[개인 프로젝트] 단축 URL 서비스 개발 프로젝트

2024.04 ~ 2024.05

#TypeScript #NestJS #MongoDB #Redis #Kafka #Docker #AWS #Git #Github

Github

https://github.com/gitaepark97/nestjs-shorten-url-service

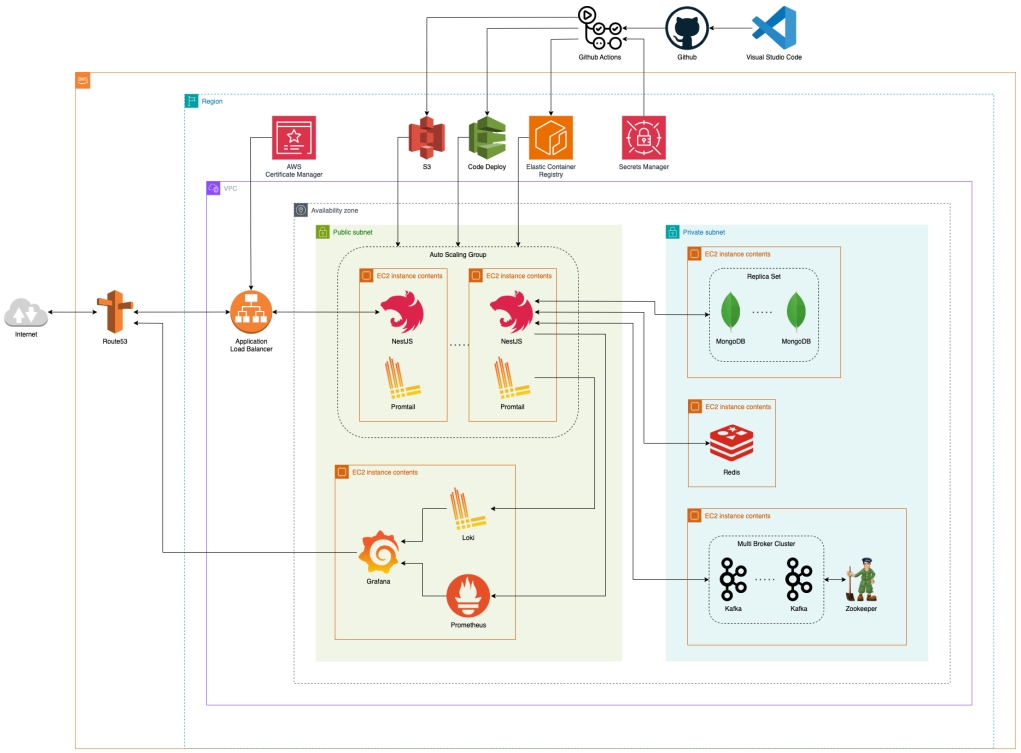
개요

유튜버 조코딩님의 2024년 취업 시장과 백엔드 개발자가 되기 위한 취업 가이드(ft. 이준형 저자님) 영상의 단축 URL 서비스 요구사항을 참고하여 개발한 단축 URL 서비스입니다.

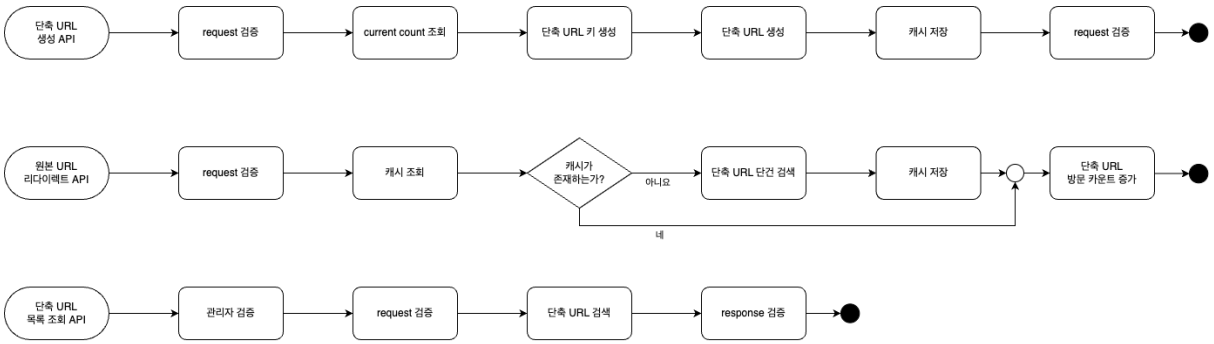
요구사항

1. bitly와 같은 단축 URL 서비스를 만들어야 합니다.
2. 단축된 URL의 키는 7글자로 생성되어야 합니다.
3. 사용자가 단축된 URL로 요청하면 원래의 URL로 리다이렉트되어야 합니다.
4. 원래의 URL로 다시 단축 URL을 생성해도 항상 새로운 단축 URL이 생성되어야 합니다. 이때 기존에 생성되었던 단축 URL도 여전히 동작해야 합니다.
5. 단축 URL이 원본 URL로 리다이렉트될 때마다 카운트가 증가되어야 하고, 해당 정보를 확인할 수 있는 API가 있어야 합니다.

시스템 아키텍처



플로우 차트



프로젝트 특징

#단축 URL 키 생성 알고리즘 설계

단축 URL을 생성하기 위해 숫자 타입의 count 값을 관리하여 해당 값을 Base65 Url로 인코딩했습니다. 이를 위해 WAS에는 Count 도메인을 작성하여 start, current, end 속성을 포함시켰습니다. 단축 URL을 생성할 때마다 current 값을 증가시키고, current와 end 값이 동일한 경우에는 MongoDB에 저장되어 있는 사용 가능한 count의 start 값을 가져오도록 했습니다. 이러한 방식을 통해 MongoDB 자체에 count를 저장하여 증가시키는 방법보다 더 빠른 속도를 얻을 수 있었습니다.

#유지 보수가 용이한 구조 설계

NestJS에서 제공하는 DI(Dependency Injection) 기능에 추상 클래스를 통한 추상화를 추가하여 의존성 역전 원칙을 준수하는 DI를 통해 의존성을 낮췄습니다. 또한 헥사고날 아키텍처의 포트와 어댑터 구조를 적용하여, 서비스의 비즈니스 로직이 외부 라이브러리에 낮게 의존하도록 설계했습니다. 이를 통해 특정 라이브러리나 데이터베이스가 변경되더라도 비즈니스 로직은 변경하지 않아도 되므로, 유지 보수가 용이한 구조를 만들 수 있었습니다.

#테스트 코드 작성

Jest를 활용하여 E2E, 통합, 유닛 테스트 코드를 작성했습니다.
외부 라이브러리를 사용하는 어댑터의 경우 외부 라이브러리와의 상호작용을 검증하기 위해 통합 테스트 코드를 작성했습니다. 이를 통해 실제 환경에서의 외부 라이브러리와의 통합 동작을 확인하고 문제가 있는 경우 신속하게 해결할 수 있었습니다.
또한, 비즈니스 로직에 대해서는 유닛 테스트 코드를 작성하여 각 함수 또는 메서드가 의도한 대로 동작하는지를 확인했습니다. 이를 통해 코드 변경 시 예상치 못한 버그를 방지하고 안정적인 소프트웨어를 유지할 수 있었습니다.

#중복되는 부가기능 처리

로깅, 요청과 응답의 검증, 모니터링에 필요한 메트릭 생성, 예외 처리는 모든 비즈니스 로직에서 필요로 하는 기능이므로 NestJS의 Middleware, Pipe, Interceptor, Filter를 활용하여 구현하여 관점 지향적으로 설계했습니다.

#캐시 적용으로 인한 응답속도 개선

원본 URL 리다이렉트 API는 동일한 단축 URL 키를 기반으로 요청할 때마다 동일한 원본 URL을 응답합니다. 또한 단축 URL 생성 API 호출 직후 원본 URL 리다이렉트 API 호출을 하는 경우가 높고 단축 URL 생성 API 호출보다 원본 URL 리다이렉트 API 호출의 수가 많을 것으로 예상됩니다. 따라서 단축 URL 생성 시와 단축 URL 조회 시 단축 URL을 캐시하면 WAS에서 MongoDB에 접근하는 횟수를 줄일 수 있습니다. 단축 URL 서비스의 WAS는 여러 Amazon EC2 인스턴스에서 동작하기 때문에, 로컬 메모리에 캐시할 경우 캐시 미스가 발생할 확률이 높습니다. 이 문제를 메모리를 공유하는 방식으로 해결하기 위해 Redis를 사용했습니다. 또한 가상 메모리를 사용하지 못하게 maxmemory를 설정했습니다.

#메시지 큐를 통한 비동기 처리

원본 URL 리다이렉트 API에서 단축 URL의 방문 카운트를 증가시키는 로직은 사용자에게 원본 URL을 리다이렉트하는 로직과 동기적으로 이루어져야할 필요가 없습니다. 이를 비동기적으로 처리하기 위해 Kafka를 적용했습니다. 이를 통해 사용자에게 더 빠른 서비스를 제공할 수 있었습니다.
메시지 큐를 사용했을 때 발생할 수 있는 메시지의 유실과 중복 문제는 각각 DLT(Dead Letter Topic)와 멱등성 producer로 해결하였습니다. 이러한 접근 방식을 통해 안정적이고 신뢰성 있는 메시지 큐 시스템을 구축하여 데이터 처리의 일관성과 정확성을 유지할 수 있었습니다.

#CI/CD 파이프 라인 구축

소스 코드를 로컬 개발 환경에서 GitHub 저장소로 push/merge하면, GitHub Actions를 활용하여 자동으로 테스트 코드를 실행하고 빌드를 진행합니다. 테스트와 빌드가 성공적으로 완료되면, Docker 이미지를 빌드하고 Amazon EC2 Container Registry(ECR)로 push합니다. 동시에 S3 버킷에 배포 스크립트를 저장합니다.
이후, Amazon CodeDeploy를 활용하여 Amazon S3에 저장된 배포 스크립트를 실행하여 Amazon EC2 Auto Scaling 그룹에 blue/green 배포를 수행합니다. 이를 통해 소프트웨어의 지속적인 통합과 배포가 자동화되어 개발 및 배포 프로세스의 효율성을 높일 수 있었습니다.

#서버 모니터링을 통한 분석

Winston과 Loki, Promtail을 사용하여 여러 Amazon EC2 인스턴스에서 실행되는 WAS의 로그를 효과적으로 수집하고 관리했습니다. 이러한 로그 수집은 애플리케이션의 문제를 빠르게 파악하고 해결하는 데 중요한 역할을 했습니다.
또한, Prometheus를 통해 여러 Amazon EC2 인스턴스에서 실행되는 WAS의 메트릭을 수집하였습니다. 이러한 메트릭 수집은 시스템의 성능 모니터링 및 성능 최적화를 위해 필수적이었습니다.
수집된 로그와 메트릭 정보는 Grafana를 통해 시각화하여, 실시간으로 시스템 상태를 모니터링하고 분석할 수 있도록 하였습니다. 이를 통해 전체 시스템의 안정성과 성능을 지속적으로 유지하고 개선할 수 있었습니다.

#가용성이 높은 아키텍처 설계

MongoDB Replica Set과 Kafka Multi Brokers, Amazon EC2 Auto Scaling Group을 활용하여 서비스의 가용성을 높이고 장애에 대응할 수 있는 강력한 기반을 마련함으로써 사용자에게 더욱 안정적이고 신뢰할 만한 서비스를 제공할 수 있도록 하였습니다.

#서버 성능 테스트

K6를 활용하여 다양한 유형의 성능 테스트를 수행하여 예상 트래픽에 대한 시스템의 처리 능력과 장기간 안정적인 운영 가능성을 확인하고 사용자 경험을 향상시킬 수 있는 개선점을 발견했습니다.

#문서화

Open API를 활용하여 API 문서를 자동화하여 개발자들이 API를 쉽게 이해하고 사용할 수 있도록 했습니다. 또한, 유스케이스 및 시퀀스 다이어그램을 작성하여 프로젝트의 요구사항과 시스템 동작을 명확하게 표현하여 개발자들 간의 원활한 의사소통을 도모했습니다.

#보안 처리

단축 URL 서비스에서는 보안을 강화하기 위해 다양한 조치를 취했습니다.
먼저, 환경변수는 AWS Secrets Manager를 활용하여 안전하게 관리했습니다. 이를 통해 민감한 정보를 안전하게 저장하고 접근을 제어할 수 있었습니다.
또한 WAS, MongoDB, Redis, Kafka 등의 외부 네트워크에서의 요청이 필요하지 않은 시스템은 Amazon VPC 내 Private subnet에 위치시켜 외부 노출을 최소화했습니다.
내부에서 외부 네트워크로의 요청이 필요한 경우에는 NAT Gateway를 통해 안전하게 요청을 전달했습니다.
더불어, Github와 AWS 등 여러 서비스에서 사용되는 Amazon IAM 역할과 사용자에게는 최소 권한 원칙을 적용하여, 필요한 권한만 부여함으로써 보안을 강화했습니다. 이를 통해 불필요한 접근을 방지하고 시스템에 대한 접근을 엄격히 제어할 수 있었습니다.