01

Среда выполнения Node.js

Импорт каталога проката: потоки, проверка данных и понятные сообщения об ошибках

Версия	0.2.7 от 2026-11-10
Авторы	Евгений Надточеев
Время	чтение ~90 мин, практика ~11 ч
Код примеров	папка examples/week-01 стартового репозитория
Лицензия	© Евгений Надточеев. Все права защищены.

Содержание

- Перед началом
- Результаты обучения
- Задача проката
- Теория
  - Node.js и браузер: одна модель выполнения
  - Цикл событий
  - Блокировка и как её увидеть
  - Модули и TypeScript без сборки
  - Потоки и обратное давление
  - Проверка данных во время выполнения
  - Деньги: целые копейки
  - Атомарная замена файла
  - Ошибки, коды выхода и вывод
  - Линтер
  - HTTP на node:http и мягкая остановка
- Разобранные примеры
  - Задержка цикла событий
  - Порядок колбэков
  - Обратное давление
  - Сводка по журналу возвратов
  - Атомарная запись небольшого файла
  - Сервер и мягкая остановка
- Практикум
  - Как работать с ИИ-ассистентом в курсе
  - Упражнения
  - Основное задание
- Типовые ошибки и диагностика
- Самопроверка
- Вопросы собеседования
- Что читать дальше
- Глоссарий
- Журнал изменений
- Приложение. Ответы на самопроверку

Перед началом

Методичка рассчитана на разработчика, который уверенно пишет на JavaScript и знает промисы, `async` / `await`, классы и модули. Из TypeScript достаточно уметь читать типы, объединения и дженерики. Терминал, Git и запуск Node.js описаны в методичке 00.

Что понадобится:

- Node.js 22.22 или новее, лучше актуальная LTS (26). Версию покажет `node -v`.
- npm, редактор с поддержкой TypeScript, терминал с `curl`.
- Около 90 минут на чтение и 11 часов на практику: 7 часов на реализацию вместе с упражнениями (сами упражнения занимают около двух часов), 2 на тесты и 2 на исправления после проверки.

Все листинги приведены целиком. Подпись над листингом показывает путь к файлу в репозитории примеров курса. Там же лежат тесты, которые эти листинги проходят. Примеры решают соседние задачи, а не основное задание недели: код задания вы пишете сами.

Результаты обучения

После этой методички вы сможете:

1. Объяснить, почему долгая синхронная работа задерживает все запросы процесса, и измерить эту задержку (К01).
2. Объяснить, в каком порядке выполняются синхронный код, микрозадачи, таймеры и `setImmediate`, и на какой порядок полагаться нельзя (К01).
3. Обработать файл любого размера, не занимая лишней памяти: читать его построчно и писать с учётом обратного давления (К02).
4. Считать внешние данные `unknown`, проверять их схемой во время выполнения и писать такие сообщения об ошибках, чтобы сотрудник сразу нашёл неверную строку (К03).
5. Хранить деньги целым числом копеек и отклонять дробные суммы (К10).
6. Заменять файл атомарно, по коду выхода различать ошибку в данных и сбой чтения или записи и закреплять каждый такой случай тестом на `node:test` (К03, К19).
7. Запустить HTTP-сервер на `node:http` и останавливать его по SIGTERM так, чтобы начатые запросы успели завершиться (К04).
8. Пользоваться ИИ-ассистентом так, чтобы решение оставалось вашим: проверять его предложения запуском и тестами и писать в MR, где он заметно помог (К24).

Задача проката

Пункт проката инструментов переезжает со старой учётной системы на новый сервис. Старая система умеет только выгружать данные в файл JSONL, где каждая строка содержит одну позицию каталога.

```
{"id":"drill-bosch-gsb-13","name":"Дрель ударная Bosch GSB 13 RE","category":"drills",
 "dailyPrice":45000,"deposit":500000,"currency":"RUB","specs":{"powerW":600}}
```

Здесь выгрузка перенесена на две строки ради ширины страницы, а в файле это одна строка. Цена за сутки равна 450 рублям, залог 5 000 рублям, обе суммы записаны в копейках.

Сейчас в каталоге несколько сотен позиций. Когда к сервису подключится сеть пунктов, их станут сотни тысяч, а файл вырастет до сотен мегабайт. Сотрудник запускает импорт и ждёт одного из двух исходов:

- каталог заменён целиком, и выведено, сколько позиций загружено;
- каталог не тронут, и выведен список строк с понятной причиной ошибки.

Хуже всего третий исход, которого сотрудник не ждёт: импорт упал посередине, и в субботу утром на витрине осталась половина каталога. Клиенты не находят инструмент, бронирования уходят конкурентам. Не лучше и случай, когда файл не прочитался, но программа сообщила, что загрузила ноль позиций, и стёрла старый каталог.

Типичные дефекты выгрузки известны заранее:

- дробные цены вроде `450.5`, которые остаются после ручной правки в таблице;
- повторяющиеся `id` после слияния выгрузок двух пунктов, причём не подряд;
- оборванная последняя строка, если выгрузку прервали;
- лишние поля с опечатками, например `dailyprice` вместо `dailyPrice`.

Во второй части недели вы сделаете проверку живости. Сервис будет работать в контейнере, и системе запуска нужен адрес, по которому видно, что процесс отвечает. Это первый HTTP-маршрут сервиса; каталог по HTTP появится позже.

Теория

Node.js и браузер: одна модель выполнения

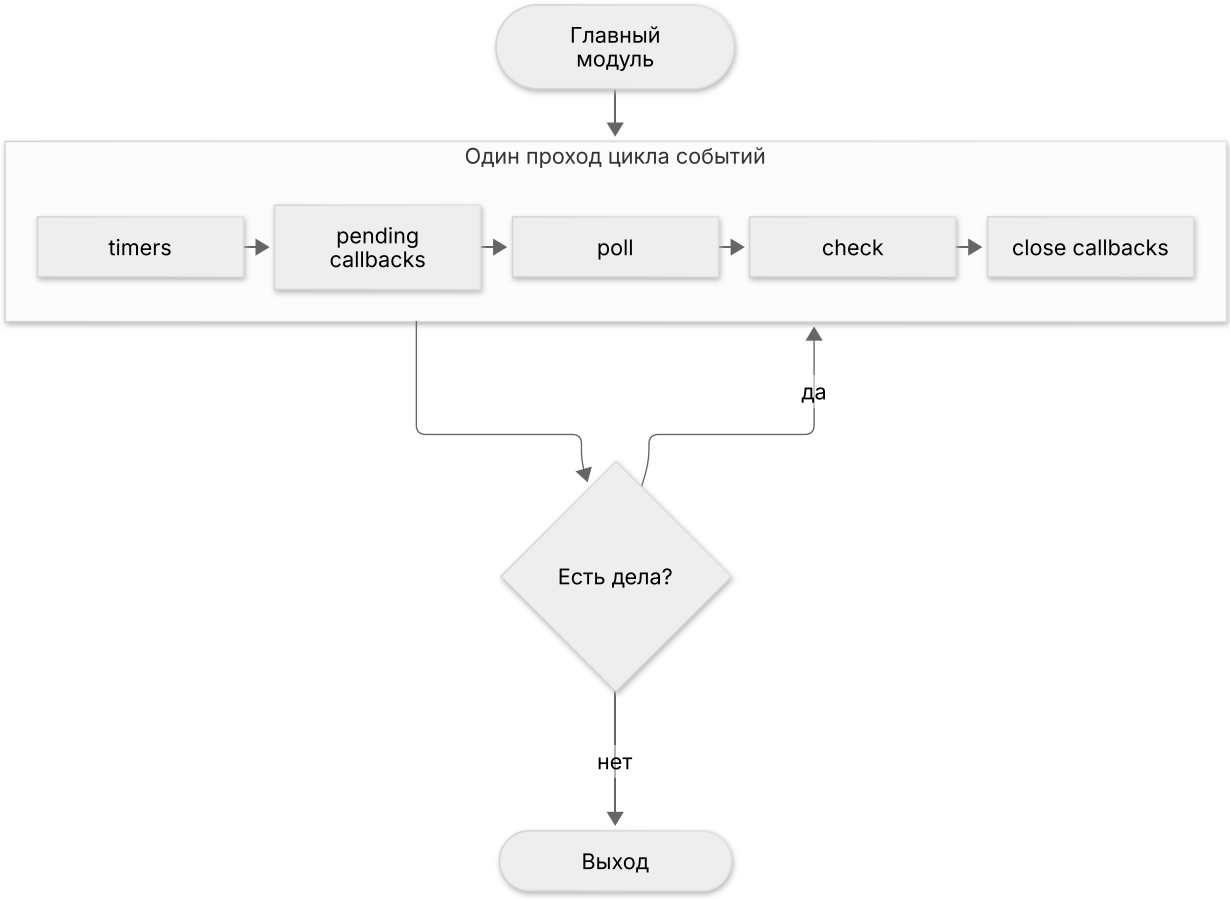
Node.js состоит из трёх частей: движка V8, который выполняет JavaScript, библиотеки libuv, которая отвечает за ввод-вывод и цикл событий, и стандартной библиотеки модулей `node:*`. В браузере те же промисы и `fetch`, но нет файловой системы, сокетов, переменных окружения и кода выхода. В Node.js нет DOM и `window`.

При этом в обеих средах ваш JavaScript выполняется в одном потоке, и это важнее всех различий. Чтение файла или ожидание сети выполняет операционная система или пул потоков libuv, а колбэк с результатом возвращается в этот единственный поток. Пока поток занят вашим кодом, ни один другой колбэк не выполнится.

Ещё одно правило пригодится уже на этой неделе. Процесс завершается, когда в цикле событий не осталось дел: открытых серверов, таймеров и незавершённых операций ввода-вывода. Сервер работает, пока слушает порт. Скрипт импорта завершается, когда дочитал файл и закрыл всё, что открыл. Если скрипт «висит», значит, что-то осталось открытым. Если он завершился раньше времени, значит, вы ждали промис, который ничто не разрешит.

Цикл событий

Цикл событий - это механизм, который берёт готовые колбэки и по очереди выполняет их в единственном JS-потоке. Он проходит по фазам, и в каждой выполняет свою очередь колбэков.



- timers: колбэки `setTimeout` и `setInterval`, у которых истёк срок.
- pending callbacks: колбэки ввода-вывода, отложенные с прошлого прохода.
- poll: процесс получает новые события ввода-вывода и выполняет их колбэки. Если делать нечего, он ждёт в этой фазе.
- check: колбэки `setImmediate`.
- close callbacks: события закрытия, например `socket.on('close')`.

Схема упрощена. С libuv 1.45 (Node.js 20 и новее) таймеры выполняются после фазы poll и ещё один раз перед входом в цикл. Поэтому в главном модуле порядок `setTimeout(fn, 0)` и `setImmediate(fn)` не определён и зависит от загрузки машины. Внутри колбэка ввода-вывода `setImmediate` всегда срабатывает раньше таймера.

Между фазами есть ещё две очереди, которые формально к циклу не относятся. Первая - очередь `process.nextTick`, она есть только в Node.js. Вторая - очередь микрозадач, то есть колбэков промисов и `queueMicrotask`.

Обе очереди выполняются полностью после каждой операции, прежде чем цикл перейдёт дальше. Поэтому бесконечная цепочка микрозадач блокирует процесс так же, как бесконечный цикл: до таймеров и сети очередь не дойдёт.

Пример `order.ts` из раздела примеров заводит по одному колбэку каждого вида. Вот какой порядок он выводит в разных средах:

Где выполняется	Порядок
CommonJS (`.cts`), Node.js 22-26	sync → nextTick → promise → immediate → timeout
ES-модуль, Node.js 22 и 24	sync → promise → nextTick → …
ES-модуль без top-level `await`, Node.js 26.12	sync → nextTick → promise → …
ES-модуль после top-level `await`, Node.js 26.12	sync → promise → nextTick → …

Последние две позиции, `immediate` и `timeout`, в главном модуле могут меняться местами. У ES-модулей порядок изменился, когда Node.js начал загружать большинство из них синхронно: в версии 26.12 главный модуль без top-level `await` выполняется вне очереди микрозадач. Поэтому не стройте логику на порядке между `nextTick` и промисами. Если одно действие должно выполниться после другого, свяжите их явно через `await`.

Блокировка и как её увидеть

Сервис проката обслуживает запросы клиентов и импорт в одном процессе. Допустим, импорт пересчитывает цены по всему каталогу синхронным циклом, и это занимает две секунды. Все две секунды не отвечает никто: ни клиент, ни проверка живости. Система запуска решит, что процесс завис, и перезапустит его.

Типичные источники блокировки в backend-коде:

- синхронные функции файловой системы (`readFileSync`) в обработчике запроса;
- `JSON.parse` и `JSON.stringify` больших объёмов;
- синхронная криптография (`pbkdf2Sync`, `scryptSync`);
- регулярные выражения с катастрофическим перебором;
- сортировки и вложенные циклы по всему набору данных.

Блокировку видно по задержке таймера. Пример `event-loop-lag.ts` заводит таймер на 5 мс, выполняет работу и измеряет, насколько позже срока таймер сработал. Синхронная работа на 300 мс задерживает таймер на все 300 мс. Если выполнить ту же работу порциями по 10 мс и между порциями вызывать `await setImmediate()`, таймер опоздает примерно на одну порцию.

Порции не ускоряют работу: она длится столько же или чуть дольше. Зато между порциями поток успевает обслужить остальные задачи. Тяжёлый расчёт, который нельзя разбить, выносят в `worker_threads` или в отдельный процесс. В задачах курса такого расчёта нет, поэтому дальше эта тема не разбирается. С чего начать, сказано в разделе «Что читать дальше». В работающем сервисе задержку цикла событий меряют гистограммой `monitorEventLoopDelay` из `node:perf_hooks` и выводят в метрики.

Модули и TypeScript без сборки

Проект курса состоит из ES-модулей: в `package.json` указано `"type": "module"`. Файлы `.ts` и `.js` загружаются как ES-модули, а `.cts` и `.cjs` как CommonJS. Расширение в относительном импорте обязательно: пишите `import './item.ts'`, без расширения (`import './item'`) импорт не сработает. Путь к текущему файлу дают `import.meta.filename` и `import.meta.dirname`; `__filename` и `__dirname` в ES-модулях недоступны.

Node.js запускает TypeScript напрямую, удаляя аннотации типов (type stripping). С версий 22.18 и 23.6 это включено по умолчанию, а с 25.2 и 24.12 считается стабильным. Но у этого подхода есть ограничения.

Node.js не проверяет типы. Он вырезает аннотации и выполняет то, что осталось. Ошибка типа в коде доживёт до продакшена, если не запускать `tsc`. В проекте курса это команда `npm run typecheck`, и она входит в `npm run check` вместе с линтером и тестами.

Поддерживается только стираемый синтаксис. Всё, что TypeScript превращает в JavaScript-код, вызывает ошибку `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX`. Это `enum`, `namespace` с кодом, свойства в параметрах конструктора и `import x = require()`. Флаг `--experimental-transform-types`, который позволял это обойти, в Node.js 26 убрали. Вместо `enum` используйте объединение строковых литералов или объект с `as const`.

Импорт типа нужно пометить словом `type`. Без него Node.js считает импорт значением и падает при загрузке:

```
SyntaxError: The requested module './t2.ts' does not provide an export named 'Item'
```

Файл `tsconfig.json` Node.js не читает. Пути вроде `@/…` и понижение синтаксиса не работают. Файлы TypeScript внутри `node_modules` Node.js не выполняет: это сделано намеренно, потому что библиотеки публикуют JavaScript.

Чтобы проверка типов совпадала с тем, как Node.js запускает код, в `tsconfig.json` включают такие настройки:

Настройка	Зачем
`"module": "nodenext"`	Модули разрешаются так же, как в Node.js
`allowImportingTsExtensions` и `noEmit`	Разрешают `.ts` в импортах; `tsc` только проверяет
`verbatimModuleSyntax`	Требует `import type`, если импортируется только тип
`erasableSyntaxOnly`	Запрещает `enum` и другое, что Node.js не умеет стирать
`strict`, `noUncheckedIndexedAccess`	У `arr[i]` тип `T | undefined`, как и бывает на деле
`exactOptionalPropertyTypes`	Различает «поля нет» и «поле равно undefined»

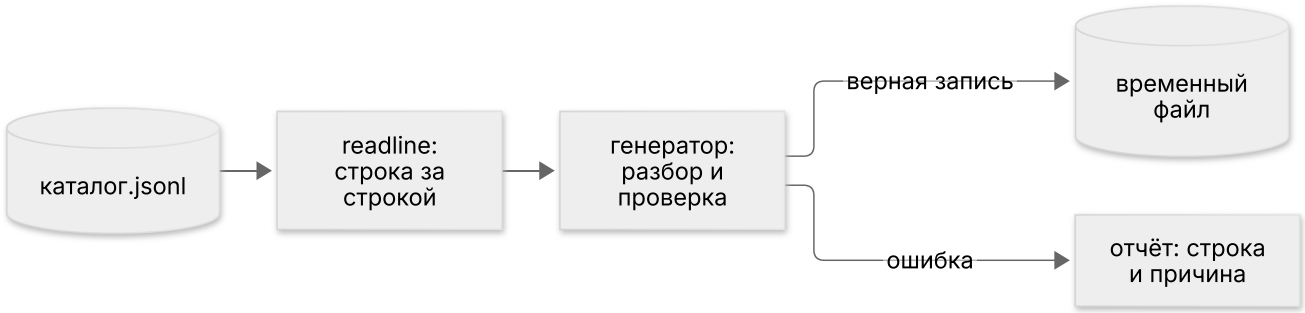
Потоки и обратное давление

Файл на сотни мегабайт нельзя прочитать целиком: `readFile` положит его в память одной строкой, а `JSON.parse` каждой записи удвоит расход. Процесс с ограничением памяти в контейнере упадёт с `JavaScript heap out of memory`. Поток (stream) отдаёт данные порциями, и в памяти одновременно находится только небольшой буфер.

У потока на запись (Writable) есть внутренний буфер с порогом `highWaterMark`. Метод `write()` принимает данные всегда, но возвращает `false`, если буфер заполнен выше порога. Это сигнал остановиться и дождаться события `drain`. Такая обратная связь от медленного приёмника к быстрому источнику называется обратным давлением (backpressure).

Код, который игнорирует `false`, работает на маленьком файле и падает на большом: всё, что диск не успел записать, копится в памяти. В примере `backpressure.ts` 5 000 строк пишутся в медленный приёмник с буфером 1 КБ. Если не ждать `drain`, в буфере скапливается больше 50 КБ, то есть весь файл. Если ждать, буфер не выходит за порог плюс одну строку.

Вручную ждать `drain` приходится редко. Функция `pipeline` из `node:stream/promises` соединяет источник, преобразования и приёмник. Она сама соблюдает обратное давление, передаёт ошибку любого звена и закрывает все потоки при сбое. Источником может быть асинхронный генератор:



Пока приёмник занят, `pipeline` не запрашивает у генератора следующую строку, а генератор не читает следующую строку из файла. Сколько памяти займёт импорт, зависит только от буферов потоков. Размер файла на это не влияет.

Для построчного чтения подходит `readline.createInterface` с циклом `for await`. Параметр `crlfDelay: Infinity` нужен для файлов из Windows, где строки разделены `\r\n`. Без него `\r` и `\n`, пришедшие с паузой, считаются двумя переводами строки, и нумерация строк собьётся. Документация отмечает, что асинхронный итератор медленнее событий `'line'`; для импорта это несущественно.

Есть ещё одна тонкость. Документация `readline` утверждает, что ошибки входного потока в асинхронный итератор не передаются. В Node.js 22, 24 и 26 итератор такую ошибку бросает, это проверено тестом. Когда текст документации и поведение расходятся, нужное поведение закрепляют собственным тестом и не полагаются на формулировку.

Проверка данных во время выполнения

Типы TypeScript стираются до запуска. `JSON.parse` возвращает `any`, и если присвоить результат переменной типа `CatalogItem`, ничего не проверится: компилятор верит вам на слово. Данные из файла, сети и переменных окружения принимайте как `unknown` и превращайте в доменный тип только после проверки.

Проверку удобно описывать схемой. В курсе для этого используется zod 4. Схема описывает ограничения, а `safeParse` возвращает либо данные, либо список нарушений. Тип выводится из схемы через `z.infer`, поэтому схема и тип не расходятся.

Несколько решений стоит принимать осознанно:

- Лишние поля считаются ошибкой. `z.strictObject` отклоняет неизвестные ключи. Иначе опечатка `dailyprice` молча потеряет цену, а обязательное `dailyPrice` окажется пропущенным.
- `JSON.parse` перехватывайте узко. Ловите только `SyntaxError`, а любую другую ошибку пробрасывайте. `catch`, который ловит всё подряд, превратит нехватку памяти в «неверный JSON».
- Нумеруйте строки по файлу, включая пустые. Сотрудник откроет файл в редакторе и будет искать эту строку.
- Пишите сообщения для читателя. Встроенная локализация zod даёт «Неверный ввод: ожидалось string, получено undefined». Разработчику это понятно, а сотруднику пункта проката нет. Для отчёта импорта задайте сообщения в схеме: «name: обязательное поле», «dailyPrice: должно быть целым числом копеек».
- Ограничивайте длину отчёта. В файле со ста тысячами битых строк не нужно сто тысяч сообщений. Показывайте первые N и общее количество.

Глобальная настройка `z.config()` меняет поведение всех схем процесса, в том числе в чужих модулях. Поэтому язык сообщений лучше передавать при вызове, как в примере `returns-summary.ts`.

Деньги: целые копейки

Числа в JavaScript хранятся в двоичном формате с плавающей точкой: `0.1 + 0.2` равно `0.30000000000000004`. Такие погрешности накапливаются в суммах и скидках, а округление в разных местах кода превращает их в расхождение на копейку, которое потом ищут по всей кассе за месяц. Поэтому суммы хранят целым числом минимальных единиц: 450 рублей - это `45000` копеек.

Целые числа в `number` точны до `Number.MAX_SAFE_INTEGER`. Это 9 007 199 254 740 991 копейка, около 90 триллионов рублей. Для цены и залога запас огромный. `z.int()` в zod 4 принимает только безопасные целые числа.

Дробную сумму в выгрузке не округляйте, а отклоняйте. Если цена равна `450.5` копейки, то в источнике, скорее всего, перепутали рубли с копейками или поправили ячейку руками. Округление спрячет эту ошибку источника, а отказ её покажет.

Валюту храните явно, даже если пока она одна. Поле `currency: "RUB"` почти ничего не стоит, а без него первая же интеграция с другой валютой превратится в поиск по всему коду.

Атомарная замена файла

Если писать результат прямо в файл каталога, сбой посередине оставит половину файла. Сбоем может стать ошибка в данных на десятитысячной строке, нехватка места на диске или остановка процесса. Программа, которая в этот момент читает каталог, увидит обрывок.

Надёжная схема состоит из трёх шагов:

1. Писать во временный файл в той же папке, что и целевой.
2. Дождаться, пока данные дойдут до диска: опция `flush: true` вызывает `fsync` перед закрытием файла.
3. Переименовать временный файл поверх целевого через `rename`.

По POSIX `rename` заменяет существующий файл так, что другой процесс видит либо старую версию, либо новую, но не промежуточную. Эта гарантия действует только в пределах одной файловой системы. Временный файл в `/tmp` может оказаться на другом разделе, и тогда `rename` упадёт с `EXDEV`. На Windows замена может не пройти, если целевой файл держит открытым другой процесс (`EPERM`, `EBUSY`).

Без `fsync` после сбоя питания можно получить новое имя, которое указывает на ещё не записанные данные, то есть пустой файл. Чтобы пережить сбой питания полностью, после `rename` нужен ещё `fsync` папки. Методичка его не требует, но знать, где кончается гарантия, нужно.

Временный файл открывайте с флагом `wx`: если файл уже есть, открытие падает, и чужой файл вы не перезапишете. При любой ошибке временный файл удаляйте. Если процесс убьют сигналом `SIGKILL`, удалить файл он не успеет. Поэтому имя временного файла должно быть узнаваемым, чтобы такие остатки было легко найти.

Ошибки, коды выхода и вывод

У программы импорта четыре исхода, и вызывающая сторона должна их различать. Сотрудник читает текст, а скрипт расписания или CI смотрит только на код выхода.

Код	Исход	Пример
0	Успех	Каталог заменён
1	Ошибка в данных	Дробная цена в строке 17
2	Неверный вызов	Не передан `--input`
3	Сбой ввода-вывода	Файл не найден, нет прав, диск заполнен

Единого стандарта кодов нет. В BSD есть `sysexits.h` со значениями 64-78, а многие утилиты используют 1 и 2. Нужно, чтобы коды различались и были описаны в README. И никогда не выдавайте сбой за успех. Если входного файла нет, это не пустой каталог. Если запись не удалась, это не успех.

Ошибку ввода-вывода удобно отличать собственным классом с исходной причиной в `cause`:

```
export class ReportWriteError extends Error {
  override name = 'ReportWriteError'
}

throw new ReportWriteError(`не удалось записать отчёт: ${file}`, { cause })
```

Код выхода задавайте через `process.exitCode`, а не через `process.exit()`. Запись в `process.stdout` при выводе в канал (pipe) на Linux и macOS асинхронна, и `exit()` завершает процесс, не дожидаясь её. Проверьте сами: скрипт выводит 200 000 строк и вызывает `process.exit(0)`, а медленный читатель канала получает около 3 700 строк. С `process.exitCode = 0` доходят все 200 000.

Сообщение для сотрудника должно содержать причину и место ошибки, без стека вызовов и внутренних путей. Подробности для разработчика уходят в лог. Отклонённый промис, который никто не обработал, по умолчанию завершает процесс с кодом 1 и стеком в stderr. Для непредвиденной ошибки это правильное поведение, но ожидаемые исходы нужно обрабатывать явно.

Линтер

Тесты замечают не каждый промис, которого никто не ждёт: на их данных он может просто не отклоняться. Такие промисы находит линтер, команда `npm run lint` (что это и чем он отличается от проверки типов, рассказывает методичка 00). Вот как он реагирует на функцию, которая пишет отчёт и не ждёт записи:

```
import { writeFile } from 'node:fs/promises'

export function saveReport(file: string, text: string): void {
  writeFile(file, text)
}
```

```
/home/you/rental/src/cli/report.ts
  4:3  error  Promises must be awaited, end with a call to .catch, end with a call
              to .then with a rejection handler or be explicitly marked as ignored
              with the `void` operator  @typescript-eslint/no-floating-promises
```

Сообщение линтер печатает одной строкой, здесь она перенесена. `4:3` означает строку 4, столбец 3, а в конце стоит имя правила: `no-floating-promises`, висящий промис (floating promise). Правило находит выражение, которое возвращает промис, если этот промис никто не ждёт и его отказ никто не обработает. Здесь это действительно ошибка. Функция вернётся раньше, чем отчёт будет записан. Если не хватит места, получится необработанный отказ промиса (unhandled rejection), и он завершит процесс с кодом 1.

Исправить это можно тремя способами. Выбирайте по смыслу, а не по длине записи:

- `await`, если результат нужен дальше, а ошибка должна прервать функцию. Для отчёта подходит этот вариант: функция становится `async` и возвращает `Promise<void>`;

- `.catch(…)` с обработкой, если ждать незачем, но отказ есть кому разобрать. Так `shutdown` в примере сервера ловит ошибку `stop()`: пишет сообщение в stderr и ставит код выхода 1;
- `void`, если результат намеренно не нужен, а отказа быть не может. Так помечена микрозадача в примере порядка колбэков: она только ставит отметку в массив.

`void` нужен не для того, чтобы заглушить линтер. Он сообщает читателю, что промис брошен намеренно. Если отказ возможен, `void` оставит его необработанным, как и раньше.

Соседнее правило `no-misused-promises` находит async-функцию там, где вызывающий код промис не ждёт: `process.once('SIGTERM', async () => …)` или обработчик запроса в `createServer`. Сигнал придёт, обработчик вернёт промис, а `process` его выбросит, и отказ внутри станет необработанным. Поэтому `shutdown` в примере сервера - обычная функция, а промис остановки она обрабатывает сама.

Ещё два правила относятся к `catch`. Правило `no-empty` запрещает пустой `catch {}`, потому что ошибка исчезает без следа. Блок с одним комментарием линтер пропускает, поэтому комментарий должен объяснять, почему ошибку можно потерять; это проверяет ревью. Правило `preserve-caught-error` требует, чтобы новая ошибка, брошенная из `catch`, несла исходную в `cause`, как `ReportWriteError` выше.

Правило можно отключить строкой над кодом: `// eslint-disable-next-line <правило> -- причина`. Это последнее средство, сначала нужно исправить код по существу. В коде курса отключений нет.

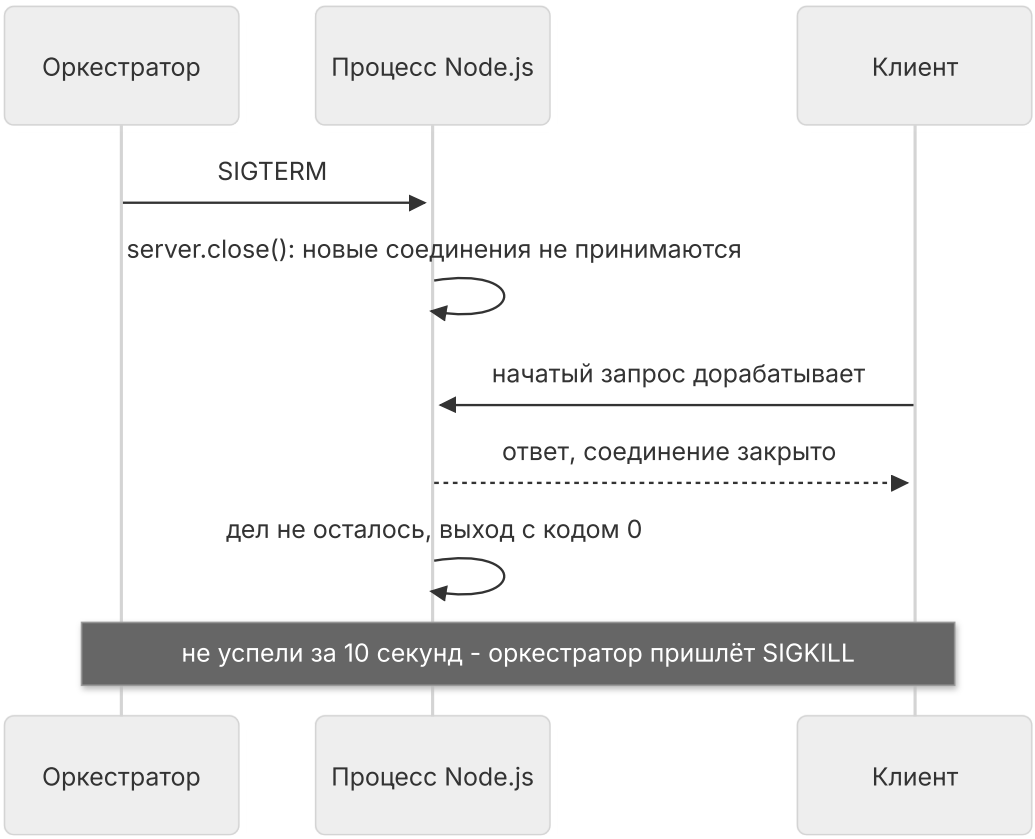
HTTP на node:http и мягкая остановка

`node:http` - низкоуровневый сервер: на каждый запрос вызывается функция с объектами запроса и ответа. Маршрутизацию, разбор тела и формат ошибок вы пишете сами; на четвёртой неделе это возьмёт на себя NestJS. Для одного маршрута проверки живости фреймворк не нужен.

Но и с одним маршрутом нужно учесть несколько вещей:

- Путь берите из `new URL(req.url, 'http://localhost').pathname`, чтобы строка запроса `?probe=1` не ломала маршрут.
- Отвечайте JSON с заголовком `content-type` и для успеха, и для ошибки. Клиент не должен угадывать формат по коду ответа.
- На неподдерживаемый метод отвечайте 405 с заголовком `Allow`, на неизвестный путь - 404. RFC 9110 требует `Allow` в ответе 405.
- Проверка живости не обращается к базе и внешним сервисам. Она отвечает на вопрос «процесс жив и обслуживает цикл событий?». Готовность принимать трафик проверяется отдельно, когда появятся зависимости.

Теперь об остановке. Docker, Kubernetes и systemd останавливают процесс сигналом SIGTERM, а если он не завершился за отведённое время, присылают SIGKILL. У `docker stop` это время по умолчанию 10 секунд. Мягкая остановка выглядит так:



Здесь есть подвох с keep-alive. Клиенты держат соединение открытым после ответа, чтобы отправить по нему следующий запрос. С Node.js 19 `server.close()` закрывает соединения, которые простаивают в момент вызова. Соединение запроса, который в этот момент ещё выполнялся, после ответа остаётся открытым до `keepAliveTimeout` или пока его не закроет клиент. В опыте с `fetch` это дало 3,4 секунды лишнего ожидания. Пример `graceful-stop.ts` решает это так: после начала остановки он вызывает `server.closeIdleConnections()` каждый раз, когда завершается ответ.

Порт берите из переменной окружения `PORT` и проверяйте строку, а не результат `Number()`. `Number('abc')` даёт `NaN`, и `listen` бросит `ERR_SOCKET_BAD_PORT`, то есть ошибка хотя бы заметна. Хуже с `Number('')`: пустая строка превращается в 0, и сервер молча займёт случайный порт, по которому его никто не найдёт. Так же молча `Number()` принимает `' 80'`, `'0x50'` и `'1e3'`. При этом в тестах `PORT=0` задают намеренно: система выдаст свободный порт, и узнать его можно через `server.address()`.

Разобранные примеры

Задержка цикла событий

examples/week-01/event-loop-lag.ts

```ts
import { setImmediate as yieldToLoop } from 'node:timers/promises'

/** Занимает поток на ms миллисекунд — так выглядит тяжёлый синхронный расчёт. */
export function blockFor(ms: number): void {
  const end = performance.now() + ms
  while (performance.now() < end) {
    // Пустой цикл намеренно: он изображает пересчёт цен по всему каталогу.
  }
}

/** Та же работа порциями: между порциями цикл событий обслуживает таймеры и сеть. */
export async function blockInSlices(totalMs: number, sliceMs: number): Promise<void> {
  for (let done = 0; done < totalMs; done += sliceMs) {
    blockFor(Math.min(sliceMs, totalMs - done))
    await yieldToLoop()
  }
}

/** На сколько миллисекунд позже срока сработал таймер, заведённый перед работой. */
export async function measureTimerLag(
  work: () => unknown,
  delayMs = 5,
): Promise<number> {
  const started = performance.now()
  const fired = new Promise<number>((resolve) => {
    setTimeout(() => resolve(performance.now()), delayMs)
  })
  await work()
  return (await fired) - started - delayMs
}
```

Что заметить:

- `measureTimerLag(() => blockFor(300))` возвращает не меньше 250 мс, а `measureTimerLag(() => blockInSlices(300, 10))` меньше 150 мс. Тесты проверяют эти границы с запасом на медленную машину.
- `setImmediate` из `node:timers/promises` позволяет «уступить» поток: остаток работы продолжится после того, как цикл обслужит накопившиеся события.

Порядок колбэков

examples/week-01/order.ts

```ts
// Порядок колбэков в ES-модуле. Сравните с order.cts и с выводом на разных
// версиях Node: место nextTick относительно промиса здесь менялось.
const order: string[] = []

setTimeout(() => order.push('timeout'), 0)
setImmediate(() => order.push('immediate'))
void Promise.resolve().then(() => order.push('promise'))
process.nextTick(() => order.push('nextTick'))
order.push('sync')

setTimeout(() => console.log(order.join(' → ')), 50)
```

Файл `order.cts` содержит тот же код. Тест проверяет только устойчивые части порядка: синхронный код идёт первым, обе микрозадачи выполняются раньше таймера и `setImmediate`, а в CommonJS `nextTick` идёт раньше промиса. Порядок `nextTick` и промиса в ES-модуле тест намеренно не проверяет. `void` перед промисом сообщает линтеру и читателю, что результат брошен намеренно (раздел «Линтер»).

Обратное давление

examples/week-01/backpressure.ts

```ts
import { once } from 'node:events'
import type { Writable } from 'node:stream'

/** Так делать нельзя: write() вернул false, а мы продолжаем писать в буфер. */
export function writeIgnoringBackpressure(
  out: Writable,
  lines: Iterable<string>,
): void {
  for (const line of lines) out.write(line)
}

/** Пишет строки и ждёт drain, когда внутренний буфер потока заполнен. */
export async function writeLines(
  out: Writable,
  lines: Iterable<string>,
): Promise<void> {
  for (const line of lines) {
    // once() отклоняет промис при 'error', поэтому сбой записи не повиснет.
    if (!out.write(line)) await once(out, 'drain')
  }
}
```

Что заметить:

- Обе функции «работают» на маленьких данных. Разница видна только по объёму буфера под нагрузкой, поэтому тест меряет `writableLength`, а не результат.
- `once` из `node:events` превращает событие в промис и отклоняет его, если поток сообщил об ошибке. Ожидание `drain` на сломанном потоке не повиснет.

Сводка по журналу возвратов

Возврат инструмента клиентом записывается в журнал: какая аренда, какой инструмент и сколько удержано за повреждение. Пример считает сводку и собирает строки с ошибками. Тот же приём нужен в импорте каталога, только у примера другая запись и он не заменяет файл.

examples/week-01/returns-summary.ts

```ts
import { createReadStream } from 'node:fs'
import { createInterface } from 'node:readline'
import { z } from 'zod'

// Журнал возвратов: одна строка — один возврат инструмента клиентом.
const ReturnRecord = z.strictObject({
  rentalId: z.string().min(1),
  itemId: z.string().min(1),
  damageKopecks: z.int().nonnegative(),
})

const MAX_PROBLEMS = 20
// Русские сообщения zod подключаем на вызов, а не глобально через z.config().
const ru = z.locales.ru()

export type Problem = { line: number; reason: string }
export type ReturnsSummary = {
  returns: number
  damageKopecks: number
  problems: Problem[]
  totalProblems: number
}

/** Сводка по журналу возвратов; память не растёт с размером файла. */
export async function summarizeReturns(file: string): Promise<ReturnsSummary> {
  const summary: ReturnsSummary = {
    returns: 0,
    damageKopecks: 0,
    problems: [],
    totalProblems: 0,
  }
  const problem = (line: number, reason: string) => {
    summary.totalProblems += 1
    if (summary.problems.length < MAX_PROBLEMS) summary.problems.push({ line, reason })
  }

  const lines = createInterface({
    input: createReadStream(file, { encoding: 'utf8' }),
    crlfDelay: Infinity,
  })
  let lineNo = 0
  for await (const text of lines) {
    lineNo += 1
    if (text.trim() === '') continue

    let raw: unknown
    try {
      raw = JSON.parse(text)
    } catch (error) {
      if (!(error instanceof SyntaxError)) throw error
```

```
      problem(lineNo, 'неверный JSON')
      continue
    }

    const parsed = ReturnRecord.safeParse(raw, { error: ru.localeError })
    if (!parsed.success) {
      const issue = parsed.error.issues[0]
      const field = issue?.path.join('.') || 'запись'
      problem(lineNo, `${field}: ${issue?.message ?? 'неизвестная ошибка'}`)
      continue
    }
    summary.returns += 1
    summary.damageKopecks += parsed.data.damageKopecks
  }
  return summary
}
```

На файле с четырьмя дефектами пример выдаёт такой отчёт:

```
строка 2: неверный JSON
строка 3: damageKopecks: Слишком маленькое значение: ожидалось, что number будет >=0
строка 4: itemId: Неверный ввод: ожидалось string, получено undefined
строка 5: запись: Нераспознанный ключ: "note"
```

Что заметить:

- Пустые строки пропускаются, но номер строки всё равно растёт.
- Сообщения переведены на русский, но написаны для разработчика. Сравните их с «itemId: обязательное поле»: в задании сообщения стоит писать так.
- Пример не ищет дубли. Для этого в импорте каталога нужно помнить уже встреченные `id`, и эта память растёт вместе с каталогом. Поэтому у импорта есть объявленный предел числа позиций.

Атомарная запись небольшого файла

examples/week-01/write-atomic.ts

```ts
import { randomBytes } from 'node:crypto'
import { rename, rm, writeFile } from 'node:fs/promises'
import path from 'node:path'

/**
 * Заменяет файл целиком или не трогает его вовсе. Подходит для небольших
 * данных, которые уже лежат в памяти: прайс-лист, настройки, отчёт.
 */
export async function writeFileAtomic(target: string, data: string): Promise<void> {
  // Та же папка: rename атомарен только в пределах одной файловой системы.
  const tmp = path.join(
    path.dirname(target),
    `.${path.basename(target)}.tmp-${randomBytes(6).toString('hex')}`,
  )
  try {
    // wx: не перезаписать чужой временный файл; flush: fsync до rename.
    await writeFile(tmp, data, { flag: 'wx', flush: true })
    await rename(tmp, target)
  } catch (error) {
    await rm(tmp, { force: true })
    throw error
  }
}
```

Что заметить:

- Здесь данные целиком лежат в памяти. В импорте каталога накапливать их нельзя: временный файл пишется потоком (`createWriteStream` с теми же `flags: 'wx'` и `flush: true`), а `rename` выполняется, только если весь вход оказался верным.
- `rm` с `force: true` не падает, если временный файл так и не был создан.
- Ошибка пробрасывается дальше. Что сказать пользователю, решит вызывающий код.

Сервер и мягкая остановка

examples/week-01/version-server.ts

```ts
import { createServer, type Server } from 'node:http'

/** Сервер с одним маршрутом: какая версия сервиса сейчас запущена. */
export function createVersionServer(version: string): Server {
  return createServer((req, res) => {
    const { pathname } = new URL(req.url ?? '/', 'http://localhost')
    const found = pathname === '/version' && req.method === 'GET'
    const body = JSON.stringify(found ? { version } : { error: 'not_found' })
    res.writeHead(found ? 200 : 404, {
      'content-type': 'application/json; charset=utf-8',
      'content-length': Buffer.byteLength(body),
    })
    res.end(body)
  })
}
```

examples/week-01/graceful-stop.ts

```ts
import type { Server } from 'node:http'

/**
 * Готовит сервер к мягкой остановке и возвращает функцию остановки:
 * новые соединения не принимаются, начатые запросы дорабатывают.
 */
export function prepareStop(server: Server): () => Promise<void> {
  let stopping = false
  // close() закрывает только соединения, которые простаивают в момент вызова.
  // Keep-alive соединение запроса, начатого раньше, иначе живёт до keepAliveTimeout.
  server.on('request', (_req, res) => {
    res.on('finish', () => {
      if (stopping) server.closeIdleConnections()
    })
  })
  return () =>
    new Promise<void>((resolve, reject) => {
      stopping = true
      server.close((error) => (error ? reject(error) : resolve()))
    })
}
```

examples/week-01/run-version-server.ts

```ts
import type { AddressInfo } from 'node:net'
import { createVersionServer } from './version-server.ts'
import { prepareStop } from './graceful-stop.ts'

const rawPort = process.env.PORT ?? '3000'
// Number() молча принял бы '' (это 0, то есть случайный порт), ' 80', '0x50' и '1e3'.
const port = /^\d{1,5}$/.test(rawPort) ? Number(rawPort) : Number.NaN
if (!(port <= 65_535)) {
  process.stderr.write(`PORT должен быть целым числом 0–65535: «${rawPort}»\n`)
  process.exit(2)
}

const server = createVersionServer(process.env.APP_VERSION ?? 'dev')
const stop = prepareStop(server)
server.listen(port, () => {
  const { port: actual } = server.address() as AddressInfo
  process.stdout.write(`${JSON.stringify({ msg: 'listening', port: actual })}\n`)
})

function shutdown(signal: NodeJS.Signals): void {
  process.stdout.write(`${JSON.stringify({ msg: 'shutdown', signal })}\n`)
  stop().catch((error: unknown) => {
    process.stderr.write(`остановка не удалась: ${String(error)}\n`)
    process.exitCode = 1
  })
  // Страховка от зависшего клиента; unref() не держит процесс, если всё закрылось.
  setTimeout(() => process.exit(1), 10_000).unref()
}

process.once('SIGTERM', shutdown)
process.once('SIGINT', shutdown)
```

Что заметить:

- `process.exit(2)` при неверном порте допустим: к этому моменту ничего не выведено в канал и не открыто, терять нечего.
- Проверяется строка из окружения, а не результат `Number()`. Тесты запускают сервер с `PORT`, равным `''`, `'abc'`, `' 80'`, `'0x50'` и `'70000'`, и ждут код 2.
- После SIGTERM процесс не вызывает `exit()` при успехе. Он закрывает сервер и завершается сам, когда в цикле событий не остаётся дел. Тест проверяет код выхода 0 и время остановки.
- Страховочный таймер помечен `unref()` и сам процесс не держит. Он сработает, только если что-то другое держит процесс дольше 10 секунд.

Практикум

Как работать с ИИ-ассистентом в курсе

ИИ-ассистент на курсе разрешён как рабочий инструмент. За код, тесты и решения отвечаете вы: на ревью и на защите вас попросят объяснить и изменить любую строку.

Можно спрашивать, как устроен API и почему падает тест, просить разобрать сообщение об ошибке, предложить тестовые случаи или проверить текст README. Задание недели целиком лучше не отдавать ассистенту: недели опираются друг на друга, и пропущенный приём первой недели обойдётся дороже на пятой.

Предложенное ассистентом проверяйте так:

- запуском: ассистент уверенно описывает устаревшее поведение и опции, которых нет, поэтому поведение Node.js, базы или брокера проверяйте на вашей версии;
- документацией из раздела «Что читать дальше», а не пересказом;
- тестом: если убрать правило, которое ассистент «учёл», тест должен упасть;
- чтением: строку, которую не можете объяснить, перепишите сами или удалите.

Секреты, `.env` и журналы с личными данными ассистенту не отправляйте.

Существенную помощь указывайте в разделе «Помощь ИИ» описания MR: что сделал ассистент, что сделали вы, как вы проверили результат и где ассистент ошибся. Помощь считается существенной, когда решение, запрос, тест или текст ассистента вошли в проект почти без изменений или его объяснение определило ваше решение. Автодополнение имени переменной к существенной помощи не относится. Если вы указали помощь и понимаете, что сделал ассистент, оценка от этого не снижается. К защите эти записи соберутся в раскрытие помощи ИИ (методичка 20).

Упражнения

1. Предскажите порядок, 15 минут. Запишите на бумаге, что выведет код, и только потом запустите его как `.mjs` и как `.cjs`:

```
setTimeout(() => console.log('A'), 0)
Promise.resolve().then(() => {
  console.log('B')
  process.nextTick(() => console.log('C'))
  Promise.resolve().then(() => console.log('D'))
})
setImmediate(() => console.log('E'))
console.log('F')
```

Объясните каждую позицию: почему `D` выводится раньше `C` и какие две буквы могут поменяться местами от запуска к запуску.

2. Измерьте блокировку, 30 минут. Добавьте в сервер версии маршрут `/reprice`, который вызывает `blockFor(2000)`. В одном терминале запросите `/reprice`, а в другом сразу после этого запросите `/version` с замером времени:

```
curl -s -o /dev/null -w '%{time_total}\n' http://127.0.0.1:3000/version
```

Замените `blockFor` на `blockInSlices(2000, 10)` и повторите. Запишите оба времени и объясните разницу.

3. Обратное давление на диске, 20 минут. Напишите скрипт, который пишет два миллиона строк отчёта в файл через `createWriteStream`: сначала не глядя на результат `write()`, потом через `writeLines`. Сравните пиковый `process.memoryUsage().rss`.

4. Сообщения для человека, 20 минут. Перепишите схему журнала возвратов так, чтобы отчёт выглядел как «itemId: обязательное поле» и «damageKopecks: не может быть отрицательным». Подсказка: параметр `error` у схемы поля.

5. Обрезанный вывод, 15 минут. Повторите опыт с 200 000 строк и `process.exit(0)`. Передайте вывод медленному читателю `node script.mjs | (sleep 1; wc -l)`. Затем замените `exit()` на `exitCode`.

6. Что останется после сбоя, 20 минут. Вставьте в `writeFileAtomic` паузу на 10 секунд между `writeFile` и `rename` и завершите процесс через `kill -9`. Какие файлы остались в папке? Что увидит программа, которая читает целевой файл? Как найти и убрать остатки?

Основное задание

Напишите CLI импорта каталога и сервер проверки живости по требованиям ниже. В вашем проекте с недели 0 уже есть инструменты, конфигурация TypeScript, файлы-образцы и заглушки `src/cli/import-catalog.ts` и `src/http/server.ts`. MR «Материалы недели 1» добавляет к ним публичные тесты `test/week-01`. Тесты падают, пока заглушки не заменены вашей реализацией.

Импорт запускается так: `node src/cli/import-catalog.ts --input <файл.jsonl> --output <каталог.jsonl>`.

Входной файл должен быть в кодировке UTF-8, с одной записью JSON на строку и переводами строк `\n` или `\r\n`. Пустые строки и строки из пробелов пропускаются, но учитываются в нумерации.

Поле	Тип и ограничения
`id`	Строка `^[a-z0-9][a-z0-9-]{1,63}$`, уникальна во всём файле
`name`	Строка, после нормализации от 1 до 200 символов
`category`	Строка от 1 до 64 символов
`daily-Price`	Целое число копеек, не меньше 0
`deposit`	Целое число копеек, не меньше 0
`currency`	Только `"RUB"`
`specs`	Необязательный объект. Значения: строки, числа или `true` / `false`. Если поля нет, в результате `{}`

Других полей быть не должно. При нормализации в `name` и `category` удаляются пробелы по краям, а в `name` любые последовательности пробельных символов заменяются одним пробелом.

Результат импорта - нормализованный JSONL, записи в нём идут в том же порядке, что во входном файле. Целевой файл заменяется, только если весь вход верен. При любой ошибке прежний файл сохраняется байт в байт, а временных файлов не остаётся.

Ошибки данных выводятся в stderr в виде `строка N: поле: причина`. Причину формулируйте для сотрудника. В ошибке о повторе `id` укажите строку, где он встретился впервые. Вход без единой записи считается ошибкой, потому что пустой импорт стёр бы каталог. Выводятся первые 50 ошибок и строка с числом остальных. В конце выводится «Каталог не изменён.»

Пределы импорта - не больше 100 000 позиций и не больше 64 КБ на строку. Превышение предела считается ошибкой данных и не должно ронять процесс. У этого обещания есть граница: `readline` отдаёт строку только целиком, поэтому длину можно проверить, лишь когда строка уже в памяти. Строка в десятки мегабайт без перевода строки при куче 32 МБ уронит процесс раньше проверки. От этого защитил бы предел, который срабатывает раньше, чем строка собрана. Для этого поток разбирают сами, без `readline`, и считают байты по ходу чтения. Предел размера файла по `stat` помогает, только если он меньше памяти процесса. В задании такая защита не требуется.

Коды выхода: 0 означает успех, 1 ошибки данных, 2 неверный вызов, 3 сбой ввода-вывода (нет файла, нет прав, не удалось записать).

Расход памяти не должен зависеть от размера файла. Файл около 80 МБ из строк обычной длины должен импортироваться при `node --max-old-space-size=32`.

Сервер проверки живости запускается командой `node src/http/server.ts`, порт берётся из `PORT` (по умолчанию 3000; при неверном значении сервер не запускается и завершается с кодом 2). На `GET` и `HEAD /health` он отвечает 200 и `{"status":"ok"}`. На запрос к `/health` другим методом он отвечает 405 с `Allow: GET, HEAD`. На остальные пути отвечает 404 и `{"error":"not_found"}`. По SIGTERM сервер останавливается мягко и завершается с кодом 0.

Тесты на `node:test` покрывают как минимум:

- успешный импорт и нормализацию;
- неверный JSON, пропущенное поле, дробную и отрицательную сумму и лишнее поле, причём в каждом случае проверяется и номер строки;
- дубль `id` не в соседних строках;
- сохранение прежнего каталога при ошибке в середине и при пустом входе;
- отсутствие входного файла и невозможность записи, в обоих случаях код 3;
- импорт большого файла при ограниченной куче;
- все три ответа сервера.

Критерии готовности:

- `npm run check` (типы, линтер и тесты) проходит на чистой копии репозитория по инструкции из README.
- Ни один `catch` не глотает ошибку молча. Пустой `catch` и промис без обработки находит `npm run lint`, а `catch`, который ловит всё подряд и подменяет исход, ищет ревью.
- README описывает вызов, коды выхода и пределы.
- Решение не читает входной файл целиком через `readFile` и не копит все записи в массиве.
- В разделе «Помощь ИИ» описания MR указана существенная помощь ассистента или написано, что её не было.

Перед сдачей проверьте себя:

```
npm ci && npm run check
node src/cli/import-catalog.ts --input fixtures/catalog-broken.jsonl \
  --output /tmp/catalog.jsonl; echo "код выхода: $?"
PORT=3000 node src/http/server.ts &
SERVER_PID=$!
URL=http://127.0.0.1:3000/health
for i in $(seq 50); do curl -sf "$URL" >/dev/null && break; sleep 0.2; done
curl -sf "$URL" >/dev/null || echo 'нет ответа за 10 секунд, смотрите вывод сервера'
curl -si "$URL"
kill "$SERVER_PID"
wait "$SERVER_PID"; echo "код: $?"
```

На эталонном решении импорт `catalog-broken.jsonl` печатает ошибки по строкам, затем «Каталог не изменён.» и `код выхода: 1`. Сервер отвечает 200 с телом `{"status":"ok"}`, а после мягкой остановки последняя строка - `код: 0`.

Сервер начинает принимать соединения не сразу после `&`, поэтому запросы идут после цикла ожидания. Цикл ждёт ответа до 10 секунд. Если сервер так и не ответил, следующая строка сообщает об этом. Дальше проверять нечего: причину ищите в выводе сервера.

Доверенная проверка запускает свои тесты на своих данных. Она сверяет номер строки, имя поля и код выхода, а текст причины остаётся на ваше усмотрение.

Типовые ошибки и диагностика

Симптом	Причина	Как найти	Как исправить
`JavaScript heap out of memory` на большом файле	Файл читается целиком, записи копятся в массиве или игнорируется `false` от `write()`	Запустить с `--max-old-space-size=32` на файле 80 МБ	Читать построчно и использовать `pipeline`
Тест: «Promise resolution is still pending but the event loop has already resolved»	Код ждёт промис, который ничто не разрешит, а других дел в цикле нет	Посмотреть логи вокруг каждого `await` и найти последний сработавший	Убрать ожидание, которое не может завершиться (см. строку ниже)
Импорт зависает на закрытии файла	Поток создан из `FileHandle` с `autoClose: false`, затем вызван `await handle.close()`	`close()` у `FileHandle` ждёт завершения операций, а поток не отпускает дескриптор	Отдать дескриптор потоку: `createWriteStream(path, { flags: 'wx', flush: true })`
Номера строк сдвинуты	Пустые строки не считаются или `\r\n` даёт лишние строки	Сравнить с номерами в редакторе на файле из Windows	Считать каждую строку и задать `crlfDelay: Infinity`
`ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX`	В коде есть `enum`, `namespace` или свойства в параметрах конструктора	Текст ошибки указывает на строку	Использовать объединение литералов или `as const`, включить `erasableSyntaxOnly` в tsconfig
`does not provide an export named 'Item'`	Тип импортирован без `type`	Имя в ошибке принадлежит типу	Писать `import type { Item }` и включить `verbatimModuleSyntax`
`ERR_MODULE_NOT_FOUND` для своего файла	Импорт без расширения	В пути из ошибки нет `.ts`	Писать `import './item.ts'`

Симптом	Причина	Как найти	Как исправить
Вывод обрезан при передаче в `head`, `wc` или файл	`process.exit()` вызван до завершения записи в канал	Подключить медленный читатель: `| (sleep 1; wc -l)`	Использовать `process.exitCode`
`EXDEV: cross-device link not permitted`	Временный файл лежит в `os.tmpdir()`, а целевой на другом разделе	Сравнить точки монтирования обоих путей	Создавать временный файл в папке целевого
Цена расходится на копейку	Дробные суммы и арифметика с плавающей точкой	Найти `parseFloat`, `/ 100`, `Math.round` в расчётах	Хранить целые копейки, дробные суммы отклонять
Процесс падает с кодом 1 и стеком отклонённого промиса	Промис без `await` и `.catch` или async-обработчик у `process.on`, `createServer`	Запустить `npm run lint`: правила `no-floating-promises`, `no-misused-promises`	Поставить `await` или `.catch` с обработкой; `void` допустим, только если отказа быть не может
Процесс падает с необработанным событием `'error'`	У потока нет обработчика ошибок	Стек указывает на `Readable` или `Writable`	Использовать `pipeline` вместо `pipe` и ждать через `once`
Остановка занимает секунды после SIGTERM	Keep-alive соединения остаются открытыми после начатого запроса	Замерить время от сигнала до выхода при активном клиенте	Вызывать `closeIdleConnections()` после каждого ответа во время остановки
`EADDRINUSE`	Порт занят прошлым запуском	Выполнить `lsof -i :3000`	Остановить прошлый процесс; в тестах брать `PORT=0`
Сервер запустился, но не отвечает на ожидаемом порту	`PORT=''` превратился в 0, и система выдала случайный порт	Посмотреть в логе фактический порт из `server.address()`	Проверять строку `PORT` регулярным выражением

Самопроверка

1. Почему сервер перестаёт отвечать на `/health`, пока выполняется синхронный пересчёт цен, хотя чтение файлов выполняет пул потоков libuv?

2. В каком порядке выполнятся `setTimeout(0)`, `setImmediate`, `process.nextTick` и `Promise.then`, заведённые в колбэке `fs.readFile`? Какую часть порядка нельзя считать гарантированной в главном ES-модуле?
3. Что вернёт `write()` при заполненном буфере, что произойдёт, если продолжить писать, и как `pipeline` решает эту проблему?
4. Зачем нужен `crlfDelay: Infinity` при чтении выгрузки?
5. Почему `const item = JSON.parse(line) as CatalogItem` ничего не проверяет, и как это исправить?
6. Почему дробную сумму в выгрузке нужно отклонить, а не округлить?
7. Почему временный файл создаётся в той же папке, что и целевой, и что даёт `flush: true`?
8. Чем отличаются исходы с кодами 1 и 3, и почему отсутствующий входной файл нельзя считать пустым каталогом?
9. Почему вместо `process.exit(1)` после вывода отчёта лучше `process.exitCode = 1`?
10. Что делает `server.close()` с keep-alive соединениями, и зачем примеру `closeIdleConnections()` после каждого ответа?
11. Что произойдёт, если в коде, который запускается через type stripping, объявить `enum`? Как узнать об этом до запуска?
12. Линтер сообщает `no-floating-promises` на строке `cache.refresh()` в обработчике запроса: кэш обновляется в фоне, ответ его не ждёт. Чем плохо поставить `void`, и как исправить?

Ответы - в приложении.

Вопросы собеседования

1. Опишите фазы цикла событий Node.js. В какой фазе выполняются колбэки `setImmediate`, а где колбэки промисов?
2. Как найти в работающем сервисе код, который блокирует цикл событий?
3. Что такое обратное давление в потоках и как оно проявляется в продакшене?
4. Чем `pipeline` лучше цепочки `.pipe()`?
5. Как безопасно заменить файл, который читают другие процессы? Какие у этого способа границы?
6. Как хранить денежные суммы в JavaScript и почему не в `number` с дробной частью?
7. Что должно происходить в Node.js-сервисе между получением SIGTERM и выходом?
8. Чем проверка живости отличается от проверки готовности?

Что читать дальше

- Node.js: The Node.js Event Loop - https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick

- Node.js: Backpressuring in Streams - https://nodejs.org/en/learn/modules/backpressuring-in-streams
- Node.js API: Streams, раздел `stream.pipeline` - https://nodejs.org/api/stream.html
- Node.js API: Modules: TypeScript - https://nodejs.org/api/typescript.html
- Node.js API: process, разделы `process.exit` и «A note on process I/O» - https://nodejs.org/api/process.html
- Node.js API: File system, `fsPromises.writeFile` и опция `flush` - https://nodejs.org/api/fs.html
- Node.js API: HTTP, `server.close` и `server.closeIdleConnections` - https://nodejs.org/api/http.html
- Node.js API: Worker threads - https://nodejs.org/api/worker_threads.html
- POSIX: функция `rename` - https://pubs.opengroup.org/onlinepubs/9799919799/functions/rename.html
- RFC 9110, раздел 15.5.6 «405 Method Not Allowed» - https://www.rfc-editor.org/rfc/rfc9110#section-15.5.6
- Документация zod 4 - https://zod.dev
- typescript-eslint: правила `no-floating-promises` и `no-misused-promises` - https://typescript-eslint.io/rules/no-floating-promises/, https://typescript-eslint.io/rules/no-misused-promises/
- Node.js CLI: `--unhandled-rejections` - https://nodejs.org/api/cli.html#--unhandled-rejectionsmode

Глоссарий

Термин	Значение
Цикл событий (event loop)	Механизм libuv, который по фазам выполняет готовые колбэки в единственном JS-потоке
Микрозадача (microtask)	Колбэк промиса или `queueMicrotask`. Очередь микрозадач выполняется полностью после каждой операции
Блокировка цикла событий	Синхронная работа, во время которой процесс не обслуживает другие события
Поток (stream)	Объект, который отдаёт или принимает данные порциями
Обратное давление (backpressure)	Сигнал от медленного приёмника источнику: остановиться, пока буфер не освободится
`highWaterMark`	Порог буфера потока, после которого `write()` возвращает `false`
JSONL	Формат, в котором на каждой строке одна запись JSON
Валидация во время выполнения (runtime validation)	Проверка данных при работе программы, а не при компиляции

Термин	Значение
Минимальные денежные единицы	Копейки для рубля. Суммы хранятся целым числом таких единиц
Атомарная замена	Замена файла, при которой читатель видит либо старую, либо новую версию целиком
Код выхода (exit code)	Число, которым процесс сообщает вызывающей стороне, чем всё закончилось
Проверка живости (liveness check)	Ответ на вопрос «процесс жив и обслуживает события?»
Мягкая остановка (graceful shutdown)	Остановка, при которой начатые запросы завершаются, а новые не принимаются
Type stripping	Запуск TypeScript в Node.js путём удаления аннотаций типов без компиляции
Линтер (linter)	Программа, которая читает код, не запуская его, и находит места, где поведение вероятно ошибочно
Висящий промис (floating promise)	Промис, результат которого никто не ждёт и отказ которого никто не обрабатывает
Необработанный отказ промиса (unhandled rejection)	Отклонённый промис без обработчика отказа. По умолчанию процесс завершается с кодом 1

Журнал изменений

Версия	Что изменилось
0.1.0	Первая версия
0.1.1	Таблица типовых ошибок помещается в область печати: PDF больше не уменьшается целиком
0.1.2	Титул: код примеров лежит в папке examples/week-01 стартового репозитория; K19 и K24 добавлены в компетенции недели; подраздел «Как работать с ИИ-ассистентом в курсе» и критерий о разделе «Помощь ИИ» в MR; граница предела 64 КБ описана точнее; заглушки и публичные тесты приходят с MR «Материалы недели 1»; упражнения входят в 7 часов реализации; убрано обещание про worker_threads; проверка себя ждёт готовности сервера
0.2.0	Раздел «Линтер»: как читать ответ `no-floating-promises`, три исправления (`await`, `.catch` и `void`), правила `no-misused-promises`, `no-empty` и `preserve-caught-error`, когда допустим `eslint-disable`; `void` перед микрозадачей в примере порядка колбэков; `npm run check` (типы, линтер и тесты) в критериях и проверке себя; типовая ошибка о падении с кодом 1 от отклонённого промиса; вопрос самопроверки 12; термины «висящий промис» и «необработанный отказ промиса»; ссылки на правила typescript-eslint и `--unhandled-rejections`
0.2.1	Язык: текст переписан простым русским, без длинных тире, выделений жирным и рубленых фраз. Содержание, код и задания не менялись

Версия	Что изменилось
0.2.2	Язык: в схемах короткий дефис вместо длинного тире
0.2.3	Основное задание: заглушки `src/cli/import-catalog.ts` и `src/http/server.ts` лежат в проекте с недели 0, MR «Материалы недели 1» добавляет только публичные тесты
0.2.4	Node.js 22.22 или новее; проверка себя ждёт сервер не дольше 10 секунд и сообщает, если он не ответил
0.2.5	Проверка себя без комментария `#`, который zsh передаёт команде как аргументы: что сценарий выводит на эталоне, описано под ним
0.2.6	Язык: подзаголовок и результаты обучения переписаны простыми словами, без перечней через запятую и двоеточие. Содержание и задания не менялись
0.2.7	Язык: теория, задание, самопроверка и ответы вычитаны целиком, перечни и подписи с двоеточием стали обычными фразами, убраны кальки и канцелярит. Исправлено по смыслу: У программы импорта четыре исхода, а не три, как было написано над таблицей кодов выхода.

Приложение. Ответы на самопроверку

1. Пул потоков выполняет только сам ввод-вывод. Колбэки с результатами, обработчики запросов и ваш расчёт выполняются в одном JS-потоке. Пока синхронный цикл пересчёта цен не закончился, цикл событий не может перейти к следующему колбэку. Запрос к `/health` уже пришёл и ждёт, пока поток освободится.

2. Внутри колбэка ввода-вывода порядок такой: `nextTick`, затем `promise`, затем `setImmediate` (фаза check идёт сразу после poll), затем `setTimeout`. В главном ES-модуле не гарантированы две вещи. Порядок `setTimeout(0)` и `setImmediate` зависит от загрузки машины. Порядок `nextTick` и промиса зависит от версии Node.js и от наличия top-level `await`: в 22 и 24 первым идёт промис, а в 26.12 без top-level `await` первым идёт `nextTick`.

3. `write()` вернёт `false`, потому что буфер выше `highWaterMark`. Данные всё равно будут приняты и лягут в память. Если продолжать писать, буфер вырастет до размера всех данных, и на большом файле процесс упадёт по памяти. `pipeline` перестаёт читать источник, пока приёмник не сообщит `drain`. При ошибке любого звена он закрывает все потоки и отклоняет промис.

4. В файлах из Windows строки разделены `\r\n`. По умолчанию readline считает `\r` и `\n` одним переводом строки, только если они пришли с промежутком меньше 100 мс. На границе порций чтения пауза возможна, и тогда появится лишняя пустая строка, а нумерация сдвинется. С `Infinity` пара `\r\n` всегда считается одним переводом строки.

5. `as` - это указание компилятору, а не проверка. После стирания типов остаётся `const item = JSON.parse(line)`, и объект без `dailyPrice` или с ценой-строкой пройдёт дальше. Результат нужно принять как `unknown`, проверить схемой (`safeParse`) и использовать только проверенные данные, а тип вывести из схемы.

6. Копейка - минимальная единица, поэтому дробь копейки означает ошибку в источнике: рубли перепутаны с копейками или ячейку исправили вручную. Округление спрячет ошибку и даст неверную цену без следа. Отказ с номером строки покажет ошибку тому, кто может исправить источник.

7. `rename` атомарно заменяет файл только в пределах одной файловой системы. Если временный файл лежит на другом разделе, замена упадёт с `EXDEV`, и атомарности не будет. `flush: true` вызывает `fsync` перед закрытием, и данные дойдут до диска раньше, чем новое имя начнёт на них указывать. Без этого после сбоя питания можно получить целевой файл нулевой длины.

8. Код 1 означает, что файл прочитан без сбоев, но данные в нём неверны, и сотрудник исправляет их по номерам строк. Код 3 означает, что программа не смогла прочитать или записать файл, и исправлять нужно окружение: путь, права, место на диске. Если считать отсутствующий файл пустым каталогом, импорт либо сотрёт рабочий каталог, либо сообщит об ошибке данных, которой нет.

9. При выводе в канал запись в stdout на Linux и macOS асинхронна. `process.exit()` завершает процесс сразу, и конец отчёта теряется. `process.exitCode` только задаёт код, а процесс завершается, когда всё дописано.

10. Начиная с Node.js 19 `server.close()` перестаёт принимать соединения и закрывает те, что простаивают в момент вызова. Соединение запроса, который ещё выполнялся, после ответа становится простаивающим, но его уже никто не закрывает: оно живёт до `keepAliveTimeout` или пока клиент не закроет его сам. Если во время остановки вызывать `closeIdleConnections()` на завершении каждого ответа, такие соединения закрываются сразу.

11. Node.js откажется выполнять файл и выдаст ошибку `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX`: `enum` порождает JavaScript-код, а type stripping только удаляет типы. Узнать об этом до запуска позволяет `tsc` с `erasableSyntaxOnly` в `tsconfig.json`.

12. `void` сообщит, что отказа быть не может, а он возможен: Redis или база могут не ответить. Отказ останется необработанным и завершит процесс с кодом 1 посреди работы. Ждать обновления ответ действительно не должен, поэтому нужен `.catch` с обработкой: записать в журнал, что кэш не обновлён, и посчитать это в метрике. Если обновление обязано закончиться до ответа, нужен `await`.

© Евгений Надточеев. Все права защищены. Версия 0.2.7. Шрифты Inter и JetBrains Mono — SIL Open Font License 1.1.