



Attack Surface Service

Ofir Yakovian

< CY-22/H2 >

BACKGROUND

Orca Security delivers the industry-leading Cloud Security Platform that identifies, prioritizes, and remediates security risks and compliance issues across various cloud estates, spanning AWS, Azure, GCP and K8s.

The Orca Platform connects to a given cloud environment and provides complete coverage across all cloud risks – spanning misconfigurations, vulnerabilities, identity, data security, and advanced threats. In that matter, a fundamental platform capability is to scan the tenant environment and identify the potential attack vectors risking their cloud assets (denoted as Attack Surfaces).

The **Breacher** service forms a backbone of the aforementioned business initiative. With Breacher, tenants can easily subscribe and analyze the attack surface of a specified cloud asset - meaning which other virtual machines can access and potentially attack it - while eliminating performance, compatibility, and security gaps.

ARCHITECTURAL DESIGN

Breacher core design guidelines adhere to modern **Spring Boot Cloud Native** applications. The Spring Framework offers a Dependency Injection (DI) feature, that lets components define their own dependencies, later injected at runtime by that the Spring Container. This enables modular applications consisting of loosely coupled components that are ideal for microservices applications, following the **Inversion of Control** (Ioc) principle.

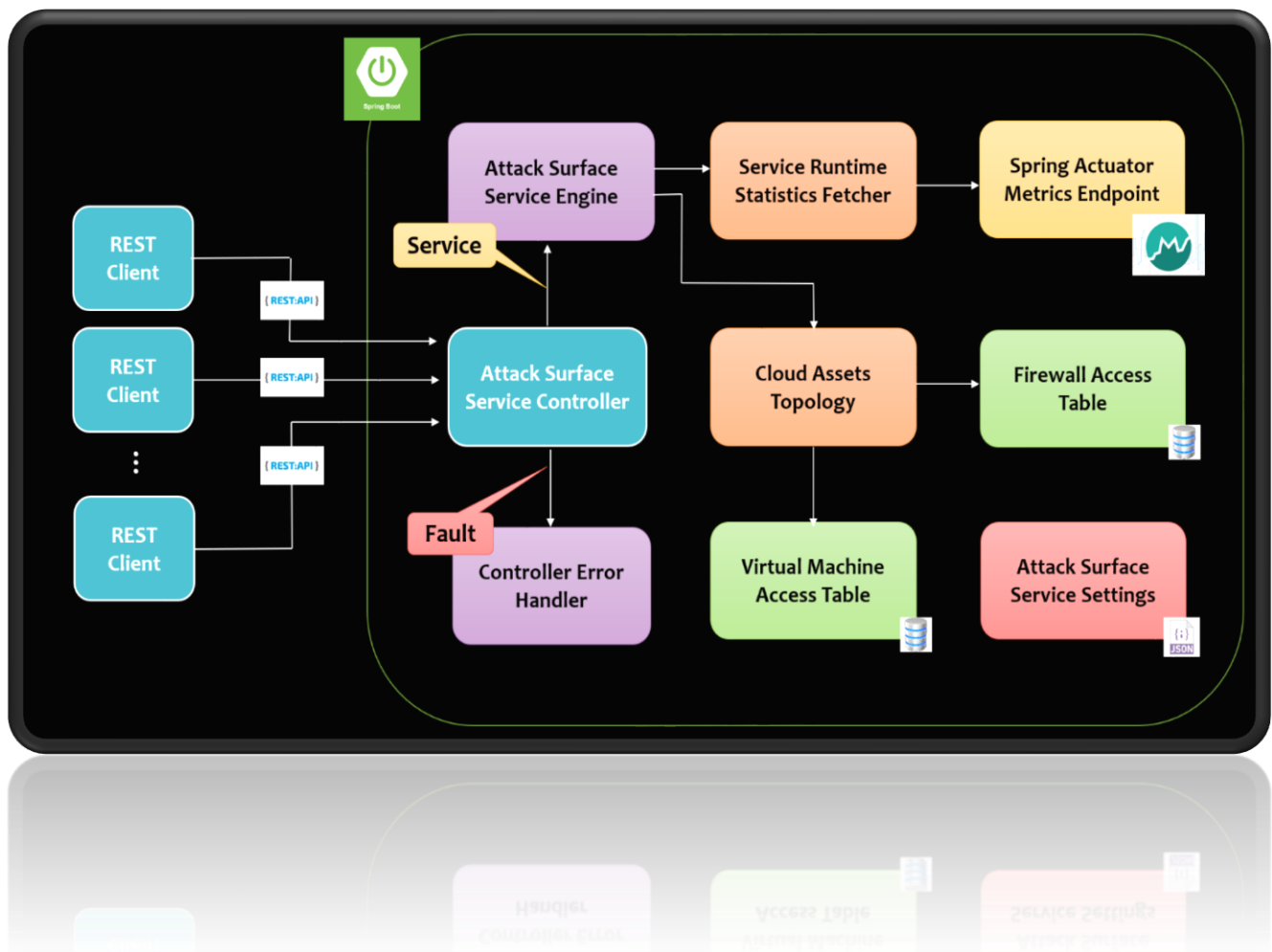
Spring Framework also offers built-in support for typical tasks that an application needs to perform, such as data binding, type conversion, validation, exception handling, resource and event management, internationalization, and much more. As capable and comprehensive as Spring Framework is, it still requires significant time and knowledge to configure, set up, and deploy applications. **Spring Boot** mitigates this effort.

Spring Boot helps Breacher to become a standalone shipped service that runs on its own, without relying on an external web server, by embedding **Tomcat** into our application during the initialization process. As a result, we can launch our service on any platform by simple activation.

As a Microservice, Breacher is developed and deployed easily, achieving a lightweight decoupled model of simple scalability, out-of-the-box compatibility with containers, minimum configuration, and faster time-to-value cycles.

1. Design Overview

The following diagram illustrates the architectural components of Breacher.



Breacher accepts a single command line argument representing the cloud topology of a tenant. As such, we assume that the tenant's cloud environment is immutable. That is, of course, a non-realistic assumption that tradeoffs for simplicity. We further present the (non-heavy) architectural changes implied by omitting this assumption in the "future development cycles" section at the end of this document.





The service lifecycle begins with analyzing the provided cloud environment. The entire JSON deserialization is housed within the [AttackSurfaceServiceSettings](#) component. For test purposes, cloud environment configurations can also be provided with a system property (to eliminate I/O operations along testing cycles). This component is responsible to diagnose the service launching mode, verify and load the cloud environment structure into memory, and interpret the provided Virtual Machines and Firewall Rules. Huge cloud environments might flood the managed heap, and so Space vs. Time tradeoffs are shortly discussed onwards. For the time being, this component is also responsible for decoupling all other components from the official schema, and appropriately handle forward and backward compatibility (FCG & BCG).





The entire dependency injection lifecycle is triggered via the [AttackSurfaceServiceController](#) component. It leverages the Spring `@RestController` annotation, which is a convenient annotation to mark a component as a REST Request Handler. It is used to establish the core RESTful Web Service using Spring MVC and take care of mapping Request Data to Request Handler Methods, on a per-endpoint basis (`/attack`, `/stats`). Once response body is generated from the handler method, it converts it to a JSON automatically, reducing compatibility requirements on API edges. Furthermore, it centralizes the entire error handling into our [ControllerErrorHandler](#) component. As such, the service encapsulates any possibly generated exception with the [InternalRestEndpointException](#) proprietary one, which aims to return a well-structured faulting HTTP response back.

Later on, Spring IoC injects the [AttackSurfaceServiceCoreEngine](#) into the former core Controller. The engine is responsible to orchestrate the actual business logic housed within the service, and forms as an ultimate bridge downstream. Henceforth, down level components need not be familiar with external API knowledge. They can rather communicate back and forth, with fully structured information used only internally – being agnostic to API version or scheme changes.

Since error handling occurs at the top layer of our service, it also eliminates exception propagation concerns from down-level components. Ultimately, it orchestrates the following components at the business layer:

- I. [CloudAssetsTopology](#) – Responsible to fabricate and maintain an inverse hash lookup table, mapping a particular cloud asset with its attack surface. This allows an O(1) handling of /attack endpoint calls. It holds the [VirtualMachineAccessTable](#) which illustrates the various cloud assets topology, intersecting the various Firewall access rules, provided by the corresponding [FirewallAccessTable](#).
- II. [RuntimeStatisticsFetcher](#) – Responsible to query and collect HTTP runtime statistics maintained by the Spring Actuator. They are provided by a [MetricsEndpoint](#) being a simple, in-memory backend that is automatically used to subscribe for selected metric updates in runtime. It provides a thin (yet robust) thread-safe store, providing strong concurrency guaranty for simultaneous REST handling threads.

 /api/v1/attack [GET]	 com.orca.spring.breacher.controller.AttackSurfaceServiceController
 /api/v1/stats [GET]	<pre>@GetMapping(AttackSurfaceServiceConstants.RestControllerEndpointMappingAttack) public List<String> analyzeMachineAttackSurface(@RequestParam(name = AttackSurfaceServiceConstants.RestControllerEndpointAttackQueryParam) String machineIdentifier) throws VirtualMachineNotFoundException, InternalRestEndpointException</pre>
	Throws: VirtualMachineNotFoundException InternalRestEndpointException
	 breacher

 /api/v1/attack [GET]	 com.orca.spring.breacher.controller.AttackSurfaceServiceController
 /api/v1/stats [GET]	<pre>@GetMapping(AttackSurfaceServiceConstants.RestControllerEndpointMappingStats) public ServiceRuntimeStatistics fetchServiceRuntimeStatistics() throws InternalRestEndpointException</pre>
	Throws: InternalRestEndpointException
	 breacher

2. Space vs. Time Tradeoffs

As stated before, we load the entire tenant's cloud configuration into memory, and therefore Cloud topologies at scale might lead to managed heap flooding, causing poor performances or severe downtimes.

On the one hand, by maintaining the inverse hash lookup table, we can easily associate a cloud Virtual Machine to its attack surface, solving the task in $O(1)$ time complexity, which quantifies the amount of time taken by the REST service to run. On the other hand, for N Virtual Machines and M Firewall Rules, the space complexity is $O(N + M)$, which quantifies the amount of space taken by the REST service to function.

To enjoy the best of both worlds, we might spare the cloud environment JSON loading in two phases, while the later phase depends on the former one:

Phase 1: JSON Streaming - This allows Breacher to begin viewing the JSON data, and prevents the JVM from having to store large files all at once. Data can come and go from the JVM as it is processed and stored. Data streams directly encounter the CAP theorem, but our basic assumption of immutable cloud environments spares that for now. Generally, the data will need to be in order, and needs to get stored somewhere, and for that we might use an external Redis Engine.

Phase 2: Redis Engine – This allows Breacher to use an external data structure store, used as a database, cache, and streaming engine. JSON streaming might easily be achieved by only loading the relevant JSON portion into memory. Practically, it might also speed the attack surface analysis, by providing an LRU cache, which lets it automatically evict old data as new comes.

Of course, this violates the “Keep It Simple Stupid” (KISS) design guideline. Simplification for current and future cooperative development of the service, from my perspective, is nothing but the skillful application of SOLID principles.

Furthermore, it creates another pain-point for the entire software lifecycle, as we now have to handle fault-tolerance in cases where the Redis Engine goes down. It also defers our knowledge of viability for the given JSON document, as we process it on a per-portion basis. Last but not least, it complicates the deployment procedure, and might increase the overall used COGS – as cloud-based storage is quite expensive.

3. Backward & Forward Compatibility

Breacher is a customer-facing application, and has to agree on the structure of a tenant environment. Without a schema, the only information we might handle back and forth would be an ordered bag of values, and those values are meaningless without context.

Coming up with an initial agreement on the fields and types of data is relatively straightforward. However, other versions of our service will eventually need to change the data to support new features or remove unsupported ones. That is, the underlying schema should evolve.

Currently, the Schema backing the service is present at [/schema/CloudEnvironment.json](#). It houses definitions of all types, and any context needed to understand the tenant environment. To back the schema with actual Plain Old Java Objects (POJOs), we use the famous [jsonschema2pojo](#) project. As a pre-build step, prior to actual compilation phase, it hooks the Maven pipeline to generate corresponding POJOs, to be later used by the [AttackSurfaceServiceSettings](#) component.

```

<plugin>
  <groupId>org.jsonschema2pojo</groupId>
  <artifactId>jsonschema2pojo-maven-plugin</artifactId>
  <version>${jsonschema2pojo.version}</version>
  <configuration>
    <includeToString>true</includeToString>
    <annotationStyle>jackson2</annotationStyle>
    <initializeCollections>true</initializeCollections>
    <includeHashCodeAndEquals>true</includeHashCodeAndEquals>
    <includeAdditionalProperties>false</includeAdditionalProperties>
    <outputEncoding>${project.build.sourceEncoding}</outputEncoding>
    <targetPackage>${project.groupId}.breacher.model</targetPackage>
    <sourceDirectory>${basedir}/src/main/resources/schema</sourceDirectory>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

These auto-generated POJOs enforce build break upon schema changes, reflecting compatibility impacted components at compile-time rather than runtime. All files under the auto-generated package (*com.orca.spring.breacher.model*) are GIT-ignored, to prevent committing them whatsoever. For JSON here is an incomplete list of backward-compatible changes:

1. **Adding a field with a default value.** Older writers will be unaware of that field so the default value will be used instead.
2. **Adding an optional field.** Older writers will be unaware of that field so null will be used instead.
3. **Removing a field.** Newer readers will ignore whatever was previously written in this field.

NON-FUNCTIONAL REQUIREMENTS (NFR)

1. Logging

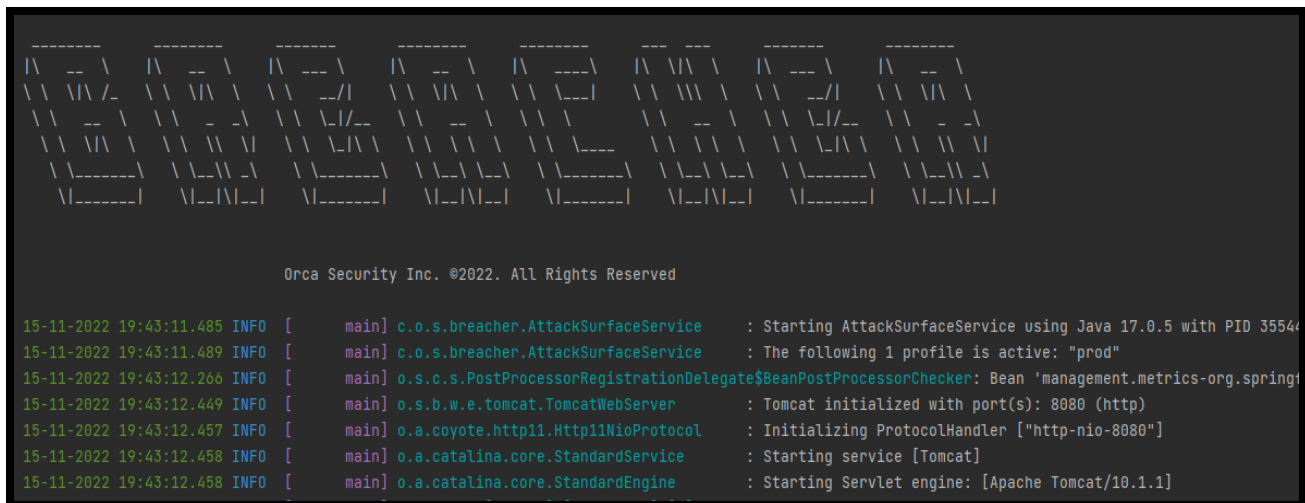
Breacher uses [Logback](#) as the logging infrastructure. Logback is a fast and reliable logging solution widely used in top-of-the-line cloud native projects. It also serves as a simple facade or abstraction for a variety of logging frameworks (including Log4j), hence no matter what logging framework is used beneath – future versions could enjoy use the same interface.

Logback integrates with Servlet containers, such as Tomcat and Jetty, to provide HTTP-access log functionality. Breacher enjoys that by building the default module on top of Logback-Core.

To ease debuggability and meet high customers serviceability needs, Breacher logs helpful key events as persistent information, to a file system backing store, in production-like environments. Along development pipelines, traces are also logged to the application console.

The entire Logback configuration for our service can be found in [logback-spring.xml](#). This configuration features:

1. Output logs to the console when application is not run with production profile. Furthermore, it output logs to a backing file, regardless of the selected profile.
2. All file logs are gathered in the `/logs` directory on the root level of the service. Log file naming convention is “breacher-`${date}`.log”. We use a daily rollover policy, or whenever the file size reaches 64 MB.
3. Logger is configured in a way that it’s thread safe. Multiple appenders can write to the same log file, with high concurrency, using to the “prudent” option. As we safely log to the same file from multiple JVMs, it slightly degrades performance.
4. The same pattern convention is used in all Log appenders: [Date & Time], [Thread name], [Source Class], [Log Level] and [Message].



2. Monitoring

Breacher uses [Actuator](#) to monitor and manage the service, by providing production-ready features like health checks, auditing, metrics gathering, HTTP tracing etc. All these features can be accessed over HTTP endpoints. Actuator also integrates with external application monitoring systems like Prometheus, Graphite, DataDog, Influx, Wavefront, New Relic and many more. These systems provide dashboards, graphs, analytics, and alarms to help monitor and manage the service from one unified interface.

Actuator uses [Micrometer](#), an application metrics facade to integrate with these external application monitoring systems. This makes it super easy to plug-in any application monitoring system with very little configuration.

Breacher exposes several endpoints to monitor and interact with it. Along development cycles, all possible Actuator endpoints are exposed, and can easily accessed via `/actuator`. At production-like environments, the service only includes a few selected built-in endpoints, such as “health”, “info”, and “metrics”. One can easily control and expose more endpoints by appropriately enriching the `application-prod.properties` file.

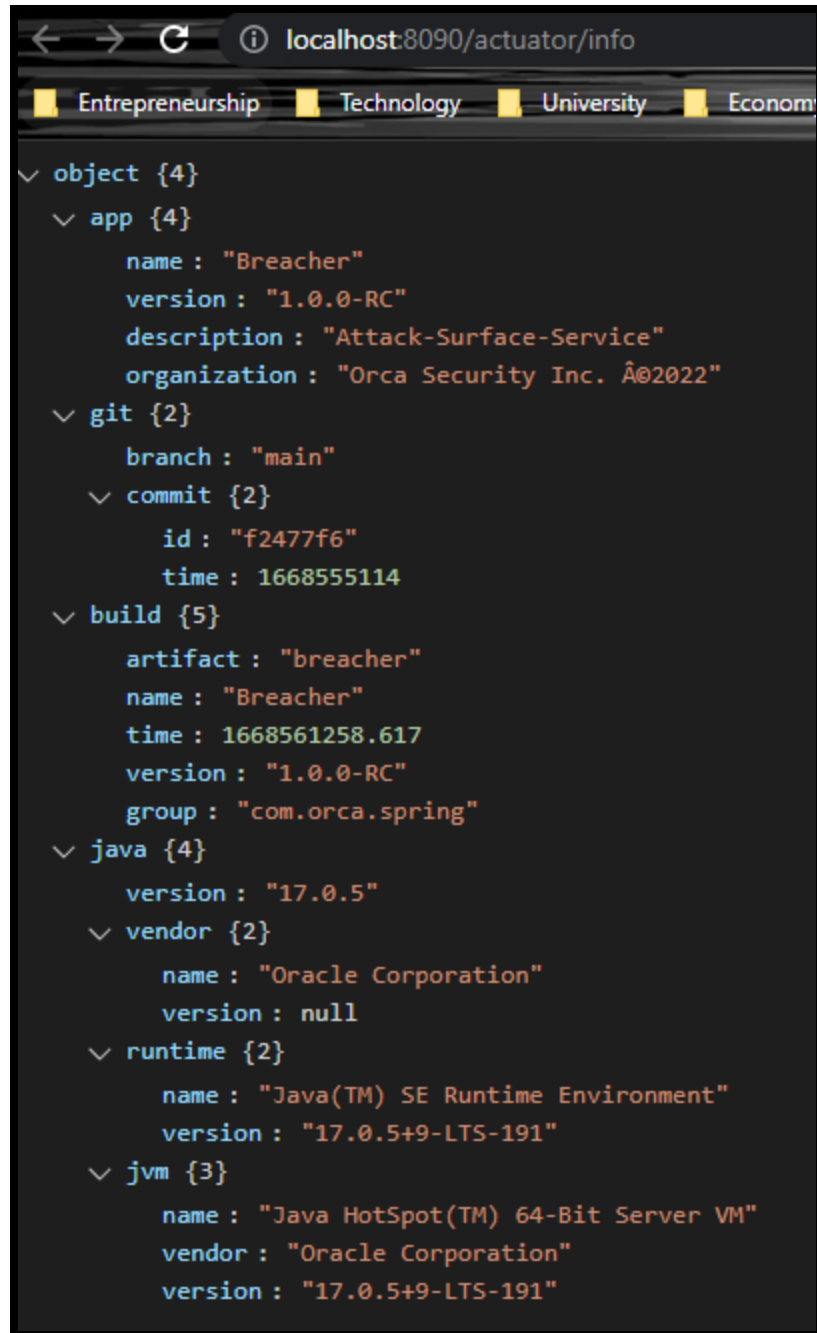
Next, we present the currently exposed endpoints.

1. Health – Mapped to [/actuator/health](#). You can use health information to check the status of the running service. It can easily integrate to any monitoring software to alert someone when a production defect arises. The information exposed by the health endpoint depends on the HTTP status code in the response, which reflects the overall health status. In general, the status will be **UP** if the application is healthy, and **DOWN** if the application gets unhealthy due to any issue.

Health Status	HTTP Mapping
UP	HTTP status 200 (OK)
DOWN	HTTP status 503 (SERVICE_UNAVAILABLE)
OUT_OF_SERVICE	HTTP status 503 (SERVICE_UNAVAILABLE)
UNKNOWN	No default mapping - resolves to UP.

2. Info – Mapped to [/actuator/info](#). Adding useful information helps to quickly identify the build artifact name, version, organization, etc. It could come in handy to check if an appropriate version of the service was deployed (using the accurate GIT commit information, for instance). Breacher currently exposes the following:

Information ID	Usage
Build	Exposes build information.
Env	Exposes Environment chosen properties.
Git	Exposes Git related information.
Java	Exposes Java runtime information.



```
localhost:8090/actuator/info
Entrepreneurship Technology University Economy

object {4}
  app {4}
    name : "Breacher"
    version : "1.0.0-RC"
    description : "Attack-Surface-Service"
    organization : "Orca Security Inc. Â©2022"
  git {2}
    branch : "main"
    commit {2}
      id : "f2477f6"
      time : 1668555114
  build {5}
    artifact : "breacher"
    name : "Breacher"
    time : 1668561258.617
    version : "1.0.0-RC"
    group : "com.orca.spring"
  java {4}
    version : "17.0.5"
    vendor {2}
      name : "Oracle Corporation"
      version : null
    runtime {2}
      name : "Java(TM) SE Runtime Environment"
      version : "17.0.5+9-LTS-191"
    jvm {3}
      name : "Java HotSpot(TM) 64-Bit Server VM"
      vendor : "Oracle Corporation"
      version : "17.0.5+9-LTS-191"
```

3. Metrics – Mapped to [/actuator/metrics](#). Breacher provides automatic meter registration for a wide variety of technologies. In most situations, the defaults provide sensible metrics that can be published to various monitoring systems. It displays a list of available meter names, that can be furthered explored to view information about a particular meter by providing its name as a selector.

Among other metric endpoints, Breacher enables and heavily relies on the built-in HTTP tracing, providing a deep observability solution. You can further inspect the HTTP endpoint to obtain information about the REST request-response exchanges, by pinpointing the “**http.server.requests**” information, as follows:



```
localhost:8090/actuator/metrics/http.server.requests
Entrepreneurship Technology University Economy Heal

object {5}
  name: "http.server.requests"
  description: null
  baseUnit: "seconds"
  measurements [3]
    0 {2}
      statistic: "COUNT"
      value: 13
    1 {2}
      statistic: "TOTAL_TIME"
      value: 0.8458536
    2 {2}
      statistic: "MAX"
      value: 0.0403423
  availableTags [6]
    0 {2}
    1 {2}
      tag: "method"
      values [1]
        0: "GET"
    2 {2}
    3 {2}
      tag: "uri"
      values [7]
        0: "/actuator/metrics/{requiredMetricName}"
        1: "/actuator"
        2: "/actuator/info"
        3: "/api/v1/stats"
        4: "/actuator/metrics"
        5: "/*"
        6: "/api/v1/attack"
    4 {2}
    5 {2}
```

This useful information is collected in runtime by the /stats REST endpoint, to collect REST API statistics such as total request count and average processing time.

3. Availability

When deployed on Cloud Native providers, Breacher can provide information about its availability, using infrastructure such as Kubernetes Probes. Spring Boot includes out-of-the box support for the commonly used “liveness” and “readiness” availability states. Since Breacher uses Spring Boot’s “actuator” support, then these states are exposed as health endpoint groups.

1. **Liveness State** - A broken “Liveness” state means that the application is in a state that it cannot recover from, and the infrastructure should restart the application. The internal state of Spring Boot applications is mostly represented by the Spring ApplicationContext. If the application context has started successfully, Spring Boot assumes that the application is in a valid state. An application is considered live as soon as the context has been refreshed, and so Breacher refreshes it on a per /stats or /attack REST query.
2. **Readiness State** – A failing “Readiness” state tells the platform that it should not route traffic to the application for now. This typically happens while CommandLineRunner is being processed, or at any time if the application decides that it is too busy for additional traffic. An application is considered ready as soon as application and command-line runners have been called. We implement that interface as part of our [AttackSurfaceService](#) entry point.

```
@Slf4j
@SpringBootApplication
public class AttackSurfaceService implements CommandLineRunner
{
    Dependencies

    ▲ Ofir Yakovian
    public static void main(String[] args)
    {
        SpringApplication.run(AttackSurfaceService.class, args);
    }

    new *
    @Override
    public void run(String... args) throws Exception
    {
        log.info("[Breacher]: Successfully initialized. [Application Context -> ({})]", applicationContext);
    }
}
```

QUALITY ASSURANCE

Breacher is composed of multiple components. A system feature largely depends on each component to function and interact correctly with each other. To keep confidence, we have tested each component individually to ensure each it is working appropriately.

The earlier we can identify a defect in the development life cycle, the lower cost to fix it. Hence, we widely use the “Shift Left” testing approach, with leveraging the unit and integrating tests as formal PR gates.

Both UTs and ITs follow the AAA (*Arrange-Act-Assert*) pattern, that divides a particular test method into three sections, each of them only responsible for the part in which it is named after. Thus, the **Arrange** section setups a specific test (creates objects and establishes mocks, if any). The **Act** section invokes the method being tested. Finally, **Assert** simply checks whether the expectations were met.

Formal **Software Test Plan** (STP) is as follows:

Test Suite	Test Case	Expected Result	Status
Service Application Context Tests	Context Loads	Spring IoC Container should be up and running.	OK
Service Cloud Environment Sample 0	Test Analyze Machine Attack Surface - Sanity Sample 0	Load Sample #0 (input-0.json) and validate a randomly chosen attack surface.	OK
Service Cloud Environment Sample 1	Test Analyze Machine Attack Surface - Sanity Sample 1	Load Sample #1 (input-1.json) and validate there are no available attack surfaces.	OK

Service Cloud Environment Sample 2	Test Analyze Machine Attack Surface - Sanity Sample 2	Load Sample #2 (input-2.json) and validate a randomly chosen attack surface.	OK
Service Cloud Environment Sample 3	Test Analyze Machine Attack Surface - Sanity Sample 3	Load Sample #3 (input-3.json) and validate a randomly chosen attack surface.	OK
Service Cloud Environment Toy Sample	Test Analyze Machine Attack Surface - Sanity Custom Sample	Create at runtime a custom cloud environment sample which consists of two attacking groups (Tag1, Tag2) and a single victim group (Tag3). Add appropriate FW rules and verify the resulting attack vectors.	OK
Service Rest Controller Endpoint Tests	Test Engine Dependency Injection for Controller – Smoke Test	Mock the fundamental engine for the REST controller and verify the DI has been done accordingly	OK
	Test Analyze Machine Attack Surface – Mock Basic Scenario	Mock the /attack REST endpoint with a valid VM identifier. Verify the resulting attack surface.	OK
	Test Analyze Machine Attack Surface – Mock Error Handling Invalid Asset Identifier	Mock the /attack REST endpoint with an invalid VM identifier. Expect a - <i>VirtualMachineNotFoundException</i>	OK
	Test Analyze Machine Attack Surface –	Mock the /attack REST endpoint with a runtime unchecked exception. Expect a - <i>InternalRestEndpointException</i>	OK

	Mock Error Handling Internal REST Failure		
	Test Fetch Service Runtime Statistics - Mock Basic Scenario	Mock the /stats REST endpoint. Verify the resulting service statistics.	OK
	Test Fetch Service Runtime Statistics – Mock Error Handling Internal REST Failure	Mock the /stats REST endpoint with a runtime unchecked exception. Expect a - <i>InternalRestEndpointException</i>	OK
Service Rest Controller HTTP Request Tests	Test Analyze Machine Attack Surface – Sanity HTTP Request Response	Listen for a connection (as it would do in production), and then send a valid /attack HTTP request. Assert the response.	OK
	Test Analyze Machine Attack Surface – Error HTTP Request Response	Listen for a connection (as it would do in production), and then send an invalid /attack HTTP request. Assert the proprietary <i>Controller Error Response</i> structure.	OK
Service Rest Controller Web Layer Tests	Test Analyze Machine Attack Surface – Sanity HTTP Request Response	Do not start the server at all, but only test the layer below that, where POJOs hand off to our REST controller (mocking MVC). Assert the replied attack surface.	OK
	Test Analyze Machine Attack Surface – Error HTTP Request Response	Do not start the server at all, but only test the layer below that, where POJOs hand off to our REST controller (mocking MVC). Assert the replied HTTP status.	OK

FUTURE DEVELOPMENT CYCLES

1. **Adhere to more NFRs** - Considering the non-functional requirements during a project is critical to understand what is being built. There are many more that we might consider implementing, such as **Scalability** (the ability for the system to scale out horizontally or vertically), **Recoverability** (recovering from failures), **Security** (authentication and secret management), **Deployability** (requirements around infrastructure provisioning), **Compliance**, **Backups / D/R** (what should be backed up for disaster recovery), etc.
2. **Implement the Redis Engine based solution** – Time vs. Space tradeoff is at the core of our service implementation. Moving towards a Redis Engine based solution (or any other external data store providing an LRU cache) might significantly boost overall robustness of our solution, while meeting performance requirements.
3. **Leverage the Kubernetes Cloud Native orchestrator** – Shipping Breacher to K8s is as easy as integrating with Spring Cloud Kubernetes. It provides implementations of well-known Spring Cloud interfaces, allowing to build and run Spring Cloud applications on Kubernetes easily. At first, we need to define a docker image for this service, and K8s Deployment and Service manifests. This would provide out-of-the-box support for most of the NFRs listed on (1).
4. **Supporting on-the-fly changes of cloud environments** – We had the assumption that tenants' cloud environments are immutable. This is of course non-realistic. Getting rid of it with a Redis Engine based solution is quite straightforward, as we just need to evict corresponding Virtual Machines affected by configuration changes. A complication arise with thread synchronization. Currently, as the environment doesn't change, all threads can access it in a lock-free manner (since they are just *readers*). If the environment does change, *writers* come in, and might cause "dirty reads" of cloud assets, possibly reporting wrong attack surfaces to customers. Providing a *readers-writers* lock is complicated without incurring performance degradations.