: [['lcov', { 'projectRoot': './src' }], ['json', { 'file': 'coverage.json' }], ['text']] }You can also pass custom coverage reporters. See Guide - Custom Coverage Reporter for more information.ts { reporter: [// Specify reporter using name of the NPM package '@vitest/custom-coverage-reporter', ['@vitest/custom-coverage-reporter', { someOption: true }], // Specify reporter using local path '/absolute/path/to/custom-reporter.cjs', ['/absolute/path/to/custom-reporter.cjs', { someOption: true }],] }You can check your coverage report in Vitest UI: check Vitest UI Coverage for more details.coverage.reportOnFailure Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.reportOnFailure, --coverage.reportOnFailure=falseGenerate coverage report even when tests fail.coverage.allowExternal Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.allowExternal, --coverage.allowExternal=falseCollect coverage of files outside the project root.coverage.excludeAfterRemap 2.1.0+ Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.excludeAfterRemap, --coverage.excludeAfterRemap=falseApply exclusions again after coverage has been remapped to original sources. This is useful when your source files are transpiled and may contain source maps of non-source files.Use this option when you are seeing files that show up in report even if they match your coverage.exclude patterns.coverage.skipFull Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.skipFull, --coverage.skipFull=falseDo not show files with 100% statement, branch, and function coverage.coverage.thresholds Options for coverage thresholds.If a threshold is set to a positive number, it will be interpreted as the minimum percentage of coverage required. For example, setting the lines threshold to 90 means that 90% of lines must be covered.If a threshold is set to a negative number, it will be treated as the maximum number of uncovered items allowed. For example, setting the lines threshold to -10 means that no more than 10 lines may be uncovered.ts{ coverage: { thresholds: { // Requires 90% function coverage functions: 90, // Require that no more than 10 lines are uncovered lines: -10, } } }coverage.thresholds.lines Type: numberAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.lines=Global threshold for lines.coverage.thresholds.functions Type: numberAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.functions=Global threshold for functions.coverage.thresholds.branches Type: numberAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.branches=Global threshold for branches.coverage.thresholds.statements Type: numberAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.statements=Global threshold for statements.coverage.thresholds.perFile Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.perFile, --coverage.thresholds.perFile=falseCheck thresholds per file.Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.autoUpdate=Update all threshold values lines, functions, branches and statements to configuration file when current coverage is better than the configured thresholds. This option helps to maintain thresholds when coverage is improved.coverage.thresholds.100 Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'CLI: --coverage.thresholds.100, --coverage.thresholds.100=falseSets global thresholds to 100. Shortcut for --coverage.thresholds.lines 100 --coverage.thresholds.functions 100 --coverage.thresholds.branches 100 --coverage.thresholds.statements 100.coverage.thresholds[glob-pattern] Type: { statements?: number functions?: number branches?: number lines?: number }Default: undefinedAvailable for providers: 'v8' | 'istanbul'Sets thresholds for files matching the glob pattern.NOTEVitest counts all files, including those covered by glob-patterns, into the global coverage thresholds. This is different from Jest behavior.ts{ coverage: { thresholds: { // Thresholds for all files functions: 95, branches: 70, // Thresholds for matching glob pattern 'src/utils/**.ts': { statements: 95, functions: 90, branches: 85, lines: 80, }, // Files matching this pattern will only have lines thresholds set. // Global thresholds are not inherited. '**/math.ts': { lines: 100, } } } }coverage.thresholds[glob-pattern].100 2.1.0+ Type: booleanDefault: falseAvailable for providers: 'v8' | 'istanbul'Sets thresholds to 100 for files matching the glob pattern.ts{ coverage: { thresholds: { // Thresholds for all files functions: 95, branches: 70, // Thresholds for matching glob pattern 'src/utils/**.ts': { 100: true }, '**/math.ts': { 100: true } } } }coverage.ignoreEmptyLines Type: booleanDefault: true (false in v1)Available for providers: 'v8'CLI: --coverage.ignoreEmptyLines=Ignore empty lines, comments and other non-runtime code, e.g. Typescript types.This option works only if the used compiler removes comments and other non-runtime code from the transpiled code. By default Vite uses ESBuild which removes comments and Typescript types from .ts, .tsx and .jsx files.If you want to apply ESBuild to other files as well, define them in esbuild options:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ esbuild: { // Transpile all files with ESBuild to remove comments from code coverage. // Required for `test.coverage.ignoreEmptyLines` to work: include: ['**/*.js', '**/*.jsx', '**/*.mjs', '**/*.ts', '**/*.tsx'], }, test: { coverage: { provider: 'v8', ignoreEmptyLines: true, }, }, })coverage.ignoreClassMethods Type: string[]Default: []Available for providers: 'istanbul'CLI: --coverage.ignoreClassMethods=Set to array of class method names to ignore for coverage. See istanbul documentation for more information.coverage.watermarks ts{ statements?: [number, number], functions?: [number, number], branches?: [number, number], lines?: [number, number] }ts{ statements: [50, 80], functions: [50, 80], branches: [50, 80], lines: [50, 80] }Available for providers: 'v8' | 'istanbul'CLI: --coverage.watermarks.statements=50,80, --coverage.watermarks.branches=50,80Watermarks for statements, lines, branches and functions. See istanbul documentation for more information.coverage.processingConcurrency Type: booleanDefault: Math.min(20, os.availableParallelism?.() ?? os.cpus().length)Available for providers: 'v8' | 'istanbul'CLI: --coverage.processingConcurrency=Concurrency limit used when processing the coverage results.coverage.customProviderModule Type: stringAvailable for providers: 'custom'CLI: --coverage.customProviderModule=Specifies the module name or path for the custom coverage provider module. See Guide - Custom Coverage Provider for more information.testNamePattern * Type string | RegExpCLI: -t , --testNamePattern=, --test-name-pattern=Run tests with full names matching the pattern. If you add OnlyRunThis to this property, tests not containing the word OnlyRunThis in the test name will be skipped.jsimport { expect, test } from 'vitest' // run test('OnlyRunThis', () => { expect(true).toBe(true) }) // skipped test('doNotRun', () => { expect(true).toBe(true) })open * Type: booleanDefault: !process.env.CICLI: --open, --open=falseOpen Vitest UI (WIP)api Type: boolean | numberDefault: falseCLI: --api, --api.port, --api.host, --api.strictPortListen to port and serve API. When set to true, the default port is 51204browser experimental Default: { enabled: false }CLI: --browser=, --browser.name=chrome --browser.headlessConfiguration for running browser tests. Please, refer to the "Browser Config Reference" article.WARNINGThis is an experimental feature. Breaking changes might not follow SemVer, please pin Vitest's version when using it.Type: booleanDefault: falseWill call .mockClear() on all spies before each test. This will clear mock history without affecting mock implementations.mockReset Type: booleanDefault: falseWill call .mockReset() on all spies before each test. This will clear mock history and reset each implementation to its original.restoreMocks Type: booleanDefault: falseWill call .mockRestore() on all spies before each test. This will clear mock history, restore each implementation to its original, and restore original descriptors of spied-on objects..unstubEnvs Type: booleanDefault: falseWill call vi.unstubAllEnvs before each test.unstubGlobals Type: booleanDefault: falseWill call vi.unstubAllGlobals before each test.testTransformMode Determine the transform method for all modules imported inside a test that matches the glob pattern. By default, relies on the environment. For example, tests with JSDOM environment will process all files with ssr: false flag and tests with Node environment process all modules with ssr: true.testTransformMode.ssr Type: string[]Default: []Use SSR transform pipeline for all modules inside specified tests. Vite plugins will receive ssr: true flag when processing those files.testTransformMode.web Type: string[]Default: []First do a normal transform pipeline (targeting browser), then do a SSR rewrite to run the code in Node. Vite plugins will receive ssr: false flag when processing those files.snapshotFormat * Type: PrettyFormatOptionsFormat options for snapshot testing. These options are passed down to pretty-format.snapshotSerializers * Type: string[]Default: []A list of paths to snapshot serializer modules for snapshot testing, useful if you want add custom snapshot serializers. See Custom Serializer for more information.resolveSnapshotPath * Type: (testPath: string, snapExtension: string, context: { config: SerializedConfig }) => stringDefault: stores snapshot files in __snapshots__ directoryOverrides default snapshot path. For example, to store snapshots next to test files:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { resolveSnapshotPath: (testPath, snapExtension) => testPath + snapExtension, }, })allowOnly Type: booleanDefault: !process.env.CICLI: --allowOnly, --allowOnly=falseAllow tests and suites that are marked as only.dangerouslyIgnoreUnhandledErrors * Type: booleanDefault: falseCLI: --dangerouslyIgnoreUnhandledErrors --dangerouslyIgnoreUnhandledErrors=falseIgnore any unhandled errors that occur.passWithNoTests * Type: booleanDefault: falseCLI: --passWithNoTests, --passWithNoTests=falseVitest will not fail, if no tests will be found.logHeapUsage Type: booleanDefault: falseCLI: --logHeapUsage, --logHeapUsage=falseShow heap usage after each test. Useful for debugging memory leaks.css Type: boolean | { include?, exclude?, modules? }Configure if CSS should be processed. When excluded, CSS files will be replaced with empty strings to bypass the subsequent processing. CSS Modules will return a proxy to not affect runtime.css.include Type: RegExp | RegExp[]Default: []RegExp pattern for files that should return actual CSS and will be processed by Vite pipeline.TIPTo process all CSS files, use /.+/.css.exclude Type: RegExp | RegExp[]Default: []RegExp pattern for files that will return an empty CSS file.css.modules Type: { classNameStrategy? }Default: {}css.modules.classNameStrategy Type: 'stable' | 'scoped' | 'non-scoped'Default: 'stable'If you decide to process CSS files, you can configure if class names inside CSS modules should be scoped. You can choose one of the options:stable: class names will be generated as _${name}_${hashedFilename}, which means that generated class will stay the same, if CSS content is changed, but will change, if the name of the file is modified, or file is moved to another folder. This setting is useful, if you use snapshot feature.scoped: class names will be generated as usual, respecting css.modules.generateScopedName method, if you have one and CSS processing is enabled. By default, filename will be generated as _${name}_${hash}, where hash includes filename and content of the file.non-scoped: class names will not be hashed.WARNINGBy default, Vitest exports a proxy, bypassing CSS Modules processing. If you rely on CSS properties on your classes, you have to enable CSS processing using include option.maxConcurrency Type: numberDefault: 5CLI: --max-concurrency=10, --maxConcurrency=10A number of tests that are allowed to run at the same time marked with test.concurrent.Test above this limit will be queued to run when available slot appears.cache * Type: falseCLI: --no-cache, --cache=falseUse this option if you want to disable the cache feature. At the moment Vitest stores cache for test results to run the longer and failed tests first.The cache directory is controlled by the Vite's cacheDir option:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ cacheDir: 'custom-folder/.vitest' })You can limit the directory only for Vitest by using process.env.VITEST:tsimport { defineConfig } from 'vitest/config' export default defineConfig({ cacheDir: process.env.VITEST ? 'custom-folder/.vitest' : undefined })sequence Type: { sequencer?, shuffle?, seed?, hooks?, setupFiles? }Options for how tests should be sorted.You can provide sequence options to CLI with dot notation:shnpx vitest --sequence.shuffle --sequence.seed=1000sequence.sequencer * Type: TestSequencerConstructorDefault: BaseSequencerA custom class that defines methods for sharding and sorting. You can extend BaseSequencer from vitest/node, if you only need to redefine one of the sort and shard methods, but both should exist.Sharding is happening before sorting, and only if --shard option is provided.sequence.shuffle Type: boolean | { files?, tests? }Default: falseCLI: --sequence.shuffle, --sequence.shuffle=falseIf you want files and tests to run randomly, you can enable it with this option, or CLI argument --sequence.shuffle.Vitest usually uses cache to sort tests, so long running tests start earlier - this makes tests run faster. If your files and tests will run in random order you will lose this performance improvement, but it may be useful to track tests that accidentally depend on another run previously.sequence.shuffle.files Type: booleanDefault: falseCLI: --sequence.shuffle.files, --sequence.shuffle.files=falseWhether to randomize files, be aware that long running tests will not start earlier if you enable this option.sequence.shuffle.tests Type: booleanDefault: falseCLI: --sequence.shuffle.tests, --sequence.shuffle.tests=falseWhether to randomize tests.sequence.concurrent Type: booleanDefault: falseCLI: --sequence.concurrent, --sequence.concurrent=falseIf you want tests to run in parallel, you can enable it with this option, or CLI argument --sequence.concurrent.sequence.seed * Type: numberDefault: Date.now()CLI: --sequence.seed=1000Sets the randomization seed, if tests are running in random order.sequence.hooks Type: 'stack' | 'list' | 'parallel'Default: 'stack'CLI: --sequence.hooks=Changes the order in which hooks are executed.stack will order "after" hooks in reverse order, "before" hooks will run in the order they were definedlist will order all hooks in the order they are definedparallel will run hooks in a single group in parallel (hooks in parent suites will still run before the current suite's hooks)TIPThis option doesn't affect onTestFinished. It is always called in reverse order.sequence.setupFiles Type: 'list' | 'parallel'Default: 'parallel'CLI: --sequence.setupFiles=Changes the order in which setup files are executed.list will run setup files in the order they are definedparallel will run setup files in paralleltypecheck Options for configuring typechecking test environment.typecheck.enabled Type: booleanDefault: falseCLI: --typecheck, --typecheck.enabledEnable typechecking alongside your regular tests.typecheck.only Type: booleanDefault: falseCLI: --typecheck.onlyRun only typecheck tests, when typechecking is enabled. When using CLI, this option will automatically enable typechecking.typecheck.checker Type: 'tsc' | 'vue-tsc' | stringDefault: tscWhat tools to use for type checking. Vitest will spawn a process with certain parameters for easier parsing, depending on the type. Checker should implement the same output format as tsc.You need to have a package installed to use typechecker:tsc requires typescript packagevue-tsc requires vue-tsc packageYou can also pass down a path to custom binary or command name that produces the same output as tsc --noEmit --pretty false.typecheck.include Type: string[]Default: ['**/*.{test,spec}-d.?(c|m)[jt]s?(x)']Glob pattern for files that should be treated as test filestypecheck.exclude Type: string[]Default: ['**/node_modules/**', '**/dist/**', '**/cypress/**', '**/.{idea,git,cache,output,temp}/**']Glob pattern for files that should not be treated as test filestypecheck.allowJs Type: booleanDefault: falseCheck JS files that have @ts-check comment. If you have it enabled in tsconfig, this will not overwrite it.typecheck.ignoreSourceErrors Type: booleanDefault: falseDo not fail, if Vitest found errors outside the test files. This will not show you non-test errors at all.By default, if Vitest finds source error, it will fail test suite.typecheck.tsconfig Type: stringDefault: tries to find closest tsconfig.jsonPath to custom tsconfig, relative to the project root.slowTestThreshold * Type: numberDefault: 300CLI: --slow-test-threshold=, --slowTestThreshold=The number of milliseconds after which a test or suite is considered slow and reported as such in the results.chaiConfig Type: { includeStack?, showDiff?, truncateThreshold? }Default: { includeStack: false, showDiff: true, truncateThreshold: 40 }Equivalent to Chai config.chaiConfig.includeStack Type: booleanDefault: falseInfluences whether stack trace is included in Assertion error message. Default of false suppresses stack trace in the error message.chaiConfig.showDiff Type: booleanDefault: trueInfluences whether or not the showDiff flag should be included in the thrown AssertionErrors. false will always be false; true will be true when the assertion has requested a diff to be shown.chaiConfig.truncateThreshold Sets length threshold for actual and expected values in assertion errors. If this threshold is exceeded, for example for large data structures, the value is replaced with something like [Array(3)] or { Object (prop1, prop2) }. Set it to 0 if you want to disable truncating altogether.This config option affects truncating values in test.each titles and inside the assertion error message.bail Type: numberDefault: 0CLI: --bail=Stop test execution when given number of tests have failed.By default Vitest will run all of your test cases even if some of them fail. This may not be desired for CI builds where you are only interested in 100% successful builds and would like to stop test execution as early as possible when test failures occur. The bail option can be used to speed up CI runs by preventing it from running more tests when failures have occurred.retry Type: numberDefault: 0CLI: --retry=Retry the test specific number of times if it fails.onConsoleLog * Type: (log: string, type: 'stdout' | 'stderr') => boolean | voidCustom handler for console.log in tests. If you return false, Vitest will not print the log to the console.Can be useful for filtering out logs from third-party libraries.tsimport { defineConfig } from 'vitest/config' export default defineConfig({ test: { onConsoleLog(log: string, type: 'stdout' | 'stderr'): boolean | void { return !(log === 'message from third party library' && type === 'stdout') }, }, })onStackTrace * Type: (error: Error, frame: ParsedStack) => boolean | voidApply a filtering function to each frame of each stack trace when handling errors. The first argument, error, is an object with the same properties as a standard Error, but it is not an actual instance.Can be useful for filtering out stack trace frames from third-party libraries.tsimport type { ParsedStack } from 'vitest' import { defineConfig } from 'vitest/config' export default defineConfig({ test: { onStackTrace(error: Error, { file }: ParsedStack): boolean | void { // If we've encountered a ReferenceError, show the whole stack. if (error.name === 'ReferenceError') { return } // Reject all frames from third party libraries. if (file.includes('node_modules')) { return false } }, }, })diff Type: stringCLI: --diff=DiffOptions object or a path to a module which exports DiffOptions. Useful if you want to customize diff display.For example, as a config object:tsimport { defineConfig } from 'vitest/config' import c from 'picocolors' export default defineConfig({ test: { diff: { aIndicator: c.bold('--'), bIndicator: c.bold('++'), omitAnnotationLines: true, }, }, })Or as a module:diff.expand Type: booleanDefault: trueCLI: --diff.expand=falseExpand all common lines.diff.truncateThreshold Type: numberDefault: 0CLI: --diff.truncateThreshold=The maximum length of diff result to be displayed. Diffs above this threshold will be truncated. Truncation won't take effect with default value 0.diff.truncateAnnotation Type: stringDefault: '... Diff result is truncated'CLI: --diff.truncateAnnotation=Annotation that is output at the end of diff result if it's truncated.diff.truncateAnnotationColor Type: DiffOptionsColor = (arg: string) => stringDefault: noColor = (string: string): string => stringColor of truncate annotation, default is output with no color.diff.printBasicPrototype Type: booleanDefault: falsePrint basic prototype Object and Array in diff outputdiff.maxDepth Type: numberDefault: 20 (or 8 when comparing different types)Limit the depth to recurse when printing nested objectsfakeTimers Type: FakeTimerInstallOptsOptions that Vitest will pass down to @sinon/fake-timers when using vi.useFakeTimers().fakeTimers.now Type: number | DateDefault: Date.now()Installs fake timers with the specified Unix epoch.fakeTimers.toFake Type: ('setTimeout' | 'clearTimeout' | 'setImmediate' | 'clearImmediate' | 'setInterval' | 'clearInterval' | 'Date' | 'nextTick' | 'hrtime' | 'requestAnimationFrame' | 'cancelAnimationFrame' | 'requestIdleCallback' | 'cancelIdleCallback' | 'performance' | 'queueMicrotask')[]Default: everything available globally except nextTick and queueMicrotaskAn array with names of global methods and APIs to fake.To only mock setTimeout() and nextTick(), specify this property as ['setTimeout', 'nextTick'].Mocking nextTick is not supported when running Vitest inside node:child_process by using --pool=forks. NodeJS uses process.nextTick internally in node:child_process and hangs when it is mocked. Mocking nextTick is supported when running Vitest with --pool=threads.fakeTimers.loopLimit Type: numberDefault: 10_000The maximum number of timers that will be run when calling vi.runAllTimers().fakeTimers.shouldAdvanceTime Type: booleanDefault: falseTells @sinonjs/fake-timers to increment mocked time automatically based on the real system time shift (e.g. the mocked time will be incremented by 20ms for every 20ms change in the real system time).fakeTimers.advanceTimeDelta Relevant only when using with shouldAdvanceTime: true. increment mocked time by advanceTimeDelta ms every advanceTimeDelta ms change in the real system time.Type: booleanDefault: trueTells fake timers to clear "native" (i.e. not fake) timers by delegating to their respective handlers. When disabled, it can lead to potentially unexpected behavior if timers existed prior to starting fake timers session.workspace * Type: string | TestProjectConfigurationCLI: --workspace=./file.jsDefault: vitest.{workspace,projects}.{js,ts,json} close to the config file or rootPath to a workspace config file relative to root.Since Vitest 3, you can also define the workspace array in the root config. If the workspace is defined in the config manually, Vitest will ignore the vitest.workspace file in the root.isolate Type: booleanDefault: trueCLI: --no-isolate, --isolate=falseRun tests in an isolated environment. This option has no effect on vmThreads and vmForks pools.Disabling this option might improve performance if your code doesn't rely on side effects (which is usually true for projects with node environment).TIPYou can disable isolation for specific pools by using poolOptions property.includeTaskLocation Type: booleanDefault: falseShould location property be included when Vitest API receives tasks in reporters. If you have a lot of tests, this might cause a small performance regression.The location property has column and line values that correspond to the test or describe position in the original file.This option will be auto-enabled if you don't disable it explicitly, and you are running Vitest with:TIPThis option has no effect if you do not use custom code that relies on this.snapshotEnvironment Path to a custom snapshot environment implementation. This is useful if you are running your tests in an environment that doesn't support Node.js APIs. This option doesn't have any effect on a browser runner.This object should have the shape of SnapshotEnvironment and is used to resolve and read/write snapshot files:tsexport interface SnapshotEnvironment { getVersion: () => string getHeader: () => string resolvePath: (filepath: string) => Promise resolveRawPath: (testPath: string, rawPath: string) => Promise saveSnapshotFile: (filepath: string, snapshot: string) => Promise readSnapshotFile: (filepath: string) => Promise removeSnapshotFile: (filepath: string) => Promise }You can extend default VitestSnapshotEnvironment from vitest/snapshot entry point if you need to overwrite only a part of the API.WARNINGThis is a low-level option and should be used only for advanced cases where you don't have access to default Node.js APIs.If you just need to configure snapshots feature, use snapshotFormat or resolveSnapshotPath options.env Type: PartialEnvironment variables available on process.env and import.meta.env during tests. These variables will not be available in the main process (in globalSetup, for example).expect expect.requireAssertions Type: booleanDefault: falseThe same as calling expect.hasAssertions() at the start of every test. This makes sure that no test will pass accidentally.TIPThis only works with Vitest's expect. If you use assert or .should assertions, they will not count, and your test will fail due to the lack of expect assertions.You can change the value of this by calling vi.setConfig({ expect: { requireAssertions: false } }). The config will be applied to every subsequent expect call until the vi.resetConfig is called manually.expect.poll Global configuration options for expect.poll. These are the same options you can pass down to expect.poll(condition, options).expect.poll.interval Polling interval in millisecondsexpect.poll.timeout Type: numberDefault: 1000Polling timeout in millisecondsprintConsoleTrace Type: booleanDefault: falseAlways print console traces when calling any console method. This is useful for debugging. This tutorial explains how to set up the vitest.config.ts configuration file used in a Vite React (TypeScript) project. It effectively breaks down each configuration option and explains its purpose in simple terms.import { defineConfig, configDefaults } from 'vitest/config'export default defineConfig({ test: { globals: true, environment: 'jsdom', css: true, setupFiles: './vitest.setup.ts', exclude: [...configDefaults.exclude, '**/e2e/**'], // Example: Exclude e2e tests coverage: { provider: 'v8', // Use Vite's default coverage provider reporter: ['text', 'json', 'html'] }, }, })Let's explain the code step by step:import { defineConfig, configDefaults } from 'vitest/config'What it does:You're bringing in a function defineConfig to define your config in a cleaner way.configDefaults gives you Vitest's built-in default settings, so you can extend or tweak them.export default defineConfig({ test: { ... }, })What it does:You're exporting the configuration so Vitest can use it when it runs your tests.Inside the test object:globals: trueThis allows you to use describe, it, expect, etc., without importing them every time. It makes your test code cleaner and easier to read.environment: 'jsdom'Simulates a browser environment using jsdom (a fake browser in Node.js). It is necessary for testing things like the DOM (e.g., document, window) in React apps.css: trueAllows importing and using CSS inside your components during tests. Without this, Vitest might throw errors when it sees CSS imports.setupFiles: './vitest.setup.ts'Runs this file before any test starts. Great for setting up things like:- @testing-library/jest-dom- Mocking functions or global objectsexclude: [...configDefaults.exclude, '**/e2e/**']Skips certain files or folders when running tests. Here you are saying:"Use Vitest's default skip list, and also ignore anything inside an e2e folder."coverage: {...}This tells Vitest to track and report how much of your code is tested. Inside coverage:- provider: "v8" — Use the default Vite/Vitest coverage tool (based on the V8 JavaScript engine).- reporter: ["text","json","html"] — You'll get: Text output in your terminal JSON file for tooling An interactive HTML report to open in your browser.Summaryglobals: trueIt lets you use describe, expect, etc., without importsjsdomIt creates a fake browser for testing DOM-based code (like React)setupFilesPrepares your environment before testsexcludeSkips files/folders you don't want testedcoverageTells you which lines of your code are testedcss: trueAllows CSS in testable componentsLearn how to Set Up Testing in Your Vite React (Typescript) Project