Web Technologies and Full Stack Development

Node event loop and npm

The notes for the lesson “Node event loop and npm”: chapters 6.1 to 6.2, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 6.1 to 6.2

About these notes

These are the notes for the University free n easy video lesson **Node event loop and npm** on Web Technologies and Full Stack Development: chapters 6.1 to 6.2, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.1 Node.js architecture, event loop and modules

Node.js lets you run JavaScript outside the browser, on a server, using one thread for JavaScript and a helper thread pool behind it. The idea exists because waiting is slow: reading a file or calling a database can take milliseconds that one thread cannot afford to waste. So Node hands every slow job to the operating system and picks up the result later through the event loop. Every Express API, chat server and streaming service in this unit stands on this one mechanism.

> You need an earlier lesson's asynchronous JavaScript (callbacks, promises, async/await) and an earlier lesson's client-server and HTTP request/response picture; within this unit, 6.3 reuses the same file calls you see here.

V8 runs the code, libuv waits for the world

Node.js is a C++ program that embeds V8, the JavaScript engine from Chrome. V8 compiles your JavaScript into machine code and executes it. Node adds a second library, libuv, which supplies the event loop and a small thread pool.

Your JavaScript runs on one thread. A thread is the sequence of instructions the CPU is currently executing. Node keeps one **call stack**: the list of functions that are running right now. Only the

function on top of the stack executes. When the running function calls another function, the new one is pushed on top. When it returns, it is popped and the function below resumes.

Slow work does not sit on that stack. For `fs.readFile`, `dns.lookup` (the usual lookup behind HTTP requests), some crypto calls and timers, Node passes the job to libuv or to the operating system, attaches your callback, and returns at once. The stack is free for the next request. `dns.lookup` uses the libuv pool; `dns.resolve` uses the operating system resolver.

When the job finishes, libuv puts the callback into a queue. The event loop checks that queue and pushes callbacks back onto the stack, but only when the stack is empty. That is the whole architecture: one stack, one queue, one loop.

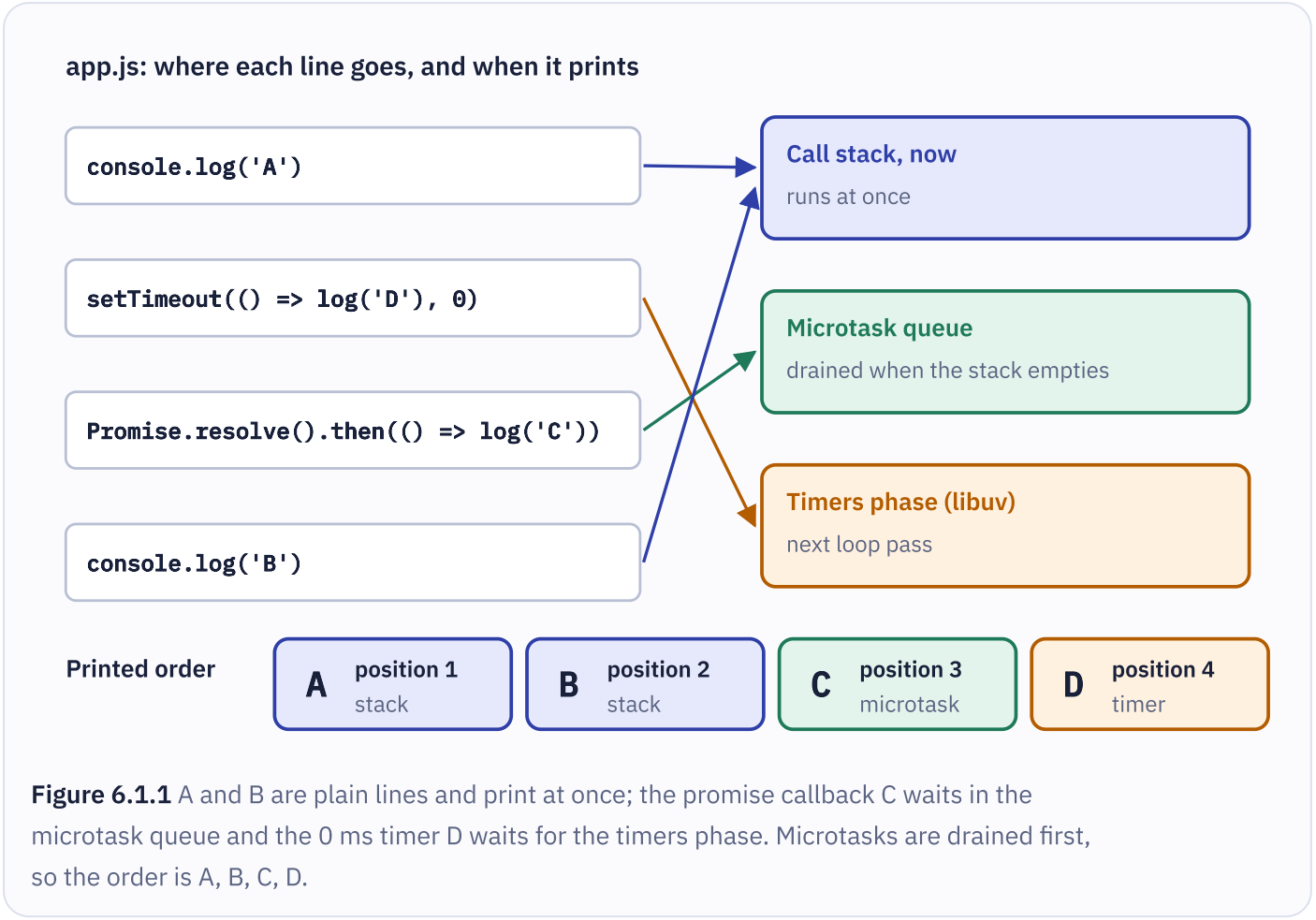

Figure 6.1.1 A and B are plain lines and print at once; the promise callback C waits in the microtask queue and the 0 ms timer D waits for the timers phase. Microtasks are drained first, so the order is A, B, C, D.

The event loop: phases and microtasks

The event loop repeats a fixed order of phases. Each phase has its own queue of callbacks.

1. **Timers**: run setTimeout and setInterval callbacks whose delay has expired.
2. **Pending callbacks**: run deferred system callbacks, such as a failed TCP connection.
3. **Poll**: wait for I/O and run its callbacks (a file read finished, socket data arrived).
4. **Check**: run setImmediate callbacks.

5. **Close**: run close handlers, such as socket.on('close').
6. After the close phase, the loop starts again at timers.

Two queues cut in between the phases: the process.nextTick queue and the promise microtask queue. After every single callback, and before the loop moves to the next phase, Node empties both. NextTick callbacks run first, then promise microtasks; the process.nextTick queue is fully drained before the promise queue is touched. So a resolved promise's `.then` always runs before any timer, even a 0 ms timer.

A blocking call keeps the thread busy until it finishes. `fs.readFileSync`, `crypto.pbkdf2Sync` and a long for loop are blocking. While they run, no other callback can start, so every user of your server waits. One 200 ms synchronous read on a busy evening makes every request behind it 200 ms late.

To watch this live, run `node --inspect app.js` and open chrome://inspect in Chrome DevTools to set breakpoints on the running stack. Some syllabi call this the Node Inspector.

Modules: core, local and third-party

A module is one file with its own scope. Nothing in it is visible to other files unless the file exports it. Node has three kinds.

- **Core modules** ship inside Node: `fs`, `path`, `os`, `http`, `events`. You load them by bare name, `const fs = require('fs')`, with nothing to install.
- **Local modules** are your own files. In math.js write `module.exports = { add, PI }`; in app.js write `const m = require('./math')`. `require` returns whatever `module.exports` holds. Some syllabi call a local file an export module.
- **Third-party modules** are packages installed into a node_modules folder, covered in 6.2.

require() resolves a string in a fixed order:

1. If the string starts with `./`, `../` or `/`, it is a file path: check for that file, then that file with .js, .json, .node.
2. Otherwise the core module list is checked first, so `'fs'` always gives the built-in.
3. Otherwise node_modules folders are searched, starting beside the file and walking up to the disk root.
4. The result is cached by resolved path, so a second require of the same file returns the cached exports and does not run the file again.

Modern Node also supports ES modules: write `import { add } from './math.js'` and `export const PI = 3.14159`. Set `"type": "module"` in package.json, or name the file

.mjs, and require() stops being available in that file.

WORKED EXAMPLE

Question. A file app.js contains the four lines below. Give the exact order in which A, B, C and D print, and give each letter its position number 1 to 4. Justify each position. console.log('A'); setTimeout(() => console.log('D'), 0); Promise.resolve().then(() => console.log('C')); console.log('B');

```
console.log('A');
setTimeout(() => console.log('D'), 0);
Promise.resolve().then(() => console.log('C'));
console.log('B');
```

1. Node starts app.js by pushing the module function on the call stack. Order so far: empty.
2. Line 1, console.log('A'), is plain synchronous code. It runs at once: A prints, position 1.
3. Line 2 calls setTimeout with delay 0 ms. Node registers a timer inside libuv and returns. Nothing prints. The callback is a macrotask, so it is not run now; it waits for the timers phase of the next loop pass.
4. Line 3, Promise.resolve().then(...), creates an already-resolved promise. Its callback is placed in the microtask queue, not run. Nothing prints.
5. Line 4, console.log('B'), is synchronous: B prints, position 2.
6. The stack is now empty. Node drains the microtask queue before entering any phase of the loop, so the promise callback runs: C prints, position 3.
7. The loop enters the timers phase. The 0 ms timer has expired, so its callback runs: D prints, position 4. The delay being 0 ms only means 'as soon as timers are checked', which is after all microtasks.

Dry run

STEP	WAITING QUEUES AFTER STEP	PRINTED SO FAR
console.log('A')	empty	A
setTimeout(..., 0)	timer: D	A
.then(...)	microtask: C, timer: D	A
console.log('B')	microtask: C, timer: D	A B
drain microtasks	timer: D	A B C
timers phase	empty	A B C D

Output

```
A
B
C
D
```

Answer. Output order is A, B, C, D at positions 1, 2, 3, 4: two synchronous lines first, then the promise microtask C, then the 0 ms timer D.

EXAM TRAPS

- Writing that setTimeout(fn, 0) runs before Promise.then(). Microtasks always run before the loop enters the next phase, so the promise callback comes first.
- Saying Node uses one thread for everything. Your JavaScript runs on one thread, but file, dns.lookup and crypto work runs on the libuv pool, 4 threads by default; dns.resolve uses the operating system resolver.
- Confusing require('fs') with require('./fs'). The first is the built-in file system module; the second loads your own file named fs.js.
- Forgetting the module cache: the second require of the same resolved path returns cached exports and does not re-run the file. A counter that never increases is usually cached.
- Passing a call instead of a function: setTimeout(fn(), 0) calls fn immediately on the stack and passes its return value to the timer.

CHECK YOURSELF

– REMEMBER **Name the two libraries inside Node.js and state one job of each.** 2 marks

V8 compiles and runs the JavaScript. libuv provides the event loop and the thread pool that carries file, DNS and crypto work.

– REMEMBER **List the five phases of the event loop in order.** 2 marks

Timers, pending callbacks, poll, check, close callbacks. Then the loop starts again at timers.

– UNDERSTAND **Why does the callback of Promise.resolve().then(f) run before a setTimeout(f2, 0) callback, even though f2 has zero delay?** 3 marks

The promise callback goes into the microtask queue. Node empties the microtask queue after every callback and before entering any phase of the loop. NextTick callbacks run first, then promise microtasks; the process.nextTick queue is fully drained before the promise queue is touched. The timer callback can only run in the timers phase, which comes later. So the promise callback always wins.

– APPLY **Predict the output order for: console.log(1); setTimeout(() => console.log(2), 0); Promise.resolve().then(() => console.log(3)); console.log(4);** 4 marks

1 and 4 are synchronous, so they print first in that order. The promise callback is a microtask, so 3 comes next. The timer runs in the timers phase, so 2 is last. Output: 1, 4, 3, 2.

– APPLY **A handler calls fs.readFileSync on each request, and each read takes 50 ms. 40 requests arrive together. What is the total time before the last response, and how long does the last user wait?** 4 marks

Synchronous reads run one after another on the single thread: 40 × 50 ms = 2000 ms = 2 s. The last user waits the full 2 s. With fs.readFile the reads are spread over the libuv pool of 4 threads: about 10 batches of 50 ms, so around 500 ms in total, and each user waits between roughly 50 ms and 500 ms instead of 2 s.

– ANALYZE **A file app.js contains require('./config') and another file lib/app.js contains require('config'). Explain what each loads and why.** 5 marks

require('./config') starts with './', so it is treated as a path and loads config.js from the current folder. require('config') has no path prefix, so Node checks the core module list first (there is no core config), then searches node_modules folders upward and loads the installed package named config. Same word, two different files.

– UNDERSTAND **What does module.exports do, and what does require() return?** 3 marks

module.exports is the object a file offers to the outside world; anything not attached to it stays private to the file. require() loads the file once, caches it, and returns exactly that module.exports object.

BEYOND THE SYLLABUS

Not for the exam. When one JavaScript thread is not enough, worker_threads starts real extra threads that can run CPU-heavy code such as image resizing without freezing the loop, and each worker has its own event loop. For several CPU cores serving traffic, cluster or a process manager forks one Node process per core behind a shared port. Newer runtimes such as Deno and Bun keep the same event loop idea while adding TypeScript and a built-in package manager. Search for worker_threads, the cluster module, AsyncLocalStorage, and the libuv thread pool size (UV_THREADPOOL_SIZE) when you meet them at work.

6.2 npm and package management

npm is two things at once: a public registry of JavaScript packages and the command-line tool that fetches them. Your project keeps a wish list in `package.json`, written as version ranges. The installer resolves those ranges against the registry, downloads code into `node_modules`, and records the exact versions it chose in `package-lock.json`. That split lets a team share one small manifest while every machine installs a repeatable set of files. You use npm in every Node project, in Express APIs, and in CI pipelines that must build the same app twice.

You need Chapter 6.1 for how `require()` searches `node_modules`, and ES modules (an earlier lesson).

package.json is the project manifest

`npm init -y` writes a starting `package.json` with default answers. The file is JSON, so keys and string values need double quotes. It records what the project is and what it needs.

- `name` and `version` identify the package. Version follows semantic versioning.
- `main` names the entry file that `require()` loads when another file imports this package. An app often uses `index.js` or `server.js`.
- `scripts` stores short commands you run with `npm run`.
- `dependencies` lists packages the running app needs, such as Express.
- `devDependencies` lists tools needed only for development or testing, such as nodemon.

Install a runtime package with `npm install express`. Add `-D` for a development tool: `npm install -D nodemon`. npm edits `package.json` and downloads code. A local install puts packages inside the project's `node_modules` folder. A global install with `-g` puts a command

on your machine for all projects. Keep libraries local; use global only for command-line tools. Put `start` in scripts as `node server.js`, then run `npm start`. Other scripts run as `npm run dev`.

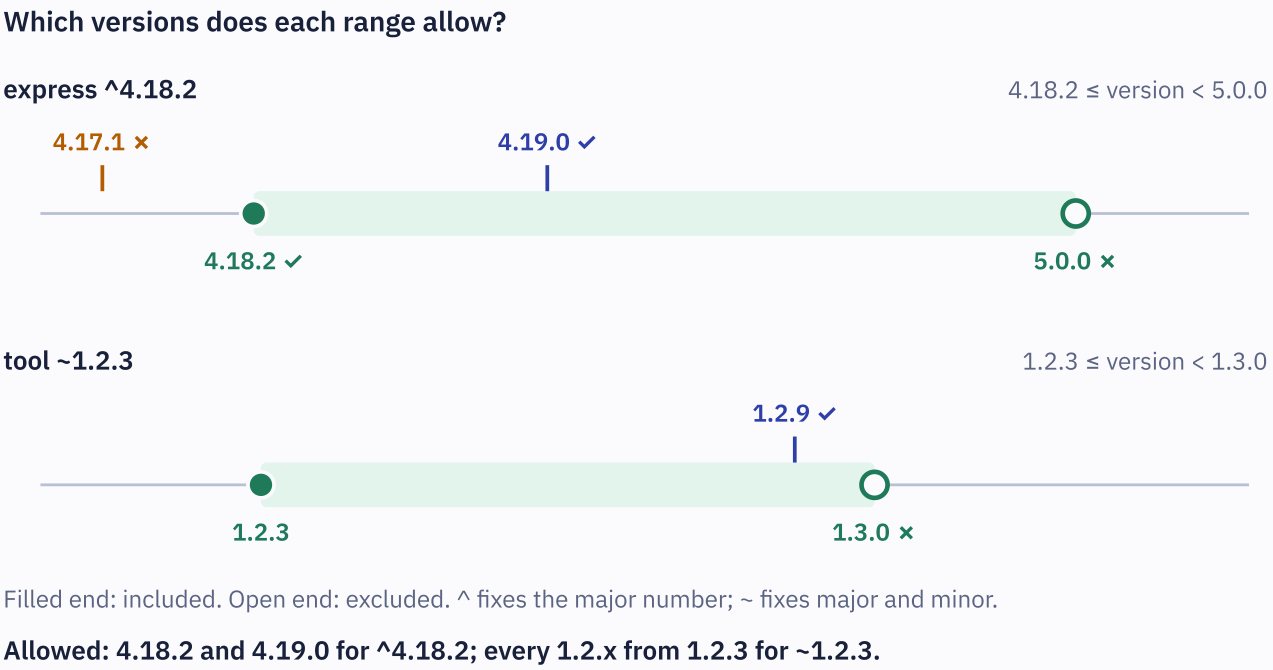

Figure 6.2.1 The caret range ^4.18.2 runs from 4.18.2 up to but not including 5.0.0, so 4.18.2 and 4.19.0 are allowed. The tilde range ~1.2.3 runs from 1.2.3 up to but not including 1.3.0, so 1.2.9 is allowed (and so is 1.2.3 itself, the filled end), while 1.3.0 is not.

Semantic versioning and version ranges

Semantic versioning writes a version as MAJOR.MINOR.PATCH. MAJOR changes when old code may break. MINOR adds features without breaking old code. PATCH fixes bugs without changing behaviour. From 4.18.2 to 4.19.0 is a minor update; 4.18.2 to 4.18.3 is a patch; 4.18.2 to 5.0.0 is a major change.

A range lets npm choose a newer version that should still work. Two ranges appear in exams.

- `^4.18.2` means at least 4.18.2 and below 5.0.0. The caret locks the major number.
- `~1.2.3` means at least 1.2.3 and below 1.3.0. The tilde locks the minor number.
- An exact version such as `4.18.2` allows only that version.

Write a dependency as `express` set to `^4.18.2` to accept future minor and patch releases. Use `~` when you want only patch releases. Use an exact version when even a minor update is risky. npm reads these ranges, asks the registry for matching versions, and chooses one set for the project.

Lock file, installs and scripts

`package-lock.json` records the exact version of every installed package, plus the versions of their dependencies. `package.json` says `^4.18.2`; the lock file may say `4.18.2` today. Commit the lock file. Then every developer and every server installs the same tree.

- `node_modules` is generated. Never edit it by hand. Usually do not commit it; it is large and can be rebuilt from the manifest and lock file.
- `npm install` reads `package.json`, may update versions inside the allowed ranges, and writes or updates the lock file.
- `npm ci` deletes `node_modules`, installs exactly the lock file, and fails if `package.json` and the lock file disagree. CI pipelines use it for repeatable builds. See Chapter 6.9.
- `npx` runs a package command without a permanent global install, for example `npx nodemon server.js`.

Run your own scripts with `npm run script-name`. `npm start` and `npm test` are shortcuts for the start and test scripts. Some syllabi group npm with task runners such as Grunt and Gulp; those are front-end build tools covered with the front-end unit. For a project, you need the commands above, not the whole npm CLI.

WORKED EXAMPLE

Question. A project has `express` set to `^4.18.2` and `tool` set to `~1.2.3`. For Express, test 4.17.1, 4.18.2, 4.19.0 and 5.0.0. For the tool, test 1.2.9 and 1.3.0. How many versions are allowed for each range?

1. Write the caret range ^4.18.2 as 4.18.2 ≤ version < 5.0.0.
2. Test 4.17.1: it is below 4.18.2, so reject it.
3. Test 4.18.2: it equals the lower bound, so accept it.
4. Test 4.19.0: it is above 4.18.2 and below 5.0.0, so accept it.
5. Test 5.0.0: it equals the upper bound, which is excluded, so reject it.
6. Write the tilde range ~1.2.3 as 1.2.3 ≤ version < 1.3.0.
7. Test 1.2.9: it is between the bounds, so accept it.
8. Test 1.3.0: it equals the upper bound, so reject it.

Answer. For ^4.18.2, 2 versions are allowed: 4.18.2 and 4.19.0. For ~1.2.3, of the two tested, 1.2.9 is allowed and 1.3.0 is not; the lower bound 1.2.3 and every 1.2.x above it are allowed too.

EXAM TRAPS

- Swapping `^` and `~`: the caret locks the major number, the tilde locks the minor number.
- Treating `^4.18.2` as allowing 5.0.0. The upper bound is excluded.
- Deleting `package-lock.json` and expecting the same install on every machine.
- Running `npm install` on a server when the lock file must be obeyed; that is a job for `npm ci`.
- Editing `node_modules` by hand. It is generated, so the edit is lost on the next install.

CHECK YOURSELF

– `REMEMBER` **What does `package-lock.json` record?** 2 marks

It records the exact versions of installed packages and their dependencies, so installs can be repeated.

– `UNDERSTAND` **Why should a team commit `package-lock.json`?** 3 marks

The lock file fixes exact versions. Every developer and server then installs the same tree instead of resolving the ranges differently.

– `APPLY` **Which of these versions are allowed by `^2.4.1`: 2.3.9, 2.4.1, 2.5.0, 3.0.0?** 4 marks

The range is 2.4.1 ≤ version < 3.0.0. Allowed: 2.4.1 and 2.5.0. Rejected: 2.3.9 is too old, 3.0.0 is a new major version.

– `ANALYZE` **A package.json says `~1.2.3` but an installed package is 1.3.0. Is that allowed? Explain.** 4 marks

No. The tilde range is 1.2.3 ≤ version < 1.3.0. Version 1.3.0 equals the excluded upper bound, so it is outside the range.

– `REMEMBER` **What does `npx` do?** 2 marks

It runs a package command without installing that package globally.

– `APPLY` **Why does a CI pipeline use `npm ci` instead of `npm install`?** 4 marks

`npm ci` deletes node_modules, installs exactly the lock file, and fails on a manifest-lock mismatch. That gives a clean, repeatable build.

– UNDERSTAND **Give one package that belongs in dependencies and one that belongs in devDependencies.** 3 marks

Express belongs in dependencies because the running app needs it. nodemon belongs in devDependencies because it is used only while developing.

BEYOND THE SYLLABUS

Not for the exam. Real teams add npm workspaces for monorepos, private registries for company packages, and audit commands to check known vulnerabilities. Package managers such as pnpm and Yarn solve the same problem with different disk layouts and lock-file formats. Search next for npm workspaces, npm audit, and lock-file review in code review.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
call stack	The single stack of function frames Node's one JavaScript thread is running; anything sitting on it blocks everything else.	6.1
Event loop	The loop that repeatedly asks the operating system what finished, then pushes those callbacks onto the call stack, phase by phase.	6.1
Timers	Event loop phase that runs callbacks whose setTimeout or setInterval delay has expired.	6.1
Poll	Event loop phase where Node waits for completed I/O and runs the matching callbacks; it blocks here when nothing else is ready.	6.1
Check	Event loop phase that runs setImmediate callbacks immediately after the poll phase.	6.1
package.json	Project manifest holding name, scripts and dependency version ranges such as "express": "^4.18.2".	6.2
package-lock.json	Lockfile recording the exact version and download URL of every installed package, so npm ci gives the same tree everywhere.	6.2
semver range	^1.4.2 means ≥1.4.2 and <2.0.0; ~1.4.2 means ≥1.4.2 and <1.5.0; 1.4.2 means that exact version only.	6.2

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

– UNDERSTAND **Explain the Node.js event loop. Name its phases in order and state exactly what runs in each.** 12 marks

Single JS thread plus libuv handles async work; Phases: timers, pending callbacks, poll, check, close; microtasks (promises) drain between phases; poll blocks when no timers are due and no I/O is ready; consequence: never block the stack or every client waits.

– UNDERSTAND **Distinguish package.json from package-lock.json. What exactly does npm install when it sees ^4.17.1 and when it sees ~4.17.1?** 10 marks

manifest vs lockfile; dependencies vs devDependencies; semver caret and tilde ranges with the upper bound; npm install may update the lockfile, npm ci installs it exactly; commit the lockfile for reproducible builds.

Analyze and evaluate

– ANALYZE **A route handler calls fs.readFileSync on every request. Analyse what this does to the event loop and to other users, then rewrite it correctly and justify the change.** 12 marks

readFileSync blocks the only JS thread; poll phase cannot run, so other requests queue; timers fire late, health checks fail; readFile with callback, promises or a stream; throughput rises because waiting moves to the OS and thread pool; same reasoning for JSON.parse of huge bodies and long loops.

Glossary (terms used in this lesson)

Blocking call Code that keeps the JavaScript thread busy, so no other callback can run until it returns.

Core modules Modules shipped inside Node itself, such as fs, http and path, loaded by bare name with require('fs').

devDependencies Packages needed only for testing, linting or building, which a production install can skip.

libuv The C library that gives Node its event loop, worker thread pool and cross-platform async file and network calls.

Local modules	Your own project files, loaded by relative path such as require('./utils') and exported with module.exports.
node_modules	The folder where npm installs packages; bare-name require() searches it upward from the current file.
setImmediate	Schedules a callback to run in the check phase, after the current poll phase finishes.
Thread pool	libuv's small set of worker threads, four by default, that perform file system, DNS and crypto work away from the main thread.

Sources and further reading

Textbooks

- Mario Casciaro, Luciano Mammino, *Node.js Design Patterns*: Chapters on the Node platform, the event loop, streams and modules.
- Ethan Brown, *Web Development with Node and Express*: Routing, middleware, static files, REST APIs and deployment.
- Evan M. Hahn, *Express in Action*: Express routing, middleware and building APIs.
- Shannon Bradshaw, Eoin Brazil, Kristina Chodorow, *MongoDB: The Definitive Guide*: Documents, collections, queries and the MongoDB shell.

Open web resources

Each link below was opened by the book script on 11 Nov 2026 and returned a page; the pages were not read line by line.

- Node.js API documentation: nodejs.org/docs/latest/api
- Express web framework documentation: expressjs.com
- npm documentation: docs.npmjs.com
- MongoDB manual: www.mongodb.com/docs/manual
- OWASP Top Ten web application security risks: owasp.org/www-project-top-ten
- Docker get started guide: docs.docker.com/get-started

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Web Technologies and Full Stack Development (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.