REPUBLIKA E SHQIPERISË
UNIVERSITETI POLITEKNIK I TIRANËS
FAKULTETI I TEKNOLOGJISË SË INFORMACIONIT
DEPARTAMENTI I INXHINIERISË INFORMATIKE

DETYRË KURSI

Lënda: Programim në Sisteme të Shpërndara

Tema: Designing Microservice-Based Applications

Dega: Inxhinieri Informatike

Grupi: III-B

Punoi: Kejdi Shahu
Piro Gjikdhima

Pranoi: MSc Megi Tartari

VITI AKADEMIK: 2024 - 2025

Abstrakt

Ky dokument paraqet një analizë të detajuar të projektimit dhe zhvillimit të aplikacioneve të bazuara në mikroshërbime, duke përfshirë praktikat më të mira, sfidat dhe zgjidhjet e rekomanduara. Materialet e përfshira mbulojnë aspekte të rëndësishme si *Containerization, Service Discovery, Fault Tolerance* dhe modelet e komunikimit, duke synuar të shërbejnë si një udhëzues gjithëpërfshirës për zhvilluesit dhe arkitektët e sistemeve.

Përmbajtja

1. Hyrje

Në zhvillimin e sistemeve moderne, kërkesat për shkallëzueshmëri, fleksibilitet dhe mirëmbajtje efikase kanë sjellë nevojën për arkitektura të reja softuerike. Një nga qasjet më të suksesshme për ndërtimin e sistemeve komplekse është **arkitektura me mikroshërbime**, e cila lejon ndarjen e funksionaliteteve në shërbime të vogla dhe të pavarura.
Kjo detyrë kursi ka për qëllim të analizojë konceptin e mikroshërbimeve, duke u fokusuar në **parimet e projektimit, metodat e komunikimit, strategjitë e menaxhimit dhe sigurisë**, si dhe praktikat më të mira për monitorimin dhe vendosjen e tyre. Përmes kësaj analize, do të eksplorohen përfitimet dhe sfidat që sjell kjo arkitekturë, si dhe teknologjitë kryesore që përdoren për implementimin dhe mirëmbajtjen e saj.

2. Kërkesat e Punës

Arkitektura e Mikroshërbimeve

- Të shpjegohet **koncepti i mikroshërbimeve**, duke përfshirë përkufizimin dhe karakteristikat kryesore.
- Të krahasohet arkitektura **monolitike** me arkitekturën **me mikroshërbime**.
- Të analizohen **përfitimet dhe sfidat** që vijnë me përdorimin e mikroshërbimeve.

Ndërtimi dhe Deployment i Mikroshërbimeve

- Të shpjegohet **koncepti i konteinerizimit**, duke theksuar rolin e **Docker** në izolimin dhe menaxhimin e mikroshërbimeve.
- Të paraqitet **roli i Kubernetes** si një platformë për orkestrimin dhe menaxhimin e mikroshërbimeve në shkallë të gjerë.
- Të diskutohet **shërbimi i zbulimit dhe orkestrimi (Service Discovery & Orchestration)** në një mjedis me shumë mikroshërbime.
- Të shpjegohet **përdorimi i CI/CD për mikroshërbime**, duke përfshirë avantazhet e automatizimit në shpërndarjen e kodit.

Komunikimi Ndërmjet Mikroshërbimeve

- Të krahasohen **metodat e komunikimit sinkron (REST, gRPC)** dhe **asinkron (Kafka, RabbitMQ)** në mikroshërbime.

- Të përshkruhet roli i **API Gateway** në menaxhimin e kërkesave nga klientët dhe në sigurimin e një pike të centralizuar për shërbimet.
- Të shpjegohet përdorimi i **Service Mesh (Istio, Linkerd)** për menaxhimin e komunikimit dhe sigurisë ndërmjet mikroshërbimeve.

Rezistenca dhe Toleranca ndaj Gabimeve

- Të shpjegohet **mekanizmi Circuit Breaker (Resilience4j, Hystrix)** dhe si ndihmon në reduktimin e impaktit të dështimeve.
- Të diskutohen teknikat e **riprocesimit të kërkesave (Retries), kohëzgjatjes (Timeouts) dhe izolimit të burimeve (Bulkheads)**.
- Të përshkruhen strategjitë për **balancimin e ngarkesës dhe shkallëzimin e shërbimeve**, duke përfshirë autoscaling dhe load balancing.

Siguria në Mikroshërbime

- Të shpjegohet **autentifikimi dhe autorizimi (JWT, OAuth2)** në një sistem të bazuar në mikroshërbime.
- Të diskutohen teknikat për **sigurinë e API-ve dhe kufizimin e shpejtësisë (Rate Limiting)** për mbrojtje nga sulmet e jashtme.
- Të paraqitet rëndësia e **sigurisë në komunikim përmes TLS** për të garantuar enkriptimin e të dhënave ndërmjet shërbimeve.

Monitorimi, Regjistrimi dhe Vëzhgueshmëria

- Të shpjegohet **rëndësia e regjistrimit të centralizuar (Centralized Logging)** dhe përdorimi i mjeteve si **ELK Stack (Elasticsearch, Logstash, Kibana) ose Loki**.
- Të paraqiten **metrikat dhe alarmet për performancën e sistemit (Prometheus, Grafana)**.

3. Arkitektura e Mikroshërbimeve

3.1. Përshkrimi i përgjithshëm

Arkitektura e mikroshërbimeve përfaqëson një qasje moderne në zhvillimin e softuerit ku aplikacionet ndahen në një seri shërbimesh të vogla, autonome dhe të specializuara. Ndryshe nga aplikacionet monolitike tradicionale, ku i gjithë funksionaliteti është i paketuar në një njësi të vetme, mikroshërbimet ndërtojnë aplikacionet si një koleksion komponentësh të pavarur, secili përgjegjës për një funksion specifik biznesi.

Çdo mikroshërbim funksionon si një aplikacion i pavarur me cikël jetësor të veçantë dhe komunikon me shërbimet e tjera përmes protokolleve të lehta si REST API ose mesazheve asinkrone. Kjo ndarje e përgjegjësive siguron që ekipet të mund të zhvillojnë, vendosin dhe shkallëzojnë shërbime individualisht, pa ndikuar në funksionimin e sistemit më të gjerë.

Adoptimi i arkitekturës së mikroshërbimeve ka njohur një rritje të konsiderueshme në vitet e fundit, veçanërisht në ndërmarrjet që kërkojnë fleksibilitet, shkallëzueshmëri dhe cikle më të shpejta zhvillimi. Organizata si Netflix, Amazon, Spotify dhe shumë të tjera kanë demonstruar suksesin e kësaj qasjeje në mbështetjen e operacioneve të tyre në shkallë të gjerë.

3.2. Përfitimet dhe sfidat

Përfitimet:

- **Shkallëzueshmëri e lartë:** Lejon rritjen selektive të shërbimeve sipas nevojës, duke ulur kostot dhe optimizuar burimet. P.sh., shërbimi i pagesave mund të shkallëzohet pa ndikuar të tjerat.
- **Fleksibilitet teknologjik:** Mundëson përdorimin e teknologjive të ndryshme për shërbime të ndryshme, sipas nevojave specifike. P.sh., Python për analiza të dhënash, Go për performancë të lartë.
- **Qëndrueshmëri e rritur:** Dështimi i një shërbimi nuk ndikon në të gjithë sistemin, duke siguruar vazhdimësinë e operacioneve.
- **Mirëmbajtje më e thjeshtë:** Shërbimet e vogla janë më të lehta për t'u kuptuar, modifikuar dhe përditësuar.
- **Zhvillim dhe shpërndarje të pavarur:** Ekipet punojnë paralelisht pa ndërvarësi të forta, duke përmirësuar proceset CI/CD.

- **Organizim sipas domeneve të biznesit:** Mikroshërbimet strukturohen sipas funksioneve të biznesit, duke përmirësuar koordinimin mes ekipeve teknike dhe biznesit.

Sfidat:

- **Kompleksitet i lartë:** Menaxhimi i shumë shërbimeve dhe ndërveprimeve të tyre kërkon mjete dhe njohuri të specializuara.
- **Komunikimi mes shërbimeve:** Nevojiten mekanizma të besueshëm për trajtimin e problemeve të rrjetit dhe vonesave.
- **Menaxhimi i të dhënave:** Konsistenca mes shërbimeve është sfidë, duke kërkuar strategji për transaksione të shpërndara.
- **Monitorimi dhe diagnostikimi:** Gjurmimi i problemeve në sisteme të shpërndara kërkon mjete të avancuara.
- **Përcaktimi i kufijve të shërbimeve:** Ndarja e duhur e domeneve të biznesit është vendimtare për shmangien e ndërvarësive.
- **Menaxhimi i versionit:** Ndryshimet duhet të trajtohen me kujdes për të ruajtur përputhshmërinë mes shërbimeve.
- **Siguria:** Kërkohet kontroll i rreptë mbi autentikimin, autorizimin dhe enkriptimin për të shmangur dobësitë e sigurisë.

Trajtimi i këtyre sfidave kërkon praktika të mirë inxhinierike, mjete të specializuara dhe strategji proaktive.

3.3. Parimet Themelore të Dizajnit të Mikroshërbimeve

1. **Pavarësia e Shërbimeve** – Çdo mikroshërbim zhvillohet, vendoset dhe funksionon në mënyrë të pavarur, duke pasur bazën e vet të të dhënave dhe minimizuar ndikimin e ndryshimeve në sistemin e përgjithshëm.
2. **Decentralizimi** – Çdo shërbim menaxhon të dhënat dhe vendimet e veta, duke shmangur varësitë e përbashkëta dhe duke rritur autonominë e ekipeve.
3. **Komunikimi përmes API-ve** – Ndërveprimi ndodh përmes protokolleve të thjeshta si REST ose gRPC, duke kufizuar ekspozimin e të dhënave dhe duke siguruar kufij të qartë midis shërbimeve.
4. **Toleranca ndaj Dështimeve** – Projektimi i sistemit supozon se shërbimet mund të dështojnë, duke përdorur mekanizma si circuit breakers, retries dhe izolim të gabimeve për të ruajtur stabilitetin.

5. **Automatizimi i Infrastrukturës** – Përdorimi i CI/CD, kontejnerizimi me Docker dhe orkestrimi me Kubernetes sigurojnë shpërndarje të shpejtë dhe të besueshme të shërbimeve.
6. **Shkallëzueshmëria** – Çdo mikroshërbim duhet të jetë i aftë të shkallëzohet në mënyrë të pavarur për të përballuar ngarkesën e rritur pa ndikuar sistemin e përgjithshëm.
7. **Përgjegjësia e Vetme** – Çdo mikroshërbim duhet të kryejë një funksion të qartë dhe të përcaktuar, duke shmangur mbingarkimin me funksionalitete të shumta.
8. **Monitorimi dhe Gjurmimi** – Sistemet e mikroshërbimeve duhet të kenë monitorim të vazhdueshëm dhe gjurmim të kërkesave për të identifikuar dhe zgjidhur problemet në kohë.
9. **Ekipet Autonome** – Zhvillimi dhe mirëmbajtja e mikroshërbimeve duhet të menaxhohet nga ekipe të pavarura, të cilat kanë kontroll të plotë mbi ciklin e jetës së shërbimit të tyre.
10. **Load Balancing (Balancimi i Ngarkesës)** – Për të siguruar performancë të qëndrueshme dhe shpërndarje optimale të burimeve, mikroshërbimet duhet të përdorin mekanizma të balancimit të ngarkesës.

4. Praktikat për Zhvillimin dhe Menaxhimin e Mikroshërbimeve

4.1. Containerization dhe Orchestration

Containerization është bërë një komponent thelbësor i çdo implementimi modern të mikroshërbimeve, duke ofruar një mënyrë të standardizuar për paketimin dhe vendosjen e aplikacioneve. Kontejnerët sigurojnë izolim, portabilitet dhe konsistencë midis mjediseve të zhvillimit, testimit dhe prodhimit.

Docker për Containerization

Docker është bërë standardi de facto për containerization, duke ofruar një platformë të lehtë për ndërtimin, shpërndarjen dhe ekzekutimin e aplikacioneve në mjedise të

izoluara. Kontejnerët Docker paketojnë aplikacionin së bashku me të gjitha varësitë e tij (biblioteka, frameworks, etj.) në një njësi të vetme ekzekutimi.

Kontejnerët sigurojnë përfitimet e mëposhtme:

- Konsistencë në mjedise të ndryshme: Eliminon problemet klasike "funksionon në kompjuterin tim" duke siguruar mjedis identik ekzekutimi në zhvillim, testim dhe prodhim.
- Izolim i burimeve: Çdo kontejner operon në një mjedis të izoluar, duke parandaluar konfliktet midis shërbimeve që përdorin versione të ndryshme të librarive të njëjta.
- Efikasitet i burimeve: Kontejnerët ndajnë kernelin e sistemit operativ bazë, duke i bërë ato më të lehtë se makinat virtuale tradicionale.
- Shpërndarje dhe shkallëzim i lehtë: Kontejnerët mund të vendosen dhe shkallëzohen shpejt për të përmbushur kërkesat në ndryshim.

Praktikat më të mira për containerization përfshijnë:

- Implementimi i ndërtimit multi-stage për të minimizuar madhësinë përfundimtare të imazhit.
- Ekzekutimi i kontejnerëve si përdorues jo-root për siguri të përmirësuar.
- Aplikimi i skanimeve të sigurisë në imazhet e kontejnerëve për të identifikuar vulnerabilitetet.

Kubernetes për Orchestration

Ndërsa kontejnerët ofrojnë njësinë bazë të paketimit dhe shpërndarjes, orkestrimi i tyre në shkallë bëhet kritik për aplikimet e bazuara në mikroshërbime. Kubernetes ka dalë si zgjidhja dominuese për orchestration, duke ofruar një platformë të fuqishme për automatizimin e vendosjes, shkallëzimit dhe menaxhimit të aplikacioneve të kontejnerizuara.

Kubernetes ofron karakteristika kyçe për mikroshërbimet:

- Vetë-shërimi: Zëvendësimi automatik i kontejnerëve që dështojnë ose nuk i kalojnë kontrollet e shëndetit.
- Shkallëzimi horizontal: Rregullimi automatik i numrit të instancave bazuar në përdorimin e CPU-së ose metrikave të tjera.

- Zbulimi i shërbimeve dhe balancimi i ngarkesës: Mekanizmat e integruar për regjistrimin dhe gjetjen e shërbimeve.
- Menaxhimi i konfigurimit dhe sekreteve: Ruajtja e sigurt dhe shpërndarja e të dhënave të konfigurimit dhe informacioneve sensitive.
- Strategjitë e shpërndarjes së avancuara: Mbështetje për përditësime graduale, deployment-e blue/green dhe canary releases.

Praktikat më të mira për orkestrimin me Kubernetes përfshijnë:

- Organizimi i burimeve në namespaces logjike që pasqyrojnë njësitë e biznesit ose ekipet.
- Përdorimi i konfigurimeve deklarative (YAML) dhe menaxhimi i tyre në versionim.
- Implementimi i kontrollit të shëndetit dhe gatishmërisë për çdo shërbim.
- Vendosja e kufijve të burimeve (CPU, memorie) për çdo kontejner për të parandaluar degradimin e sistemit.

4.2. Zbulimi i Shërbimeve (Service Discovery)

Në një arkitekturë mikroshërbimesh, instancat e shërbimeve krijohen dhe shkatërrohen vazhdimisht si rezultat i shkallëzimit, përditësimeve dhe dështimeve. Zbulimi i shërbimeve siguron që klientët dhe shërbimet e tjera të mund të lokalizojnë dhe komunikojnë me instancat e disponueshme pa adresa fikse.

Mekanizmat e Zbulimit të Shërbimeve

Ekzistojnë disa zgjidhje të specializuara për zbulimin e shërbimeve:

Netflix Eureka është një shërbim zbulimi i bazuar në server, i projektuar për aplikacionet në cloud. Si pjesë e ekosistemit Spring Cloud, Eureka integrohet lehtësisht me aplikacionet Java. Eureka funksionon sipas modelit client-server, ku çdo shërbim regjistrohet në serverin Eureka gjatë nisjes dhe dërgon heartbeats të rregullta për të konfirmuar disponueshmërinë.

HashiCorp Consul ofron një zgjidhje më të plotë që kombinon zbulimin e shërbimeve me menaxhimin e konfigurimit, monitorimin e shëndetit dhe segmentimin e rrjetit. Consul përdor një model të shpërndarë me një bazë të dhënash në memorie të konsistencës eventuale. Përveç API RESTful, Consul ofron edhe një

DNS interface për zbulimin e shërbimeve, duke e bërë integrimin të lehtë me infrastrukturën ekzistuese.

etcd është një depo e shpërndarë key-value që përdoret nga Kubernetes dhe shërben si depo e qëndrueshme për të dhënat e konfigurimit, metadata dhe për koordinimin e shërbimeve. Ndërsa nuk është ekskluzivisht një zgjidhje zbulimi shërbimesh, etcd ofron primitiva të nevojshme për ndërtimin e mekanizmave të zbulimit.

Strategjitë e Regjistrimit të Shërbimeve

Ekzistojnë dy modele kryesore për regjistrimin e shërbimeve:

1. Vetë-regjistrimi (Self-registration): Shërbimet regjistrojnë veten e tyre në regjistrarin e shërbimeve gjatë nisjes dhe çregjistrohen gjatë mbylljes. Ky model i jep shërbimeve përgjegjësinë për të menaxhuar regjistrimin e tyre.

2. Regjistrimi nga palët e treta (Third-party registration): Një agjent i veçantë monitoron shfaqjen e instancave të reja dhe i regjistron ato automatikisht. Kubernetes përdor këtë model me sistemin e tij të integruar DNS.

Zgjidhja e Emrave dhe Balancimi i Ngarkesës

Pasi shërbimet janë regjistruar, klientët duhet të jenë në gjendje t'i zbulojnë dhe të komunikojnë me to:

- Zgjidhja e emrave të shërbimeve (Service name resolution): Klientët përdorin emra logjikë shërbimesh në vend të adresave fizike. Regjistri i shërbimit i zgjidh këto emra në adresa aktuale.

- Client-side load balancing: Biblioteka si Netflix Ribbon implementojnë algoritma balancimi në anën e klientit, duke zgjedhur midis instancave të disponueshme të shërbimit për çdo kërkesë.

- Server-side load balancing: Një load balancer i dedikuar (si NGINX, HAProxy ose një API Gateway) drejton trafikun në instancat e duhura.

Praktikat më të mira për zbulimin e shërbimeve përfshijnë:

- Implementimi i kontrolleve të shëndetit për të siguruar që vetëm instancat e shëndosha të shërbimeve të marrin trafik.
- Ruajtja e informacionit të metadata-s me regjistrimet e shërbimeve për të mundësuar rrugëzimin e avancuar (p.sh., shërbime në versione të ndryshme).
- Përdorimi i cache-ing në anën e klientit për të reduktuar vonesat dhe varësinë nga shërbimi i zbulimit.
- Sigurimi i disponueshmërisë së lartë për vetë regjistrarin e shërbimeve.

4.3. API Gateway dhe Menaxhimi i API-ve

API Gateway shërben si pikë hyrëse e unifikuar për klientët (web, mobile, etj.) që ndërveprojnë me një grup mikroshërbimesh. Ajo absorbson kompleksitetin e komunikimit me shërbime të shumta dhe siguron një ndërfaqe të pastër për klientët.

Roli i API Gateway

API Gateway siguron disa funksione kritike:

- Rutimi i kërkesave: Drejton kërkesa të ardhshme në shërbimet e duhura të backend.
- Agregimi i përgjigjeve: Kombinon përgjigjet nga shërbime të shumta në një përgjigje të vetme për klientin, duke reduktuar numrin e kërkesave.
- Transformimi i protokollit: Konverton midis protokolleve të ndryshme (p.sh., nga REST në gRPC) ose formateve (XML në JSON).
- Autentikimi dhe autorizimi: Siguron një shtresë të centralizuar sigurie për të gjitha shërbimet.
- Limitimi (Rate limiting): Mbron shërbimet nga mbingarkesa duke kufizuar numrin e kërkesave.
- Caching: Ruan përgjigjet e shpeshta për të përmirësuar performancën dhe për të reduktuar ngarkesën në mikroshërbime.
- Monitorimi dhe analitika: Siguron pikë të centralizuar për mbledhjen e metrikave dhe monitorimin e trafikut.

Implementimet e API Gateway

Disa nga zgjidhjet e njohura të API Gateway për mikroshërbime përfshijnë:

Spring Cloud Gateway është një API Gateway moderne e bazuar në projektin Spring WebFlux, që ofron një model programimi reaktiv. E përshtatshme veçanërisht për ekosistemin Java/Spring, Gateway mbështet rutimin dinamik, filtrat dhe mbështetje të integruar për zbulimin e shërbimeve.

Kong është një API Gateway e bazuar në NGINX, që ofron performancë të lartë dhe një ekosistem të pasur shtesash. Kong mbështet funksione si autentikimi, versionimi, caching dhe monitorimi, dhe mund të shkallëzohet për të menaxhuar aplikacione komplekse.

NGINX/NGINX Plus është një zgjidhje e shpejtë dhe e lehtë që funksionon si një proxy i kundërt, load balancer dhe API Gateway. Ndërsa NGINX ofron funksionalitete bazë, NGINX Plus shton karakteristika më të avancuara që janë veçanërisht të dobishme për mikroshërbimet.

Amazon API Gateway është një shërbim i menaxhuar për krijimin, publikimin dhe mirëmbajtjen e API-ve në çdo shkallë. I integruar me shërbimet e tjera AWS, API Gateway ofron sigurim të lehtë të SSL, autentifikim, limitim të shkallës dhe caching.

Versionimi i API-ve

Ndërsa mikroshërbimet evoluojnë, strategjitë e efektshme të versionimit bëhen kritike:

- Versionimi në path (URI): Përfshirja e versionit në URI (p.sh., /v1/resources).
- Versionimi me header: Përdorimi i custom headers (p.sh., X-API-Version: 1).
- Versionimi me accept header: Specifikimi i versionit në header-in Accept (p.sh., Accept: application/vnd.company.v1+json).
- Versionimi me parametra kërkese: Shtimi i versionit si parametër kërkese (p.sh., ?version=1).

Pavarësisht nga qasja, praktikat më të mira për menaxhimin e API-ve përfshijnë:

- Dizajnimi i API-ve për përputhshmëri me versionet e mëparshme kur është e mundur.
- Përdorimi i strategjive të migrimit gradual për të shmangur ndërprerjet për klientët.

- Krijimi i dokumentacionit të qartë për çdo version API dhe ndryshimet midis versioneve.
- Sigurimi i një periudhe të arsyeshme paralajmëruese para heqjes së versioneve të vjetra.

API Gateway-t dhe menaxhimi efektiv i API-ve janë thelbësore për përdorueshmërinë, performancën dhe evolucionin e një platforme mikroshërbimesh. Ata mundësojnë prezantimin e një fasade të pastër ndaj klientëve ndërsa absorbojnë kompleksitetin e sistemit të brendshëm.

4.4. Toleranca ndaj Gabimeve dhe Disponueshmëria e Lartë

Në një arkitekturë të shpërndarë si mikroshërbimet, gabimet janë të pashmangshme. Dizajnimi për tolerancë ndaj gabimeve dhe disponueshmëri të lartë është thelbësor për të siguruar që sistemi të mbetet funksional edhe kur komponentë të veçantë dështojnë.

Modeli Circuit Breaker

Modeli Circuit Breaker parandalon dështimet zinxhir duke "prishur qarkun" kur detektohen gabime të vazhdueshme. Në vend që të vazhdojnë të bëjnë thirrje që ka të ngjarë të dështojnë, shërbimet klient përdorin një circuit breaker që monitoron dështimet dhe, kur arrihet një prag i caktuar, ndërpret përpjekjet për një periudhë të specifikuar kohore.

Resilience4j është një bibliotekë tolerante ndaj gabimeve, e frymëzuar nga Netflix Hystrix por e dizajnuar për Java 8 dhe programimin funksional. Ajo ofron implementime të circuit breaker, rate limiting, retry, bulkheading dhe izolimit të kohës.

Hystrix, megjithëse tani në mirëmbajtje, vendosi standardin për tolerancën ndaj gabimeve në mikroshërbime. Ai ofron izolim nga varësitë e dështuara, ndërprerje të thirrjeve që zgjasin shumë, dhe komanda fallback për të ofruar përgjigje alternative.

Strategjitë Fallback

Kur një shërbim nuk është i disponueshëm, strategjitë fallback ofrojnë alternativa për të shmangur dështimin e plotë:

- Përgjigjet e cached: Kthimi i të dhënave të ruajtura në cache nga thirrjet e mëparshme të suksesshme.
- Degradimi i hijshëm: Ofrimi i funksionalitetit të kufizuar kur shërbimi i plotë nuk është i disponueshëm.
- Të dhënat default: Kthimi i përgjigjeve të paracaktuara kur nuk është e mundur të merren të dhëna aktuale.

Izolimi i Gabimeve (Bulkheading)

Bulkheading ndan sistemin në kompartimente të izoluara për të kufizuar ndikimin e dështimeve:

- Bulkheading i bazuar në thread pool: Secili shërbim ose grup shërbimesh merr një pool të dedikuar thread-esh.
- Bulkheading i bazuar në semafora: Kufizon numrin e thirrjeve të njëkohshme në një shërbim.

Testimi i Rezistencës

Testimi i sjelljes së sistemit gjatë kushteve të gabimeve është thelbësor:

- Chaos Engineering: Praktika e futjes së gabimeve të qëllimshme në sistemet e prodhimit për të testuar rezistencën.
- Netflix Chaos Monkey: Një mjet që ndërpret instancat e shërbimeve rastësisht për të testuar tolerancën ndaj gabimeve të sistemit.
- Latency Monkey: Simulon vonesat e shërbimeve për të testuar sjelljen e sistemit nën kushte të ngadalta të rrjetit.

Praktikat më të mira për tolerancën ndaj gabimeve përfshijnë:

- Dizajnimi i shërbimeve të pavarura me varësi minimale.
- Vendosja e timeout-eve të arsyeshme për të gjitha thirrjet e jashtme.
- Implementimi i mekanizmave retry me backoff eksponencial.
- Krijimi i testeve të simulimit të dështimeve si pjesë e procesit të testimit.
- Monitorimi dhe alarmimi për circuit breakerët që aktivizohen shpesh.

4.5. Modelet e Komunikimit midis Mikroshërbimeve

Komunikimi efektiv midis mikroshërbimeve është një nga sfidat kryesore në një arkitekturë të shpërndarë. Modelet e komunikimit ndahen kryesisht në dy kategori: sinkrone dhe asinkrone, secila me raste të veta të përdorimit dhe trade-offs.

Komunikimi Sinkron

Në komunikimin sinkron, klienti pret që të marrë një përgjigje nga shërbimi para se të vazhdojë përpunimin:

RESTful APIs janë mënyra më e zakonshme e komunikimit sinkron në mikroshërbime. Bazuar në HTTP, REST ofron një model të thjeshtë kërkesë-përgjigje me metoda standarde (GET, POST, PUT, DELETE) dhe formate të gjera të shkëmbimit të të dhënave (JSON, XML).

Përfitimet e REST përfshijnë:
- Thjeshtësi dhe familjaritet
- Mbështetje të integruar në shumicën e gjuhëve dhe frameworket
- Dokumentim të lehtë me standarde si OpenAPI/Swagger

gRPC është një framework RPC (Remote Procedure Call) me performancë të lartë i zhvilluar nga Google. Ndryshe nga REST, gRPC përdor Protocol Buffers (protobuf) për serializimin e të dhënave dhe HTTP/2 për transportin, duke ofruar efikasitet dhe performancë më të mirë.

Avantazhet e gRPC:
- Kornizë e fortë kontraktuale me protobuf
- Performancë e përmirësuar me serializim më të shpejtë
- Streaming dykahësh
- Gjenerimi automatik i kodit klient për shumë gjuhë programimi
- Mbështetje e integruar për trajtimin e gabimeve

GraphQL është një gjuhë kërkesash për API dhe një runtime për përmbushjen e këtyre kërkesave. E zhvilluar nga Facebook, GraphQL lejon klientët të kërkojnë

saktësisht të dhënat që u nevojiten, duke kombinuar shpesh informacione nga burime të shumta në një kërkesë të vetme.
Përfitimet e GraphQL:

- Fetching fleksibil i të dhënave që parandalon over-fetching dhe under-fetching
- Tipizim i fortë me një skemë deklarative
- Versionim i lehtësuar i API
- Bashkim i kërkesave të shumta në një kërkesë të vetme
- Mbështetje e mirë për tools dhe dokumentim të integruar

Komunikimi Asinkron

Në komunikimin asinkron, shërbimet vazhdojnë ekzekutimin pa pritur për një përgjigje, duke përdorur mesazh dhe evente: **Message Brokers** dhe **Message Queues** sigurojnë infrastrukturën për komunikim asinkron. Zgjidhjet popullore përfshijnë:

Apache Kafka është një platformë shpërndarjeje eventesh e shkallëzuar horizontalisht me kapacitet të lartë përpunimi. Kafka ofron replikim të të dhënave, tolerancë ndaj gabimeve dhe përpunim të eventeve në kohë reale, duke e bërë atë ideal për sisteme të mëdha mikroshërbimesh.

Karakteristikat kryesore:

- Arkivimi i qëndrueshëm i të dhënave
- Shkallëzim horizontal me shpërndarje të particioneve
- Përpunim i renditur i eventeve brenda particioneve
- Kapacitet i lartë transmetimi
- Garanci "at-least-once" dhe "exactly-once"

RabbitMQ është një ndërmjetës mesazhesh që implementon protokollin AMQP. RabbitMQ ofron fleksibilitet në modelet e mesazheve dhe është i përshtatshëm për skenarë kompleksë rrugëzimi.

Karakteristikat kryesore:

- Topologji fleksibël me exchanges dhe queues
- Modele të ndryshme mesazhi (publish/subscribe, routing, RPC)
- Konfirmime transaksionale
- Plugin-e për zgjerueshmëri
- Performancë e mirë për throughput mesatar

Event Sourcing dhe **CQRS**
Event Sourcing është një paradigmë ku ndryshimet në gjendje kapen si një sekuencë eventesh. Në vend që të ruhet gjendja aktuale, sistemi mban një log të të gjitha eventeve që ndikojnë në gjendjen, duke lejuar:

- Histori të plotë auditimi
- Rindërtim të gjendjes në çdo pikë kohore
- Veçanërisht i dobishëm për domain-e ku historia e ndryshimeve është e rëndësishme

Command Query Responsibility Segregation (CQRS) ndan operacionet e leximit (Queries) nga operacionet e shkrimit (Commands), duke lejuar optimizimin e secilit në mënyrë të pavarur. Kjo qasje shpesh kombinohet me Event Sourcing dhe është veçanërisht e dobishme në sistemet me raportin e lartë lexim-shkrim.

Praktikat më të Mira për Komunikimin në Mikroshërbime

- Zgjedhja e protokollit të duhur: Përdorni komunikim sinkron për operacione që kërkojnë përgjigje të menjëhershme dhe komunikim asinkron për operacione të gjata ose që mund të kryhen në sfond.
- Izolimi i gabimeve: Implementoni circuit breakers, timeout-e dhe retry-s për të parandaluar dështimet zinxhir.
- Dizajni i API-ve të shëndosha: Strukturoni API-të në mënyrë që ndryshimet të mos prishin klientët ekzistues.
- Dokumentim i qartë: Sigurohuni që çdo API të ketë specifikime të qarta dhe dokumentim të përditësuar.

- Versionim i kujdesshëm: Menaxhoni ndryshimet API përmes versionimit për të shmangur ndërprerjet.
- Konsideroni joreaktivitetin (unreactivity): Për API sinkrone, minimizoni ndërlidhjet midis shërbimeve duke reduktuar thirrjet kthyese.
- Vlerësoni overheadin e rrjetit: Në dizajnin e komunikimit, konsideroni koston e latencës së rrjetit dhe impaktin në performancën e përgjithshme.

5. Strategjitë e Deployment dhe CI/CD për Mikroshërbime

Një nga përfitimet kryesore të mikroshërbimeve është aftësia për të deployuar shërbime të pavarur, shpesh dhe me rrezik të ulët. Kjo kërkon praktika të matura DevOps dhe pipeline-e të automatizuara CI/CD (Continuous Integration/Continuous Deployment).

5.1. Continuous Integration (CI)

Continuous Integration fokusohet në integrimin e shpeshtë të ndryshimeve në kodin bazë, duke siguruar që ndryshimet e reja nuk prishin funksionalitetin ekzistues:
Automatizimi i ndërtimit dhe testimit:

- Konfigurimi i ndërtimeve automatike kur kodi dërgohet në repository
- Ekzekutimi i testeve njësi, integrimi dhe funksionale për të verifikuar cilësinë
- Analiza statike e kodit për të identifikuar probleme potenciale
- Ndërtimi i imazheve të kontejnerëve dhe testimi i tyre

Mjetet e CI:

Jenkins: Një server automatizimi open-source shumë i përshtatshëm me mbështetje të gjerë plugin-esh.
GitLab CI: Një zgjidhje e integruar CI për projektet e hostuara në GitLab.
GitHub Actions: Mundëson automatizimin e workflow-ve direkt në repository-t GitHub.
CircleCI: Një platformë CI/CD cloud e fokusuar në shpejtësi dhe fleksibilitet.

5.2. Continuous Deployment (CD)

CD shtrihet përtej CI për të automatizuar procesin e Deployment

Pipeline-et e Deployment:

- Mjediset progresive: Deployment i automatizuar në mjedise të ndryshme (dev, test, staging, production)
- Testime post-deployment: Verifikimi i funksionalitetit dhe performancës pas çdo deployment
- Mekanizmat rollback: Aftësia për t'u kthyer shpejt në versionin e mëparshëm në rast problemi

Strategjitë e Deployment:

- Blue/Green Deployment: Mbahen dy mjedise identike, me vetëm një aktiv në çdo moment:
    - Ndryshimet behen deploy në mjedisin jo-aktiv (green)
    - Testohen plotësisht pa ndikuar në përdoruesit
    - Trafiku kalon nga mjedisi aktual (blue) në mjedisin e ri (green)
    - Ofron rollback të menjëhershëm duke rikthyer trafikun

- Canary Releases: Ndryshimet zbulohen gradualisht te përdoruesit:
    - Version i ri vendoset paralelisht me të vjetrin
    - Një përqindje e vogël e trafikut drejtohet në versionin e ri
    - Monitorimi i kujdesshëm për probleme
    - Përqindja rritet gradualisht derisa të gjithë trafiku të shkojë në versionin e ri

- Rolling Updates: Instancat e shërbimit përditësohen një nga një:
    - Përshtatshme për aplikacione të shkallëzuara horizontalisht
    - Parandalon ndërprerjet e shërbimit
    - Zakonisht implementohet me lehtësi në platforma si Kubernetes

5.3. Menaxhimi i Konfigurimit

Menaxhimi efektiv i konfigurimit është kritik për Deployment të qëndrueshëm:

Eksternalizimi i konfigurimit:

- Ndarja e konfigurimit nga kodi
- Ruajtja e konfigurimit në sisteme të dedikuara (Spring Config Server, Consul, etcd)
- Konventat për konfigurimin bazuar në mjedis

Menaxhimi i sekreteve:

- Përdorimi i sistemeve të dedikuara të sekreteve (HashiCorp Vault, AWS Secrets Manager, Kubernetes Secrets)
- Enkriptimi i të dhënave sensitive
- Rotacioni i kredencialeve

5.4. Monitorimi dhe Gjurmimi

Monitorimi efektiv bëhet edhe më i rëndësishëm në një mjedis të shpërndarë mikroshërbimesh:
Mbledhja e metrikave:

- Metrika të niveleve të infrasrukturës (CPU, memorie, rrjet)
- Metrika të aplikacioneve (kohë përgjigje, throughput, error rate)
- Metrika të biznesit (transaksione, përdorues aktivë)

Instrumentimi i aplikacioneve:

- Prometheus për mbledhjen e metrikave
- Grafana për vizualizimin e metrikave
- ELK Stack (Elasticsearch, Logstash, Kibana) për mbledhjen dhe analizën e log-eve
- Jaeger ose Zipkin për tracing të shpërndarë

Alerts dhe dashboards:

- Konfigurimi i alarmeve për prurje dhe anomali
- Dashboards për vizualizim në kohë reale të shëndetit të sistemit
- Korrelimi i eventeve nga burime të shumta

6. Strategjitë e Databazave për Mikroshërbime

Menaxhimi i të dhënave në një arkitekturë mikroshërbimesh kërkon qasje të ndryshme nga aplikacionet monolitike.

6.1. Databaza për Shërbim (Database per Service)

Çdo mikroshërbim menaxhon dhe ruan të dhënat e veta në një databazë të dedikuar. Kjo siguron pavarësi, por shton kompleksitet kur kërkohet konsistencë në shërbime të shumta.

Përfitimet:

- Pavarësi e plotë për çdo shërbim
- Zgjedhja e databazës së duhur për nevojat specifike të çdo shërbimi
- Izolimi i gabimeve dhe performancës
- Shkallëzim i pavarur

Sfidat:

- Menaxhimi i konsistencës së të dhënave përgjatë databazave të shumta
- Kompleksiteti i transaksioneve që përfshijnë shumë shërbime
- Overhead operacional në menaxhimin e databazave të shumta

6.2. Zgjedhja e Teknologjisë së Databazës

Arkitektura e mikroshërbimeve mundëson zgjedhjen e tipave të ndryshëm të databazave sipas nevojave të shërbimeve individuale:
Databazat SQL (Relacionale):

- Të përshtatshme për të dhëna të strukturuara me relacione komplekse

- Ofrojnë garanci ACID (Atomicity, Consistency, Isolation, Durability)
- Shembuj: PostgreSQL, MySQL, SQL Server

Databazat NoSQL:

- Document stores (MongoDB, Couchbase): Për të dhëna gjysmë të strukturuara
- Key-value stores (Redis, DynamoDB): Për caching dhe lookups të shpejta
- Column-family stores (Cassandra, HBase): Për shkrimin e përmbajtjeve të mëdha të të dhënave
- Graph databases (Neo4j, JanusGraph): Për të dhëna të lidhura në mënyrë komplekse

Databazat Time-series:

- Të optimizuara për të dhëna të bazuara në kohë
- Përshtatshme për monitorim dhe metrika
- Shembuj: InfluxDB, TimescaleDB

6.3. Menaxhimi i Transaksioneve të Shpërndara

Trajtimi i transaksioneve që përfshijnë shumë shërbime kërkon teknika të specializuara:
Saga Pattern:

- Sekuencë operacionesh lokale, ku çdo operacion publikon një event që mund të trigger-ojë operacionin e radhës
- Implementimi i compensating transactions për rollback në rast dështimi
- Mund të implementohet në mënyrë koreografike (event-driven) ose orkestrative (koordinator qendror)

Eventual Consistency:

- Pranimi që të dhënat mund të jenë temporalisht jo-konsistente
- Sigurimi që sistemet do të konvergojnë në një gjendje konsistente
- Përdorimi i eventeve dhe mesazheve për të propaguar ndryshimet

Two-Phase Commit (2PC):

- Një protokoll për transaksione atomike të shpërndara
- Ka penalitete performance dhe disponueshmërie
- Përdoret rrallë në mikroshërbime për shkak të overheadit

6.4. Strategjitë për Mirëmbajtjen e Databazave

Evolucioni i skemës:

- Përdorimi i mjeteve për migrimet e databazave (Flyway, Liquibase)
- Praktika të versionimit të skemës
- Ndryshime kompatabile që mbështesin deployment-e graduale

Backing up dhe disaster recovery:

- Strategji specifike për çdo tip databaze
- Testimi i rregullt i proceseve të rikuperimit
- Replikimi dhe shpërndarja gjeografike për disponueshmëri të lartë

7. Siguria në Mikroshërbime

Siguria në një arkitekturë të shpërndarë kërkon qasje të ndryshme nga aplikacionet tradicionale monolitike.

7.1. Autentikimi dhe Autorizimi

Autentikimi i Centralizuar:

- Implementimi i Identity Provider të centralizuar (OAuth 2.0, OpenID Connect)
- Përdorimi i JSON Web Tokens (JWT) për transmetimin e të dhënave të identitetit
- Single Sign-On (SSO) për përvojë të qëndrueshme përdoruesi

Autorizimi i Shpërndarë:

- Role-Based Access Control (RBAC)
- Attribute-Based Access Control (ABAC) për politika më komplekse

- Zbatimi i autorizimit në çdo nivel (API Gateway, shërbim individual)

7.2. Komunikimi i Sigurt

TLS Everywhere:

- Enkriptim i të gjithë komunikimit midis shërbimeve
- Menaxhim i automatizuar i certifikatave (përdorimi i Let's Encrypt, cert-manager)
- Mutual TLS (mTLS) për autentifikim të dyanshëm shërbim-me-shërbim

Politikat e Rrjetit:

- Kufizimi i komunikimit midis shërbimeve (përdorimi i Kubernetes Network Policies)
- Implementimi i parimit të privilegjit minimal
- Segmentimi i rrjetit për izolim të përmirësuar

7.3. Skanimi i Vulnerabiliteteve

Skanimi i kodit:

- Analiza statike e kodit për probleme sigurie
- Skanimi i varësive për vulnerabilitete të njohura (npm audit, OWASP Dependency Check)
- Skanimi i imazheve të kontejnerit për probleme sigurie (Trivy, Clair)

Testimi i vazhdueshëm i sigurisë:

- Integrim i testeve të sigurisë në pipeline-et CI/CD
- Penetration testing i rregullt
- Bug bounty programs

7.4. Monitorimi i Sigurisë

Loggimi i centralizuar i sigurisë:

- Mbledhja e të gjitha logeve të sigurisë në një sistem të centralizuar
- Identifikimi i modeleve anomale që mund të tregojnë për sulme
- Korelimi i eventeve përgjatë sistemeve të ndryshme

Detektimi i ndërhyrjeve:

- Monitorimi për aktivitet të dyshimtë
- Përgjigja e automatizuar ndaj kërcënimeve të njohura
- Raportimi dhe alertimi për incidente potenciale sigurie

8. Konkluzione

Arkitektura e mikroshërbimeve ofron fleksibilitet, shkallëzim dhe mirëmbajtje të lehtësuar, por kërkon një qasje të disiplinuar ndaj dizajnimit dhe menaxhimit të sistemit. Kalimi gradual nga një arkitekturë monolitike modulare në mikroshërbime mund të ndihmojë në shmangien e ndërlikimeve të panevojshme. Struktura e shërbimeve duhet të pasqyrojë ndarjen natyrore të domeineve të biznesit, duke siguruar një ekuilibër mes nivelit të detajimit dhe lehtësisë së menaxhimit.
Automatizimi dhe praktikat e vazhdueshme të testimit dhe shpërndarjes ndihmojnë në rritjen e efikasitetit dhe qëndrueshmërisë së sistemit. Po ashtu, toleranca ndaj dështimeve dhe një strategji e fuqishme monitorimi janë kritike për funksionimin e qëndrueshëm të aplikacioneve të bazuara në mikroshërbime.
Në fund, suksesi i mikroshërbimeve varet nga zbatimi i praktikave më të mira dhe përshtatja e vazhdueshme e arkitekturës me nevojat e biznesit dhe teknologjisë.