# S3 EventBridge Health Check

we have set of s3 and Eventbridge which needs to connected to a spring based application.

Ok before initiazliaing the application, customer want to proof check that the application is able to connect with the was component successfully and if it fails to any one, it will interrupt the application.

To implement a proof-check mechanism in your Spring-based application for validating connectivity to S3 and EventBridge before initialization, follow these steps:

## 1. Custom Health Check Implementation

Create custom health indicators for S3 and EventBridge using Spring's `HealthIndicator` interface.

### S3 Health Check

```
java


import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.s3.S3Client;

@Component
public class S3HealthIndicator implements HealthIndicator {
    private final S3Client s3Client;

    public S3HealthIndicator(S3Client s3Client) {
        this.s3Client = s3Client;
    }

    @Override
    public Health health() {
        try {
            s3Client.listBuckets(); // Example operation to test connectivity
            return Health.up().withDetail("S3", "Available").build();
        } catch (Exception e) {
            return Health.down().withDetail("S3", "Not Available").withException(e).build();
        }
    }
}
```

### EventBridge Health Check

```
java


import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
```

```
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;

@Component
public class EventBridgeHealthIndicator implements HealthIndicator {
    private final EventBridgeClient eventBridgeClient;

    public EventBridgeHealthIndicator(EventBridgeClient eventBridgeClient) {
        this.eventBridgeClient = eventBridgeClient;
    }

    @Override
    public Health health() {
        try {
            eventBridgeClient.listEventBuses(); // Example operation to test connectivity
            return Health.up().withDetail("EventBridge", "Available").build();
        } catch (Exception e) {
            return Health.down().withDetail("EventBridge", "Not Available").withException
(e).build();
        }
    }
}
```

## 2. Initialize Application After Health Check

You can use Spring's `ApplicationRunner` or `CommandLineRunner` to run pre-initialization checks.

### ApplicationRunner with Validation

```
java


import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthEndpoint;

@Component
public class PreInitializationRunner implements ApplicationRunner {
    private final HealthEndpoint healthEndpoint;

    public PreInitializationRunner(HealthEndpoint healthEndpoint) {
        this.healthEndpoint = healthEndpoint;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        Health health = healthEndpoint.health();
        if (!"UP".equals(health.getStatus().getCode())) {
            throw new RuntimeException("Application failed to connect to required AWS compon
ents: " + health);
        }
    }
}
```

## 3. Ensure Proper Shutdown on Failure

Configure the application to terminate if the health check fails. This ensures no further initialization happens if any AWS component is unreachable.

### Add Exit Code Generator

```java
import org.springframework.boot.ExitCodeGenerator;
import org.springframework.stereotype.Component;

@Component
public class HealthCheckFailureExitCodeGenerator implements ExitCodeGenerator {
    private final HealthEndpoint healthEndpoint;

    public HealthCheckFailureExitCodeGenerator(HealthEndpoint healthEndpoint) {
        this.healthEndpoint = healthEndpoint;
    }

    @Override
    public int getExitCode() {
        if (!"UP".equals(healthEndpoint.health().getStatus().getCode())) {
            return 1; // Non-zero exit code indicates failure
        }
        return 0;
    }
}
```

## 4. Configure AWS Clients

Ensure the AWS SDK clients (S3Client, EventBridgeClient) are correctly configured in your application.

### AWS Client Beans

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;

@Configuration
public class AwsClientConfig {

    @Bean
    public S3Client s3Client() {
        return S3Client.builder().build();
    }

    @Bean
    public EventBridgeClient eventBridgeClient() {
        return EventBridgeClient.builder().build();
    }
}
```

## 5. Test Your Setup

Run the application in an environment where S3 and EventBridge are configured. Simulate connectivity issues to validate the interruption mechanism.

## 6. Optional: Integrate with Spring Actuator

If you use Spring Actuator, the custom health indicators will automatically show up under `**/actuator/health**`. This helps in monitoring the system's health during runtime.

**Enable Actuator in `application.yml`**

```yaml


  management:
    endpoints:
      web:
        exposure:
          include: health
```

With this setup:

- The application will validate connectivity to S3 and EventBridge during startup.
- If connectivity fails, the application will terminate with an appropriate error message.

In the spring project..
default.yml
k:
 a: 5
 b: 6
 c: 7

k1:
 c: 8

To use the configuration in `default.yml` effectively in your Spring project, you can leverage Spring's `@ConfigurationProperties` or `@Value` annotations. Here's how you can handle the given structure:

### `default.yml` Content

```yaml


  k:
    a: 5
    b: 6
    c: 7

  k1:
    c: 8
```

### 1. Using `@ConfigurationProperties`

This approach allows you to bind hierarchical properties into a Java class.

**Create a Configuration Class**

```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.Map;

@Component
@ConfigurationProperties(prefix = "k")
public class KProperties {
    private int a;
    private int b;
    private int c;

    // Getters and setters
    public int getA() {
        return a;
    }

    public void setA(int a) {
        this.a = a;
    }

    public int getB() {
        return b;
    }

    public void setB(int b) {
        this.b = b;
    }

    public int getC() {
        return c;
    }

    public void setC(int c) {
        this.c = c;
    }
}

@Component
@ConfigurationProperties(prefix = "k1")
public class K1Properties {
    private int c;

    // Getters and setters
    public int getC() {
        return c;
    }

    public void setC(int c) {
        this.c = c;
    }
}
```

### Enable Configuration Properties

In the main Spring Boot class, annotate it with `@EnableConfigurationProperties` if it's not already enabled.

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
```

```
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

### Use the Configuration

You can now autowire the `KProperties` and `K1Properties` wherever needed.

```
java


import org.springframework.stereotype.Service;

@Service
public class MyService {
    private final KProperties kProperties;
    private final K1Properties k1Properties;

    public MyService(KProperties kProperties, K1Properties k1Properties) {
        this.kProperties = kProperties;
        this.k1Properties = k1Properties;
    }

    public void printProperties() {
        System.out.println("k.a: " + kProperties.getA());
        System.out.println("k.b: " + kProperties.getB());
        System.out.println("k.c: " + kProperties.getC());
        System.out.println("k1.c: " + k1Properties.getC());
    }
}
```

## 2. Using `@Value`

If the configuration structure is simple and you only need individual values, use `@Value`.

### Example

```
java


import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ConfigValues {

    @Value("${k.a}")
    private int a;

    @Value("${k.b}")
    private int b;

    @Value("${k.c}")
    private int c;

    @Value("${k1.c}")
    private int k1c;

    public void printValues() {
        System.out.println("k.a: " + a);
        System.out.println("k.b: " + b);
        System.out.println("k.c: " + c);
```

```
            System.out.println("k1.c: " + k1c);
        }
    }
```

### 3. Loading the Configuration File

Make sure `default.yml` is placed under `src/main/resources`.

Spring Boot automatically loads `application.yml` or `application.properties`. If you're using a custom file like `default.yml`, specify it in `application.yml`:

```yaml

spring:
  config:
    import: "classpath:default.yml"
```

Alternatively, you can use a profile-specific configuration (e.g., `application-default.yml`) and Spring will load it based on the active profile.

### 4. Running the Application

When you run the application, it will read the configuration values and populate them in the corresponding beans. You can now use these beans to access the configuration.

add the profile based for s3 and eventbridge

the pre check will run only in the case of non integration test profilr

To add profile-based configurations and ensure the pre-check runs only in non-integration test profiles for S3 and EventBridge in a Spring project, follow these steps:

---

### 1. Define Profile-Specific Configurations

`application.yml`

Define general application properties.

```yaml

spring:
  profiles:
    active: dev # Default profile
```

`application-dev.yml`

Define properties for the development profile.

```yaml
aws:
  s3:
    bucket-name: "dev-bucket"
  eventbridge:
    bus-name: "dev-event-bus"
```

**`application-integration-test.yml`**

Define properties for the integration test profile.

```yaml
aws:
  s3:
    bucket-name: "test-bucket"
  eventbridge:
    bus-name: "test-event-bus"
```

---

## 2. Conditionally Run Pre-Check Logic

Use Spring's **`@Profile`** annotation to ensure that pre-check logic runs only in specific profiles.

### Create the Pre-Check Runner

```java
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.actuate.health.HealthEndpoint;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Component;

@Component
@Profile("!integration-test") // Exclude this runner for the integration-test profile
public class PreCheckRunner implements ApplicationRunner {

    private final HealthEndpoint healthEndpoint;

    public PreCheckRunner(HealthEndpoint healthEndpoint) {
        this.healthEndpoint = healthEndpoint;
    }

    @Override
    public void run(ApplicationArguments args) {
        if (!"UP".equals(healthEndpoint.health().getStatus().getCode())) {
            throw new RuntimeException("Application failed to connect to AWS components.");
        }
    }
}
```

---

## 3. Profile-Specific Beans for S3 and EventBridge

You can conditionally create S3 and EventBridge beans based on the profile.

### AWS Client Configuration

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;

@Configuration
public class AwsClientConfig {

    @Bean
    @Profile("!integration-test") // Only create S3 client bean in non-integration-test prof
iles
    public S3Client s3Client() {
        return S3Client.builder().build();
    }

    @Bean
    @Profile("!integration-test") // Only create EventBridge client bean in non-integration-
test profiles
    public EventBridgeClient eventBridgeClient() {
        return EventBridgeClient.builder().build();
    }
}
```

---

## 4. Custom Health Indicators

Implement health checks for S3 and EventBridge only for non-integration-test profiles.

### S3 Health Indicator

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.s3.S3Client;

@Component
@Profile("!integration-test") // Enable only for non-integration-test profiles
public class S3HealthIndicator implements HealthIndicator {

    private final S3Client s3Client;

    public S3HealthIndicator(S3Client s3Client) {
        this.s3Client = s3Client;
    }

    @Override
    public Health health() {
        try {
            s3Client.listBuckets();
            return Health.up().withDetail("S3", "Available").build();
        } catch (Exception e) {
            return Health.down().withDetail("S3", "Not Available").withException(e).build();
        }
```

```
    }
}
```

### EventBridge Health Indicator

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;

@Component
@Profile("!integration-test") // Enable only for non-integration-test profiles
public class EventBridgeHealthIndicator implements HealthIndicator {

    private final EventBridgeClient eventBridgeClient;

    public EventBridgeHealthIndicator(EventBridgeClient eventBridgeClient) {
        this.eventBridgeClient = eventBridgeClient;
    }

    @Override
    public Health health() {
        try {
            eventBridgeClient.listEventBuses();
            return Health.up().withDetail("EventBridge", "Available").build();
        } catch (Exception e) {
            return Health.down().withDetail("EventBridge", "Not Available").withException
(e).build();
        }
    }
}
```

---

## 5. Activate Profiles

You can activate profiles via:

- **Command-line argument:** `--spring.profiles.active=integration-test`
- **Environment variable:** `SPRING_PROFILES_ACTIVE=integration-test`
- **Default in `application.yml`:**

```yaml
spring:
  profiles:
    active: integration-test
```

---

## 6. Testing

- **For development:** Start the application with the `dev` profile and ensure the pre-check runs.
- **For integration tests:** Start the application with the `integration-test` profile and ensure pre-checks and unnecessary beans are excluded.

This setup ensures that pre-check logic runs only in applicable environments, skipping the integration test profile as desired.