Great! Here's a **full working example** that:

1. Creates a **custom Spring Security JAR**.
2. Defines a **configuration properties class** to require an access-key.
3. Shows how a **downstream Spring Boot app**:
   * Uses this JAR.
   * Reads the key either from application.yml or from **Azure Key Vault**.
   * Fails at startup if the key is missing.

**🔐 1. Your Custom Spring Security JAR**

**✅ MySecurityProperties.java (in your JAR)**

package com.example.security;

import jakarta.validation.constraints.NotBlank;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.validation.annotation.Validated;

@Validated

@ConfigurationProperties(prefix = "my.security")

public class MySecurityProperties {

@NotBlank

private String accessKey;

public String getAccessKey() {

return accessKey;

}

public void setAccessKey(String accessKey) {

this.accessKey = accessKey;

}

}

❗️**Do NOT annotate this with @Component**. Let the downstream app decide when to enable it.

**🚀 2. Downstream Spring Boot App**

**✅ Main Class**

@SpringBootApplication

@EnableConfigurationProperties(MySecurityProperties.class)

public class DemoApp {

public static void main(String[] args) {

SpringApplication.run(DemoApp.class, args);

}

}

**✅ application.yml**

my:

security:

access-key: my-secret-access-key-from-config

Or **omit** this to test the validation failure on startup.

**🔒 3. Azure Key Vault Integration (Spring Cloud Azure)**

If the downstream app wants to load this access-key from Azure Key Vault instead of defining it in application.yml, configure like this:

**✅ Add to application.yml:**

spring:

cloud:

azure:

keyvault:

secret:

property-sources:

- endpoint: https://<your-keyvault-name>.vault.azure.net/

application:

name: demo-app

my:

security:

access-key: ${my-security-access-key}

In Azure Key Vault, create a secret named: my-security-access-key  
Spring maps my.security.access-key to that name automatically (Spring replaces . with -).

**💥 What Happens if the Key Is Missing?**

At startup, you'll see an error like:

Caused by: javax.validation.ConstraintViolationException: accessKey must not be blank

This is exactly what you want: a **fail-fast** behavior when a required config is missing.

**✅ Summary**

| **Component** | **Responsibility** |
| --- | --- |
| MySecurityProperties in JAR | Declares required config |
| @EnableConfigurationProperties(...) in app | Activates validation |
| application.yml or Azure Key Vault | Provides actual value |
| Spring Boot Validator | Fails startup if missing |

Let me know if you'd like this packaged as a starter (spring-boot-starter-security-custom) or want help deploying this on Azure!

Great question! Understanding the difference between **Full-Stack Monitoring** and **Application-Only Monitoring** in **Dynatrace** is key to choosing the right mode for your Kubernetes workloads like in Azure AKS.

**🧩 1. Full-Stack Monitoring (ClassicFullStack)**

**Dynatrace injects OneAgent into the Kubernetes nodes** and monitors everything running on them.

**✅ What it does:**

* Automatically injects OneAgent into **every pod**
* Monitors **infrastructure**, **container runtime**, and **application**
* Tracks **processes, CPU/memory/network**, **logs**, and **deep code-level tracing**
* Provides **full-stack visibility** from Node ➝ Pod ➝ App ➝ Database/API

**🛠️ Use Case:**

* When you want deep observability, like:
  + **Auto-instrumentation of Java, Node.js, .NET, etc.**
  + Full topology mapping and dependency tracing
  + Metrics + logs + traces in one place

**🧠 Example Features:**

* Davis AI detects memory leaks, abnormal CPU
* Full **service flow and backtrace**
* Smartscape view includes infrastructure

**🔴 Cons:**

* Slightly **higher overhead**
* Requires **OneAgent to run as a privileged DaemonSet** or with injected sidecars

**🎯 2. Application-Only Monitoring (ApplicationMonitoring)**

**Only monitors your application layer**. No infrastructure visibility.

**✅ What it does:**

* Injects OneAgent **only into application pods**
* Focuses on **code-level telemetry** (like traces, errors, requests)
* Skips monitoring of:
  + Node metrics
  + OS-level metrics
  + Container runtime
* Typically uses **OneAgent Operator** with **applicationMonitoring mode**

**🛠️ Use Case:**

* You only need APM (Application Performance Monitoring)
* You’re running in restricted environments:
  + Managed Kubernetes (no access to host/node)
  + FaaS platforms (like Azure Functions)
* Lightweight scenarios

**🧠 Example Features:**

* Request/response tracing
* Service-level monitoring
* Supports modern APM without node access

**🟡 Cons:**

* **No infrastructure visibility**
* No container or node metrics
* Less useful for root cause analysis involving system-level issues

**📋 Summary Table**

| **Feature** | **Full-Stack Monitoring** | **Application-Only Monitoring** |
| --- | --- | --- |
| **OneAgent location** | On every node and injected into pods | Only injected into app pods |
| **Infrastructure metrics** | ✅ Yes | ❌ No |
| **Pod/container monitoring** | ✅ Yes | ❌ No |
| **Auto code instrumentation** | ✅ Yes | ✅ Yes |
| **Overhead** | Higher | Lower |
| **Best for** | Full observability | App-only APM (e.g., in PaaS) |

**🔧 Configuration in Dynakube YAML**

# Full-stack mode

spec:

oneAgent:

classicFullStack: {}

# App-only mode

spec:

oneAgent:

applicationMonitoring: {}

Let me know which mode suits your AKS use case (dev, stg, perf) and I can help you tailor the setup!