UNIT 4 – Microservices and Error HandlingAssignment 4

Q1. Compare Express, Hapi, and Fastify as Node.js frameworks for microservices development. Discuss their performance, scalability, and ease of use.

Answer:

Express

- **Overview:** Minimal, unopinionated, battle-tested web framework. Huge ecosystem and middleware availability.
- **Performance:** Good; not the fastest but very adequate for many services. Single-threaded Node event loop behavior applies.
- **Scalability:** Scales horizontally easily (stateless services behind load balancer). Many community patterns for clustering, containers, and microservices.
- **Ease of use:** Very easy to learn; small API surface; lots of tutorials and middleware (routing, authentication, validation).
- **When to choose:** When you want simplicity, maximum ecosystem support, quick prototypes, or when team familiarity matters.

Hapi

- **Overview:** More opinionated than Express; designed around configuration, plugins, and strong input validation and policy-driven design.
- **Performance:** Comparable to Express; slightly heavier by default because of richer feature set.
- **Scalability:** Good — plugin architecture encourages modular services; suits larger teams needing conventions and built-in patterns.
- **Ease of use:** Slightly steeper learning curve because of conventions and configuration patterns, but excellent for consistent, maintainable service APIs.
- **When to choose:** When you want built-in validation/auth/security features and a structured plugin-driven architecture.

Fastify

- **Overview:** Modern, high-performance web framework focused on low overhead and JSON throughput.
- **Performance:** Very fast — optimized for throughput and low overhead, uses schema-based serialization (AJV) for fast JSON handling.
- **Scalability:** Excellent for high-traffic microservices; low memory footprint helps scale horizontally in containerized environments.
- **Ease of use:** API is familiar (similar routing style to Express) but encourages schema-based validation which requires some upfront work.
- **When to choose:** When performance/throughput matters (APIs with heavy JSON payloads), or when you want minimal overhead at scale.

Comparison summary (practical):

- **Performance winner:** Fastify (for raw throughput).
- **Ecosystem/maturity winner:** Express.
- **Best for structured large teams:** Hapi (for convention and security features).
- **Overall recommendation:** Use Express for general-purpose services and developer velocity; choose Fastify for high-throughput endpoints; choose Hapi where policy/validation and a strict plugin model matter.

Q2. Explain the role of error monitoring and logging in Node.js applications. How can tools like Sentry or Loggly assist in troubleshooting and improving application reliability?

Answer:
Role of error monitoring & logging

- **Detect & triage:** Capture runtime errors, stack traces, and contextual data so developers can reproduce and fix issues quickly.
- **Operational visibility:** Track application health metrics (error rates, request latencies), which helps spot regressions and performance degradation.

- **Postmortem & debugging:** Persistent logs and error traces provide the history needed for root-cause analysis.

- **Alerting & SLA adherence:** Trigger alerts when error thresholds or SLOs are violated.

How tools help

- **Sentry (error monitoring):**
    - Captures exceptions with stack traces and context (user, request, breadcrumbs).
    - Groups similar errors automatically (reducing noise).
    - Integrates with issue trackers and Slack for alerts.
    - Releases & performance monitoring show regressions introduced by deployments.
    - Good for real-time error visibility and developer workflow integration.
- **Loggly / Logstash / ELK / Datadog Logs (log aggregation):**
    - Centralize logs from many instances/services and make them searchable.
    - Provide structured logging (JSON), dashboards, and long-term retention.
    - Support log-based alerting and correlation between metrics and logs.

Practical integration tips

- **Structured logs:** Use JSON logs including request id, user id, service, and correlation id. (Libraries: `pino`, `winston`.)

- **Error enrichment:** Attach contextual info (HTTP headers, request body, environment) but scrub sensitive data (PII, credentials).

- **Correlation:** Add a correlation/request ID to incoming requests and propagate it to logs and errors to trace requests end-to-end.

- **Sampling & rate limiting:** Avoid log overload by sampling verbose logs and rate-limiting repetitive errors.

- **Alerting runbooks:** Configure alerts with meaningful thresholds and include runbooks describing steps to investigate.

Benefit to reliability: Faster detection + actionable context reduces mean time to resolution (MTTR), improves uptime, and helps teams prioritize fixes based on impact.

Q3. What is the CI/CD pipeline, and why is it critical in the context of Node.js microservices? Describe how automated build and deployment pipelines can be configured using tools like Jenkins or CircleCI.

Answer:
CI/CD pipeline (definition & purpose)

- **Continuous Integration (CI):** Automates build, linting, tests for each code change (PR/commit) to detect regressions early.
- **Continuous Delivery / Deployment (CD):** Automates packaging and deployment to environments (staging/production), often with gating (manual approval) or automated promotion (continuous deployment).

Why critical for Node.js microservices

- **Many small services:** Microservices increase the number of deployable units; automation ensures consistent builds and avoids human error.
- **Frequent releases:** Enables rapid iteration and safe rollouts.
- **Consistency:** Guarantees the same build/test/deploy steps across teams and services.
- **Testing across services:** Automates contract/integration tests to catch breaking changes early.

Typical CI/CD stages

1. **Source**: commit triggers pipeline (GitHub/GitLab/Bitbucket webhook).
2. **Build**: install deps (`npm ci`), transpile TypeScript/babel.
3. **Static checks**: linting (`eslint`), type checks.

4. **Unit tests**: run Jest/Mocha with coverage.
5. **Integration tests**: service contract tests or test against test DB.
6. **Package**: build Docker image and push to registry.
7. **Deploy**: deploy image to staging/production (Kubernetes, ECS, etc.).
8. **Post-deploy checks**: smoke tests, health checks; roll back on failure.

Example with Jenkins

- **Jenkinsfile** (pipeline-as-code):
    - `pipeline { agent any; stages { stage('Build') { steps { sh 'npm ci' } } stage('Test') { steps { sh 'npm test' } } stage('Docker') { steps { sh 'docker build -t myservice:$BUILD_NUMBER .' } } stage('Push') { steps { withCredentials(...) { sh 'docker push ...' } } } } }`
- Jenkins executes on each PR/commit and can trigger deployment jobs or integrate with Kubernetes.

Example with CircleCI

- `.circleci/config.yml` defines workflows:
    - jobs: `build`, `test`, `docker/push`, `deploy`
    - workflows: run `build` → `test` → `docker/push` → `deploy` when `test` passes.
- CircleCI provides convenience for containerized builds and caching `node_modules`.

Best practices

- Use immutable artifacts (Docker images with tags).
- Run tests in containers matching prod environment.
- Use separate pipelines for infra-as-code changes.
- Canary or blue/green deployments for safe rollouts.

- Pipeline secrets managed by the CI tool's secret store or external vault.

Q4. Describe the key design patterns used in microservices development. Provide an example of how these patterns can be implemented in a Node.js application to enable communication and coordination between microservices.

Answer:
Key design patterns

1. **API Gateway**
    - Single entry point for clients; handles routing, authentication, rate-limiting, and API composition.
    - Implementation: Nginx, Kong, or a Node service (e.g., Express/Fastify).
2. **Service Discovery**
    - Dynamic discovery of service instances (important in autoscaling).
    - Implementation: Consul, etcd, Kubernetes DNS + Service resources.
3. **Circuit Breaker**
    - Prevents repeated calls to failing services; fallback logic and bulkhead patterns.
    - Implementation: `opossum` (Node circuit breaker library).
4. **Bulkhead / Isolation**
    - Isolate resources per service/operation (threads, connection pools).
5. **Event-driven / Message Bus**
    - Decouple services via asynchronous messaging (Kafka, RabbitMQ). Useful for eventual consistency and fan-out.
6. **Saga Pattern**

    - Manage distributed transactions via compensating actions for long-running multi-service workflows.

7. **Strangler/Anti-corruption Layer**
    - Incrementally replace monoliths with microservices; use adapters to translate old interfaces.

Example Node.js implementation (small scenario)
Scenario: **Order service** needs to create orders and notify **Inventory** and **Billing** services.

- **Synchronous approach (REST + API Gateway)**:
    - Client → API Gateway → Order Service (Express/Fastify).
    - Order Service calls Inventory Service and Billing Service via REST.

Use **circuit breaker** when calling external services:

```
 const CircuitBreaker = require('opossum');
const inventoryCall = () => fetch('http://inventory/lock', {...});
const breaker = new CircuitBreaker(inventoryCall, { timeout: 3000,
errorThresholdPercentage: 50 });
```

    -

- **Asynchronous (recommended for decoupling)**:
    - Order Service publishes `order.created` event to message broker (Kafka/RabbitMQ).
    - Inventory consumes `order.created` and reserves stock; Billing consumes and charges payment.
    - If Billing fails, a compensating event `order.payment_failed` can trigger Inventory to release stock (Saga pattern).

Implementation sketch (publisher):

```
 // using amqplib (RabbitMQ)
const amqp = require('amqplib');
const channel = await conn.createChannel();
channel.assertExchange('orders', 'topic');
channel.publish('orders', 'order.created', Buffer.from(JSON.stringify(order)));
```

    -

Why patterns help: They provide resilience (circuit breakers), decoupling and scalability (message bus), and operational safety (API gateway, service discovery).

Q5. Explain the process of containerizing a Node.js microservice using Docker. Discuss how Kubernetes orchestrates these containers, and outline the key steps involved in deploying and managing Node.js microservices with Kubernetes.

Answer:
Containerizing with Docker (basic steps)

Write a Dockerfile

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

1.

Build image

```
docker build -t myservice:1.0.0 .
```

2.

Test locally

```
docker run -p 3000:3000 myservice:1.0.0
```

3.

Push to registry

```
docker tag myservice registry.example.com/myservice:1.0.0
docker push registry.example.com/myservice:1.0.0
```

4.

Kubernetes orchestration (what it provides)

- **Scheduling & scaling:** Places containers (pods) on nodes and supports auto-scaling (Horizontal Pod Autoscaler).
- **Service discovery & load balancing:** `Service` objects provide stable DNS and load balancing between pods.
- **Self-healing:** Restarts failed pods and replaces unhealthy pods using liveness/readiness probes.
- **Rolling updates & rollbacks:** `Deployment` can roll out new versions with minimal downtime.
- **Config & secrets:** `ConfigMap` and `Secret` for environment/configuration without baking into images.
- **Namespaces & RBAC:** Multi-tenant isolation and fine-grained access control.

Key Kubernetes resources for a Node microservice

1. **Deployment** — defines desired pod replica count and container spec (image, env).
2. **Service** — exposes the pods to internal cluster traffic (ClusterIP) or externally (LoadBalancer/NodePort).
3. **Ingress** — optional layer for HTTP(S) routing and TLS termination.
4. **ConfigMap/Secret** — for configuration and sensitive values.
5. **HorizontalPodAutoscaler (HPA)** — scale pods based on CPU, memory, or custom metrics.
6. **PodDisruptionBudget** — control voluntary disruptions during upgrades.

Typical deploy steps

1. CI builds Docker image and pushes to registry.
2. CI updates Kubernetes manifests (image tag) or creates a new `Deployment` via `kubectl` or GitOps.
3. Kubernetes performs rolling update; readiness probes ensure traffic only goes to healthy pods.
4. Monitor with metrics/alerts; scale with HPA or custom metrics.

Best practices

- Keep images small (multi-stage builds, alpine base).
- Use readiness & liveness probes to avoid routing to bad pods.
- Externalize config via ConfigMaps/Secrets.
- Use resource requests/limits to help the scheduler.
- Use probes and graceful shutdowns (SIGTERM handling) so pods can finish work before termination.

Q6. Discuss the best practices for developing microservices in Node.js, focusing on error monitoring, logging, and CI/CD. Use a real-world case study to illustrate how these practices improve scalability, reliability, and efficiency in production environments.

Answer:
Best practices (concise)

1. **Observability**
    - Structured logs (JSON) with correlation IDs.
    - Distributed tracing (opentelemetry, Jaeger).
    - Error monitoring (Sentry) and metrics (Prometheus + Grafana).
2. **Resilience**
    - Circuit breakers, timeouts, retries with exponential backoff, bulkheads.
    - Graceful shutdown and resource cleanup.
3. **Security & validation**
    - Input validation, rate limiting, secure headers (helmet), least privilege for secrets.
4. **Stateless services**

    - Keep business state in external stores (Redis, DB) to allow horizontal scaling.

5. **CI/CD**
    - Automated tests, build, Docker image creation, automated deployment with canary/blue-green.
    - Use feature flags for controlled rollouts.
6. **Contracts & versioning**
    - API contracts (OpenAPI/Swagger), consumer-driven contract tests.
7. **Resource control**
    - Set CPU/memory requests and limits; use autoscaling.

Real-world case study (concise)
Company: *Hypothetical e-commerce platform*

- **Problem:** Frequent outages and slow recovery when catalog service experienced high load; deployments produced regressions occasionally.
- **Actions implemented:**
    1. **Logging & error monitoring:** integrated structured logging (`pino`) and Sentry. Added correlation IDs propagated across services.
    2. **Tracing & metrics:** installed OpenTelemetry + Prometheus + Grafana; identified slow endpoints and DB queries.
    3. **CI/CD improvement:** moved to Docker-based pipelines with Jenkins; added automated unit & integration tests and stage environment with automated smoke tests; enabled blue/green deployments and automated rollbacks on failed health checks.
    4. **Resilience:** added circuit breakers around third-party payment gateway calls; introduced message queue (RabbitMQ) for order processing to decouple from synchronous payment latency.
    5. **Autoscaling & kube hygiene:** deployed on Kubernetes with HPA, readiness/liveness probes, and resource limits.
- **Results:** Reduced mean time to detection (MTTD) via alerts, reduced MTTR because errors had complete stack/context in Sentry, eliminated cascading failures with circuit breakers and queueing, and improved capacity handling with HPA and optimized DB queries. Deployment failure rate fell and releases became more

frequent and safer.

Conclusion: Observability + good CI/CD + resilience primitives together improve reliability, scale, and developer velocity.

Q7. Discuss the core principles of microservices architecture. Compare the monolithic and microservices approaches in terms of scalability, maintainability, and fault tolerance.

Answer:
Core principles

1. **Single Responsibility / Bounded Context:** Each service owns a distinct business capability.
2. **Independently deployable:** Services can be deployed and scaled independently.
3. **Decentralized data management:** Each service manages its own data store; avoid shared monolithic DBs.
4. **Explicit contracts:** Services communicate via well-defined APIs or messages.
5. **Resilience & observability:** Design for failure with monitoring, tracing, retries, and fallbacks.
6. **Automation:** CI/CD and infra automation to manage many services.

Comparison: Monolith vs Microservices

- **Scalability**
    - *Monolith:* Vertical scaling or replicate entire app—inefficient when only one part needs more resources.
    - *Microservices:* Horizontal scaling per service; more granular and cost-efficient.
- **Maintainability**
    - *Monolith:* Single codebase simpler initially; can become large and hard to change as app grows (tight coupling).

    - *Microservices:* Smaller codebases are easier to reason about, but require more operational governance; complexity moves to deployment/coordination.

- **Fault tolerance**
    - *Monolith:* Single process failure can take entire app down unless architected with redundancy.
    - *Microservices:* Faults can be isolated to a service; patterns like circuit breakers and retries reduce blast radius—but distributed failures can be harder to diagnose.

- **Deployment & velocity**
    - *Monolith:* Simple CI/CD early on; slows down as app grows due to risk of full release.
    - *Microservices:* Enables independent releases and faster team velocity, but requires strong automation and testing discipline.

- **When to choose**
    - Start with a monolith if product is small and team is small (simpler to develop and operate). Move to microservices when scale, team size, or domain complexity justify the additional operational overhead.

Q8. Design a microservices-based architecture using Node.js, detailing how different services interact. Discuss the role of RESTful APIs, message brokers, and service discovery in microservices communication.

Answer:
Example architecture (e-commerce): Services: **API Gateway**, **Auth Service**, **Product Service**, **Catalog Service**, **Order Service**, **Inventory Service**, **Payment Service**, **Notification Service**, **Search Service**.

How they interact

- **Client interaction**
    - All client requests go to **API Gateway**: handles auth validation, rate-limiting, TLS, and routes to services.

- **Synchronous interactions (REST)**
    - For read-mostly queries: Gateway → Product/Catalog/Order via RESTful APIs (HTTP/JSON).
    - REST is used for request/response operations where the caller requires immediate results (e.g., view product, place order initial validation).
- **Asynchronous interactions (message broker)**
    - When an order is placed, **Order Service** writes order and publishes `order.created` event to message broker (Kafka/RabbitMQ).
    - **Inventory Service** subscribes to `order.created` and reserves stock.
    - **Payment Service** subscribes and attempts charge; publishes `payment.succeeded` or `payment.failed`.
    - **Notification Service** subscribes to `payment.*` or `order.*` for emails/SMS.
    - Asynchronous messages decouple services and improve resilience and scalability.
- **Service discovery**
    - In dynamic environments (Kubernetes or autoscaling hosts), use built-in DNS-based discovery (K8s Services) or Consul/etcd for discovering service instances and load balancing.
    - Clients or API Gateway resolve service endpoints via service name rather than hard-coded host:port.
- **Data management**
    - Each service owns its data store (e.g., Orders in PostgreSQL, Inventory in Redis/Postgres, Search in Elasticsearch).
    - Use event-driven or CDC (change-data-capture) to sync data where necessary (e.g., update search index when product changes).
- **Cross-cutting concerns**
    - **Authentication/Authorization:** Auth service issues JWTs; API Gateway validates tokens and attaches identity to requests.
    - **Tracing & correlation:** Propagate correlation ID through headers for tracing.

    - **Monitoring & logging:** Centralized metrics and logs (Prometheus + Grafana + ELK/Splunk) for observability.

- **Resilience & patterns**
    - **Timeouts, retries, circuit breakers** for external calls.
    - **Saga pattern** or compensation for multi-step transactions (e.g., if payment fails after inventory reserved, emit an event to release inventory).

Why this mix works

- **REST**: simple, synchronous, good for request/response flows and public APIs.
- **Message broker**: asynchronous decoupling, suitable for workflows, fan-out, and eventual consistency.
- **Service discovery**: enables dynamic scaling and deployment without manual config changes.

If you'd like, I can:

- Convert these into **8 short one-page answers** tuned for 10–15 mark exams,
- Produce **sample Kubernetes manifests + Dockerfile** for the sample microservice, or
- Create a **diagram (architecture)** and a `README` describing how to run a small demo system locally with Docker Compose.

Which of those helps you next?

});

test("divide(): divide by zero", () => {
  expect(() => divide(4, 0)).toThrow("Divide by zero");
});

Run tests:

npm test