RED HAT
DEVELOPERS

Using Microprofile API's to build cloud native applications

Heiko Braun
Principal Software Engineer
Red Hat

Oct 2017

Word of caution

What to expect from this talk

It’s rather broad, not necessarily deep

- Aims at providing an overview of the API and spec in the MicroProfile 1.2 release
  - Only ~45 min
- Connect those specs and API’s with architectural purpose
  - Where, how and why would I use that?
- Touches on a wide range of topics, but cannot explain everything in great detail
  - Should leave you with an idea what to followup 📖upon
  - Reading tips, further resources, etc (Watch for)
- Happy to chat about all of this later on

The context: Cloud native

The CNCF Definition

https://www.cncf.io/

Cloud native computing uses an open source software stack to be:

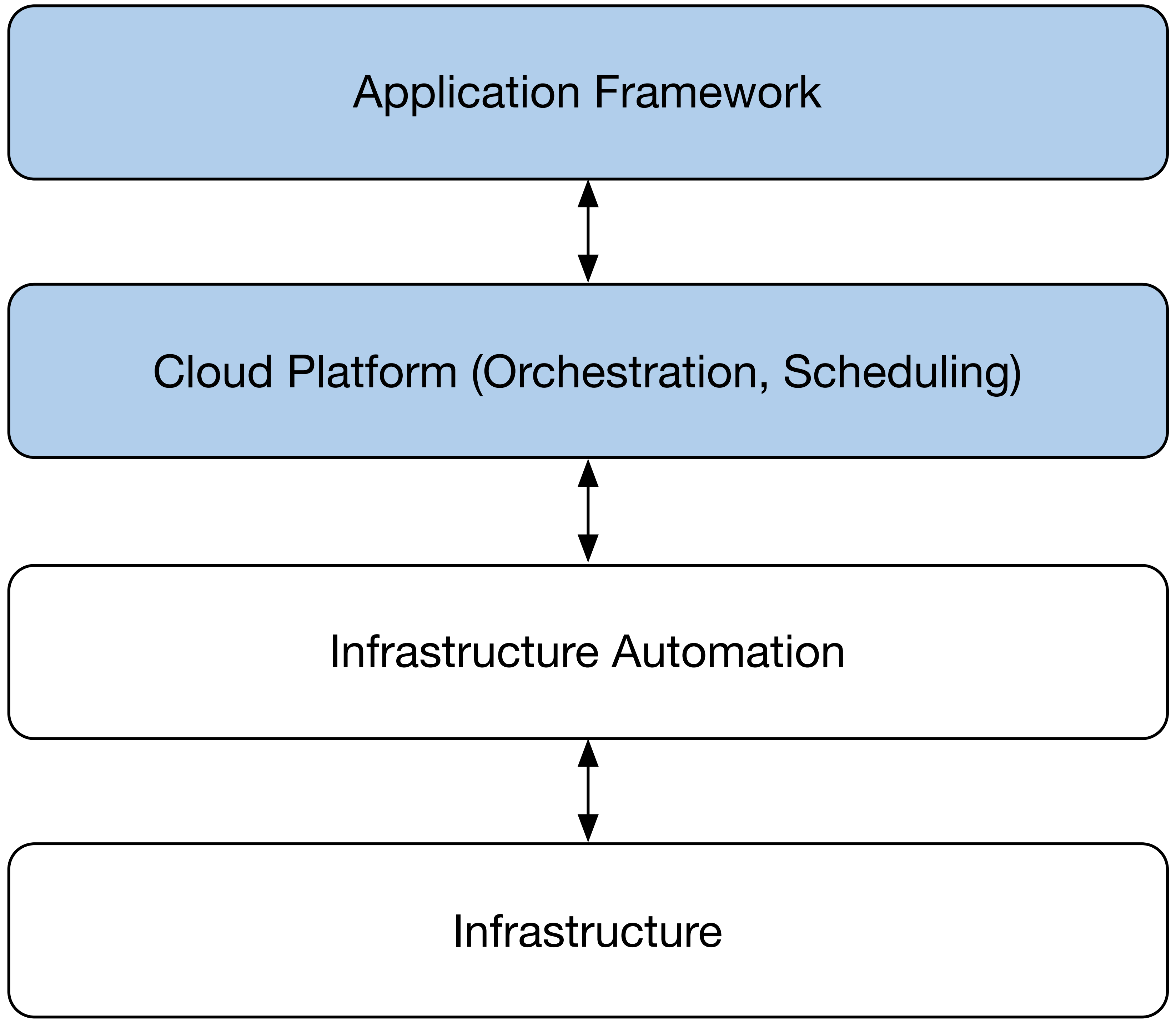
- **Containerized**:
  - Each part (applications, processes, etc) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.
- **Dynamically orchestrated**:
  - Containers are actively scheduled and managed to optimize resource utilization.
- **Microservices oriented**:
  - Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications.

Cloud native development

Focus areas

Relevant to this discussion:

- Contracts between application and cloud platform
  - i.e. programming models, etc
- Common cloud platform concepts and their impact
  - i.e what does containerised and scheduled mean? What's the impact?
- Overarching architectural concerns
  - Service oriented architectures, distributed systems, etc


Application Framework
Cloud Platform (Orchestration, Scheduling)
Infrastructure Automation
Infrastructure

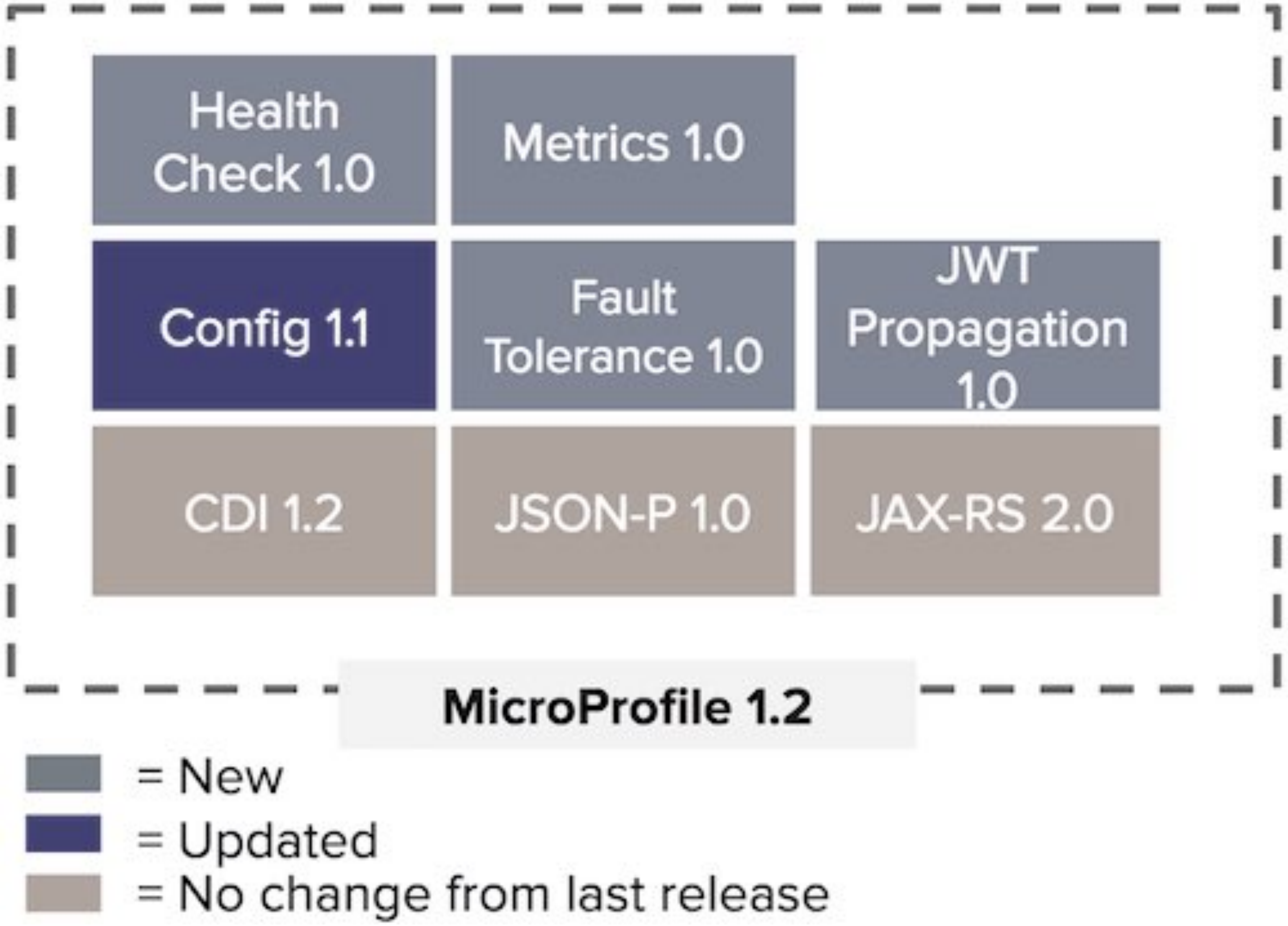
https://thenewstack.io/kubernetes-an-overview/

Eclipse MicroProfile

Optimising Enterprise Java for Microservice Architectures

It’s an open forum to collaborate on enterprise Java™ microservices

- https://microprofile.io
- Version 1.2 released recently
- Now includes
  - Health Check 1.0
  - Metrics 1.0
  - Fault Tolerance 1.0,
  - JWT Propagation 1.0
  - Config 1.1
- In addition to CDI 1.2, JSON-P 1.0 and JAX-RS 2.0.



Health Check 1.0

Metrics 1.0

Config 1.1

Fault Tolerance 1.0

JWT Propagation 1.0

CDI 1.2

JSON-P 1.0

JAX-RS 2.0

MicroProfile 1.2

▨ = New

■ = Updated

□ = No change from last release

https://projects.eclipse.org/projects/technology.microprofile/releases/microprofile-1.2

Configuration

Application configuration

Because everybody needs it

Rationale

- Configuration without the need to repackage the whole application binary
  - Aggregate configuration from **many different ConfigSources** into **a single, merged view**
- Bundle default configuration with the application, but allow overrides from outside
- Common sense configuration sources and and the ability to bring our own
- Support for dynamic and static configuration values

https://**github.com/eclipse/microprofile-config**/releases

Configuration API 1/2

Accessing configuration values

- Main entry point is the *ConfigProvider*
- Internally values are String/String tuples
- Converters do the type conversion

```
public class ConfigUsageSample {
  public void useTheConfig() {
    // get access to the Config instance
    Config config = ConfigProvider.getConfig();
    String serverUrl =
config.getValue("acme.myprj.some.url", String.class);
    callToServer(serverUrl);
  }
}
```

```
$> java -jar some.jar \
 -Dacme.myprj.some.url=http://other.server/other/endpoint
```

Default ConfigSources

Ordinals guide precedence

A Microprofile-Config implementation must provide ConfigSources for the following data out of the box:

- System properties (default ordinal=400)
- Environment variables (default ordinal=300)
- A ConfigSource for each property file META-INF/microprofile-config.properties found on the classpath. (default ordinal = 100)

Extension points

Bring your own ConfigSource

- Discovery using the *java.util.ServiceLoader* mechanism.
- Implementation of *ConfigSource* interface.
- For dynamic sources you can you can also register a *ConfigSourceProvider*

```
public class CustomDbConfigSource implements ConfigSource {
  @Override
  public int getOrdinal() {
  return 112;
  }
  @Override
  public Set<String> getPropertyNames() {
  return readPropertyNames();
  }
  @Override
  public Map<String, String> getProperties() {
  return readPropertiesFromDb();
  }
  @Override
  public String getValue(String key) {
  return readPropertyFromDb(key);
  }
  @Override
  public String getName() {
  return "customDbConfig";
  }
}
```

Config API 2/2

CDI based: @Inject @ConfigProperty

- Supports type conversion
- Access either *Config* instance or value
- Can treat missing values as failures or not (*Optional<T>*)
- Distinction between static and dynamic values (through *Provider<T>*)

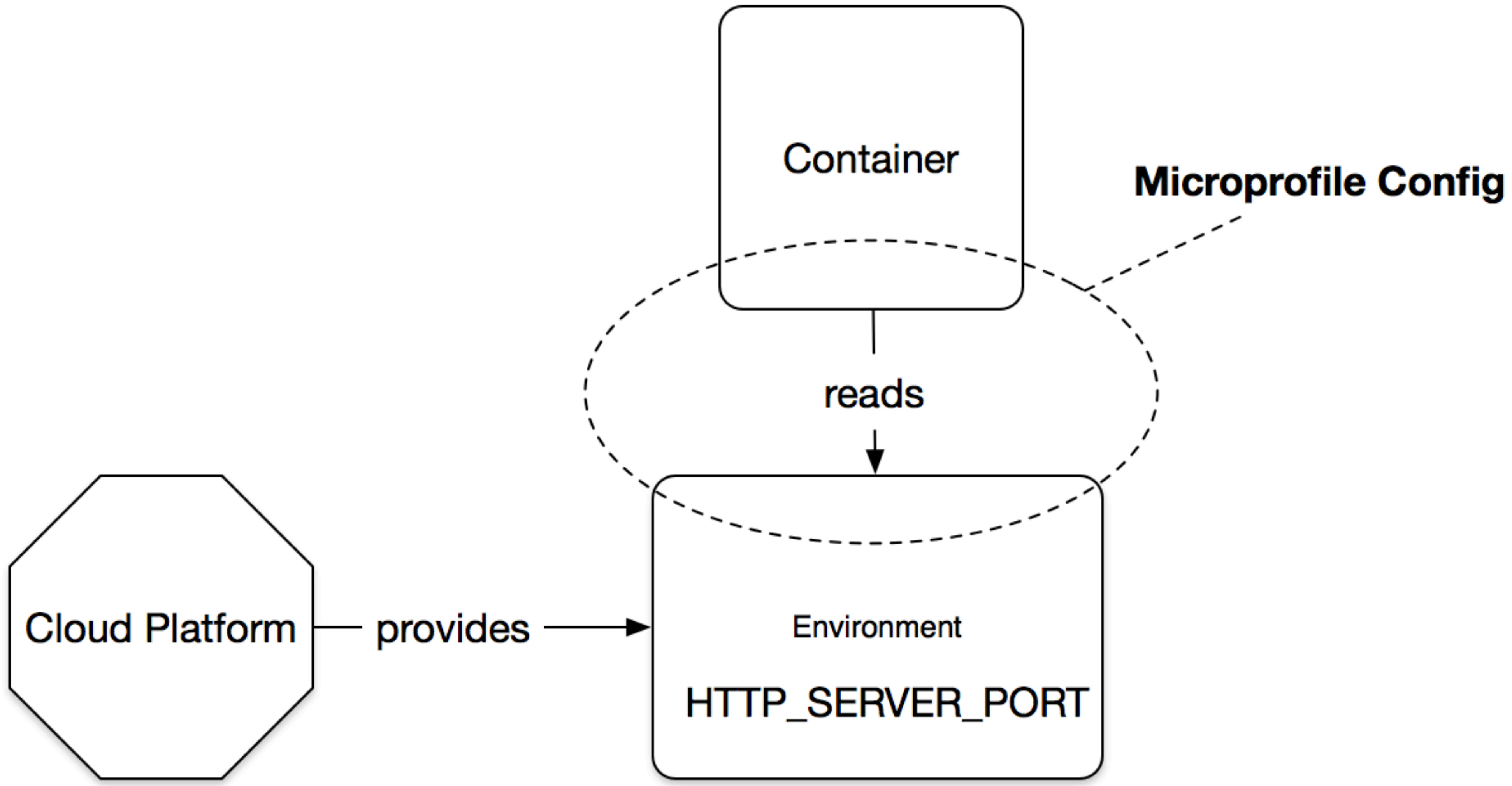
```
@ApplicationScoped
public class InjectedConfigUsageSample {
  @Inject
  private Config config;

  @Inject
  @ConfigProperty(name="myprj.some.url")
  private String someUrl;

  @Inject
  @ConfigProperty(name="myprj.some.port")
  private Optional<Integer> somePort;

  @Inject
  @ConfigProperty(
   name="myprj.some.dynamic.timeout",
   defaultValue="100")
  private javax.inject.Provider<Long> timeout;
}
```

Cloud Native Fit for MicroProfile Configuration

Use Case: Container configuration

Platform provided configuration, consumed by container:

- Configuration values managed by the cloud platform (i.e. ConfigMaps)
- Injected into the container at startup
- Consumed through ENV VAR’s



https://12factor.net/config

Health Checks

Health checks

Is the computing node alive and ready?

Rationale

- Used to determine if a computing node needs to be discarded (terminated, shutdown) and eventually replaced by another instance (aka liveness)
  - Catch a deadlocks. Restarting makes the application more available despite bugs
- Needed to determine if computing node is ready to perform work (aka readiness)
  - Determine if traffic will routed there, i.e. upon startup/rollover

https://**github.com/eclipse/microprofile-health**/releases

Health check API 1/2

Exposing health check procedures

Exposing health checks to a cloud platform

- Implement *HealthCheck* interface
- To return a *HealthCheckResponse*
- Belongs to the application domain
- Invoked from the outside
- One or multiple checks, but single outcome

```java
public class CheckDiskspace implements HealthCheck {
  @Override
  public HealthCheckResponse call() {
    return HealthCheckResponse.named("diskspace")
    .withData("free", "780mb")
    .up()
    .build();
  }
}
```

Health Check API 2/2

CDI based: @ApplicationScoped @Health

Exposing beans as health checks:

- @Health annotation for discovery
- Implementation of *HealthCheck*
- In future also method level bindings

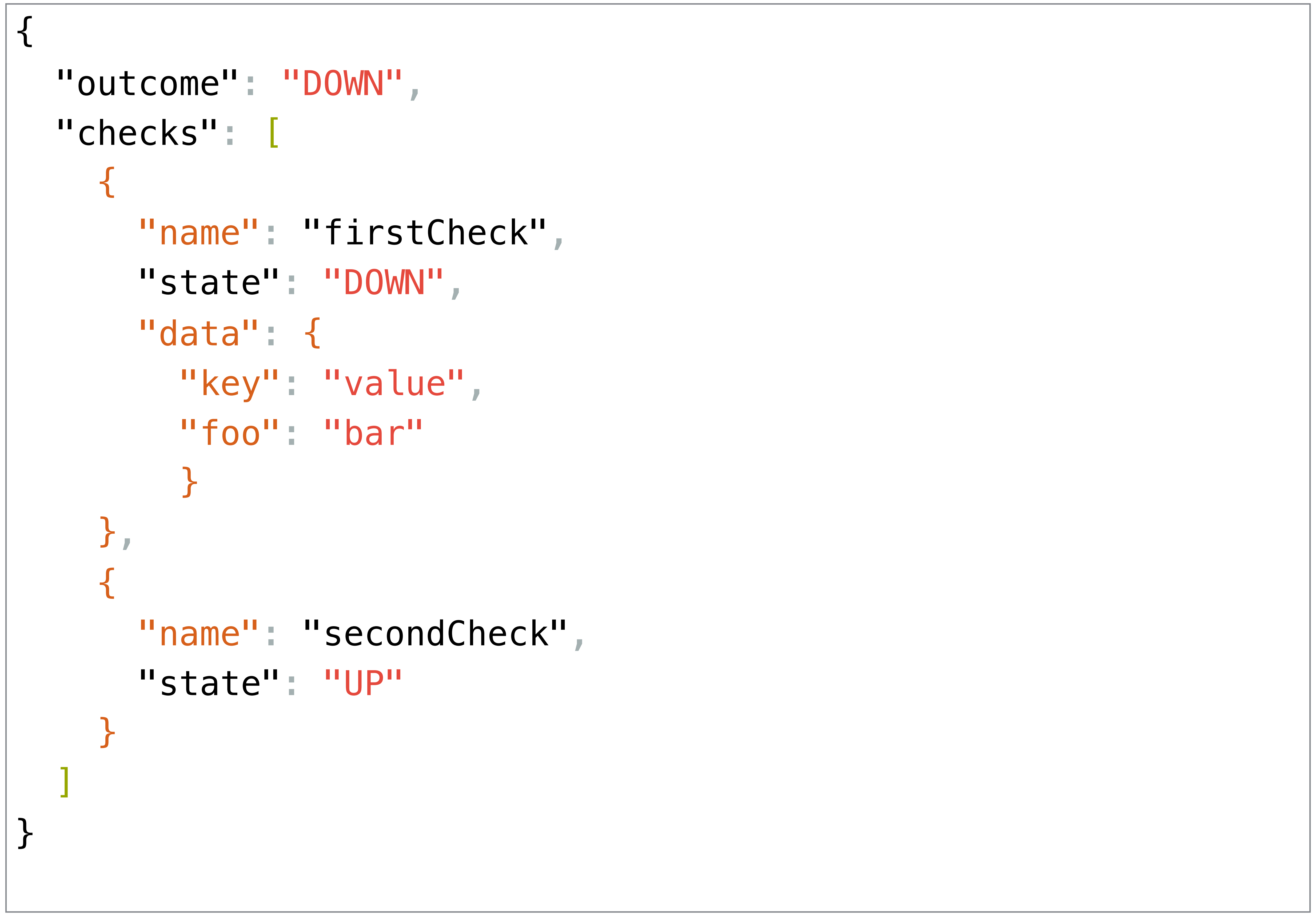
```
@Health
@ApplicationScoped
public class MyCheck implements HealthCheck {
  public HealthCheckResponse call() {
    [...]
  }
}
```

Health check protocol

On the wire

HTTP Binding on canonical endpoint (**/health**)

- Calls into all health check procedures
- Leads to a composite response
- But with a **single outcome** (UP/DOWN)
- HTTP Status code reflects the outcome (200/500)



```
{
  "outcome": "DOWN",
  "checks": [
    {
      "name": "firstCheck",
      "state": "DOWN",
      "data": {
        "key": "value",
        "foo": "bar"
        }
    },
    {
      "name": "secondCheck",
      "state": "UP"
    }
  ]
}
```

Cloud native fit for MicroProfile Health Checks

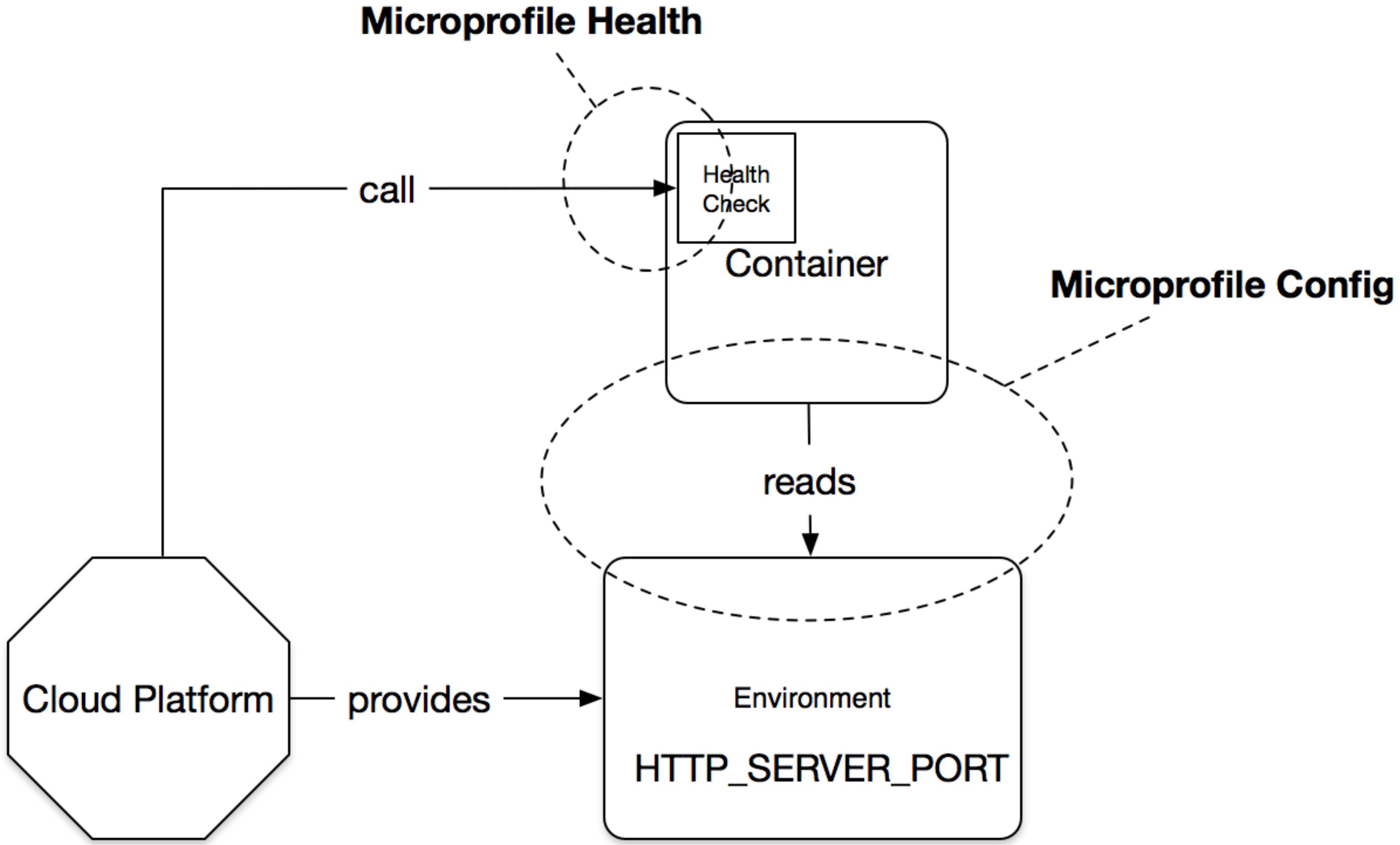
Essential contract with the orchestration framework and scheduler

Driven by the cloud platform

- Continuously checking your containers
- Platform determines readiness and liveness
- Most platform have HTTP health protocol support
    - Usually just interested in the HTTP Status Code

Provided by the application developer

- Concrete health check procedures
    - i.e. when is my service ready?
    - i.e. how to determine liveness?



https://**kubernetes.io**/docs/tasks/configure-pod-container/configure-**liveness-readiness-probes**/

Metrics

Metrics

Service monitoring

Rationale

- Used for long term and continuous monitoring of service performance
- Provide long term trend data for capacity planning and pro-active discovery of issues
- Aims at delivering:
    - A coherent model to that define metrics
    - A Java API to provide measurements from the application level
    - A reasonable set of out-of-the-box metrics for runtimes
    - A REST interface to retrieve the data from monitoring agents

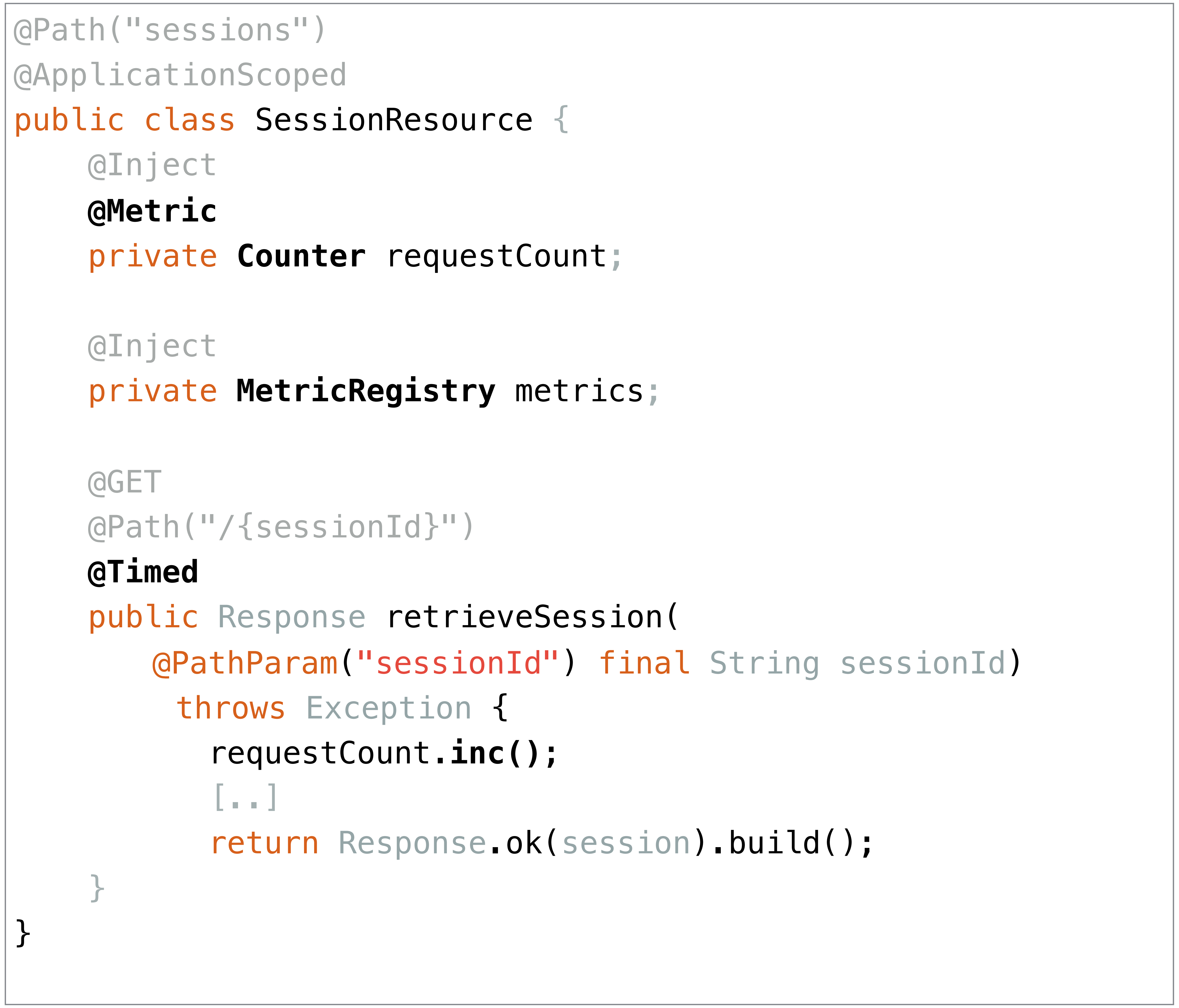
📖 https://github.com/eclipse/microprofile-metrics/releases

Metrics API

Exposing application level metrics

Internally the MetricsRegistry acts as the store

- Metrics can be added to or retrieved from the registry either using the *@Metric* annotation or using the *MetricRegistry* object directly
- REST API interfaces with the *MetricsRegistry*



```
@Path("sessions")
@ApplicationScoped
public class SessionResource {
    @Inject
    @Metric
    private Counter requestCount;

    @Inject
    private MetricRegistry metrics;

    @GET
    @Path("/{sessionId}")
    @Timed
    public Response retrieveSession(
        @PathParam("sessionId") final String sessionId)
         throws Exception {
           requestCount.inc();
           [..]
           return Response.ok(session).build();
    }
}
```

Metric Model

Common types to express measurements

Metric	Description
Gauge	Simple Value
Counter	Incrementing or decrementing value
Meter	Measures the rate at which an event occurs
Histogram	Measures the distribution of values
Timer	Combination of meter and histogram

Retrieving metrics

Using the REST interface

Supports both JSON or Prometheus formats

- MAY include one or more metrics
- Http OPTIONS for meta data
- Http GET for values
- **/metrics** is the canonical entry point
- Different scopes: application, base, vendor

```
curl -X OPTIONS http://.../metrics/application
     -H "Accept: application/json"

{
    “io.microprofile.[…].SessionResource.requestCount”: {
        "displayName": “SessionResource.requestCount",
        "type": “counter",
        "unit": "none"
    }
}
```

```
curl http://.../metrics/application/SessionResource.requestCount
    -H "Accept: application/json”

{
  "SessionResource.requestCount" : 6
}
```

Cloud native fit for MicroProfile Metrics

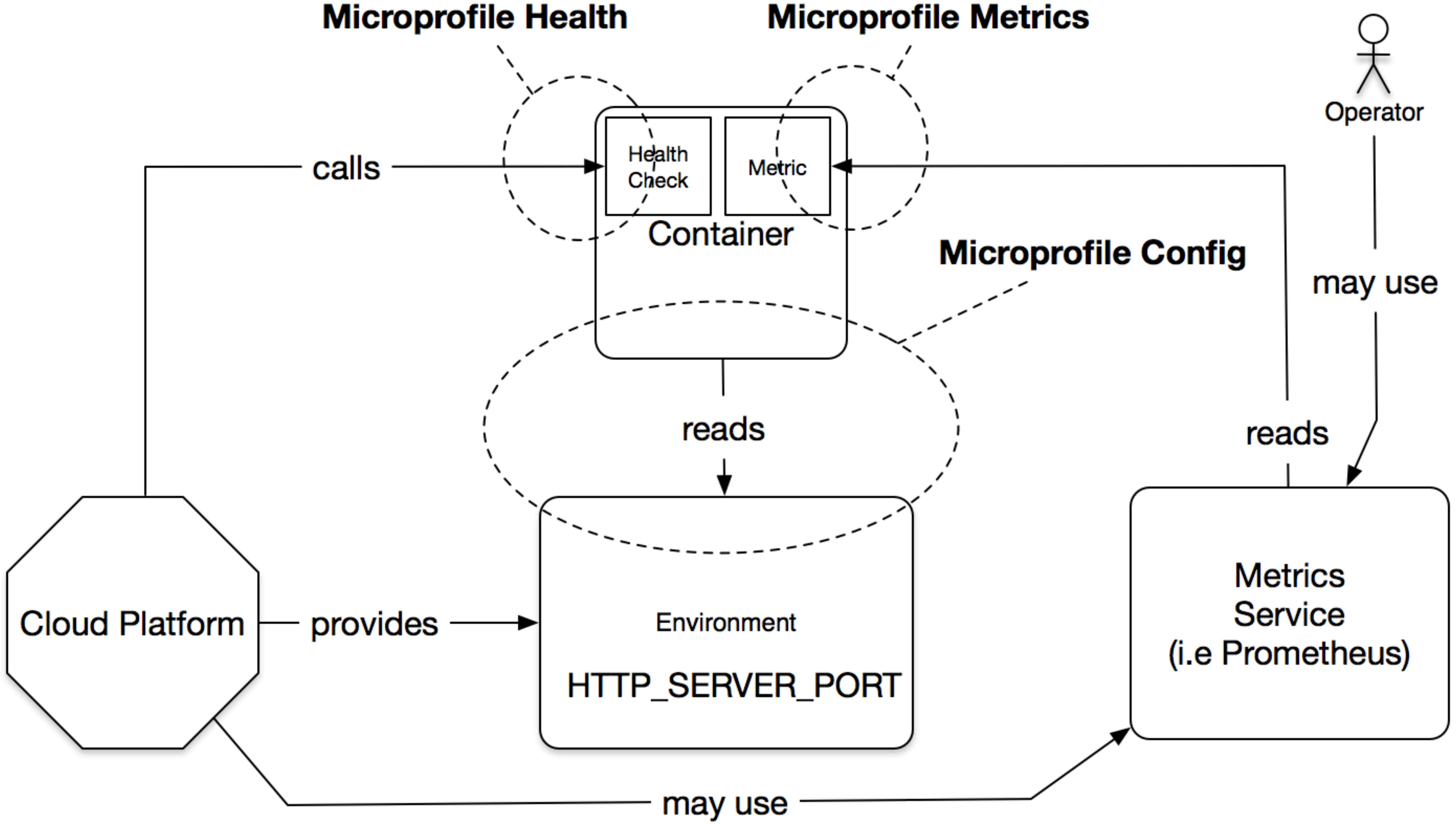
Integration with monitoring and analysis capabilities

To support the cloud platform

- Might guide scheduling decisions of the cloud platform (i.e replicas, increasing capacity)

To support the operational roles

- Provides monitoring capabilities
- Trends analysis
- Continuous experimentation
  - Data to support A/B, Blue/Green deployments



https://prometheus.io/, https://speakerdeck.com/yanaga/b-and-canary

Fault Tolerance

RED HAT
DEVELOPERS

Fault Tolerance

The fun part in distributed systems

Rationale

- Fallacies of distributed computing
  - Networking just isn’t reliable (latency, partitioning, etc)
- Introduce failure detection and handling capabilities to deal with this
- Separate execution logic from execution
- Deliver a *composable* set of fault handling primitives

https://github.com/eclipse/microprofile-fault-tolerance/releases

Failure detection and handling

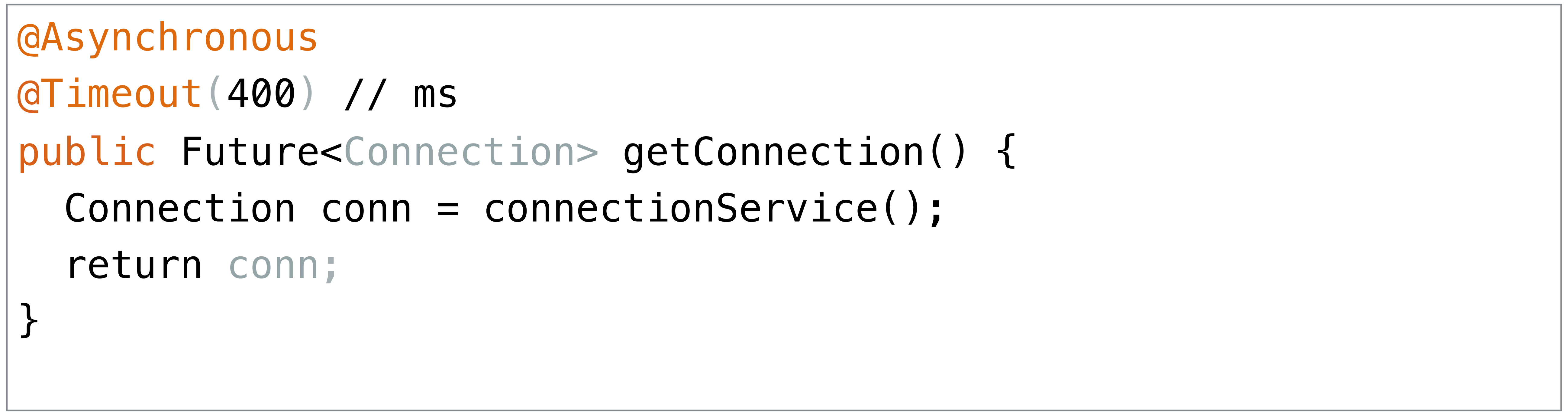
Strategies to deal with problems

Type	Description
Timeout	Timeouts on executions (No difference between no response or late response)
Retry	Retry rather then fail
Fallback	Provide an alternative solution for a failed execution
CircuitBreaker	Keep reoccurring failures to cascading through the system
Bulkhead	Isolate system resources to prevent starvation

https://www.slideshare.net/**ufried/patterns-of-resilience**

Fault Tolerance API 1/2

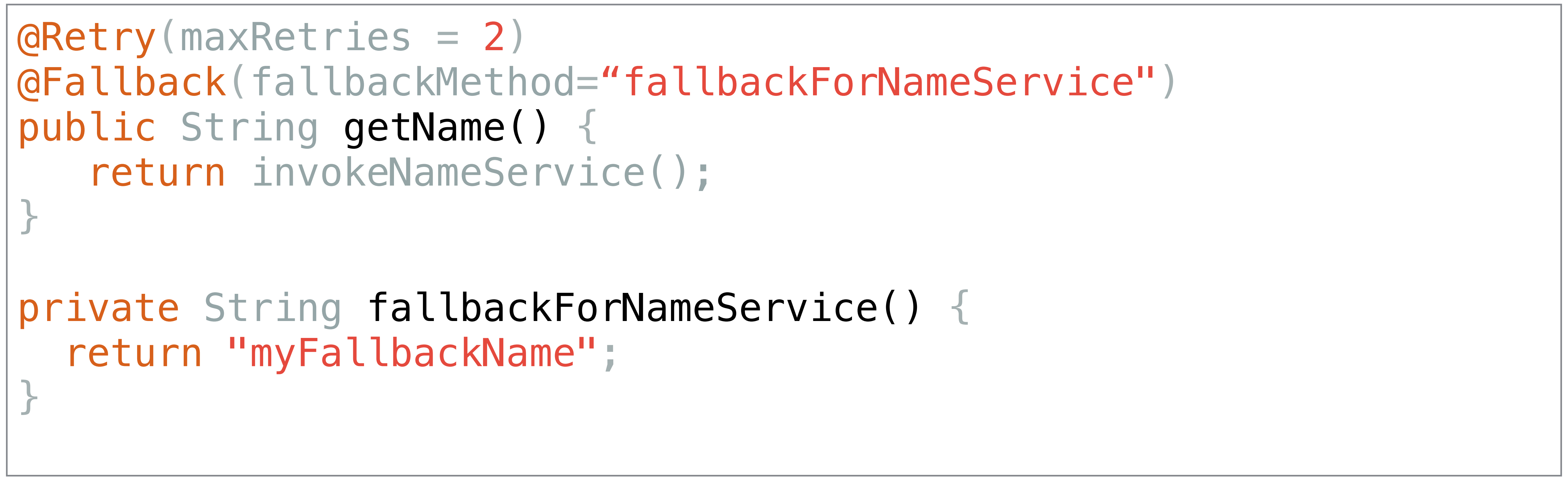
Basic failure handling approaches



```java
@Asynchronous
@Timeout(400) // ms
public Future<Connection> getConnection() {
  Connection conn = connectionService();
  return conn;
}
```

Composable set of strategies

- Usually @Asynchronous
  - Clear execution control when owning the thread
- Compose a strategy from primitive types



```java
@Retry(maxRetries = 2)
@Fallback(fallbackMethod=“fallbackForNameService")
public String getName() {
   return invokeNameService();
}

private String fallbackForNameService() {
  return "myFallbackName";
}
```

Fault Tolerance API 2/2

Circuit breakers and bulkheads

More sophisticated strategies

- CircuitBreaker:
  - Cycles through closed/open/half-open
  - Failure threshold drives state machine
  - Fail fast approach
- BulkHead:
  - Good for shared contexts
  - Semaphore or thread-pool isolation strategy
  - Prevent resource starvation to create cascading errors

```
@CircuitBreaker(
  successThreshold = 10,
  requestVolumeThreshold = 4, failureRatio=0.75,
  delay = 1000)
public SomeResult remoteCall() {
  return performRemoteInvocation();
}
```

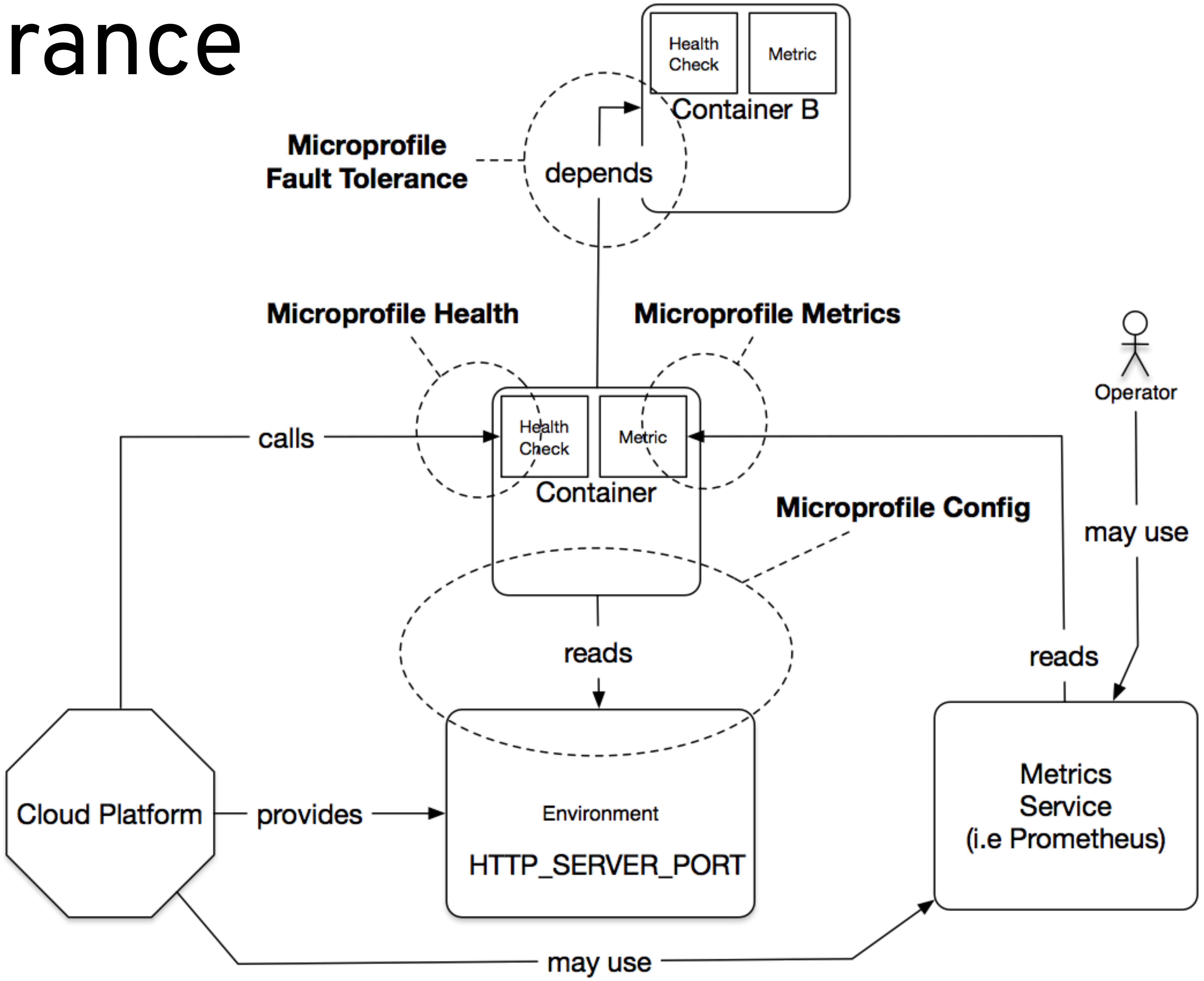
```
@Asynchronous
@Bulkhead(value = 5, waitingTaskQueue = 8)
public Future<QueryResult> accessSharedRemoteSystem()
{
  return queryResult();
}
```

Cloud native fit for MicroProfile Fault Tolerance

Resilience and fault tolerance in service oriented architectures

Pretty much an application level concern

- Any dependency on other, remote services
- A way to embrace failures
  - Remote invocations
  - 3rd party systems
- Not just networking failures, but also planned unavailability
  - i.e. Rolling Deployments



JWT

JWT

Token based authentication and authorisation

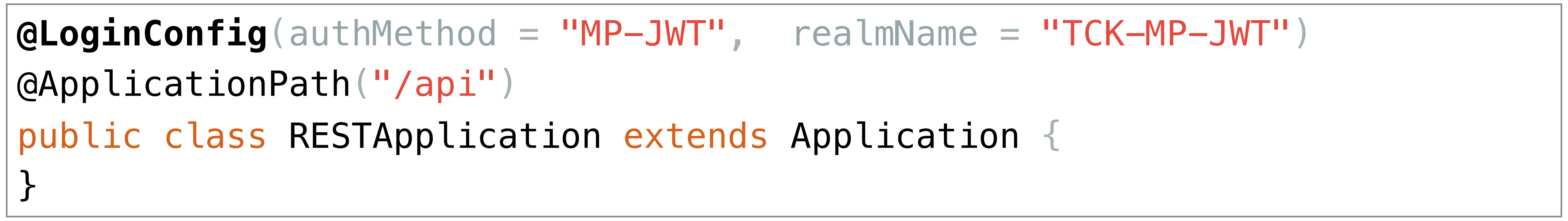
Rationale

- In stateless architectures, support a way to **authenticate**, **authorize** and **verify identities** based on a security token
- Across client-service and service-service interactions
- i.e supporting using OpenID Connect(OIDC) based JSON Web Tokens(JWT)
  for role based access control(RBAC) of microservice endpoints
- Provide a **common set of claims** (unique identifier, user name, groups)
  - **java.security.Principal** interface extension that makes this set of required claims available
- Integration with container API's
  - Login config for JAX-RS
  - Injection of JasonWebToken in CDI

📖 https://github.com/eclipse/microprofile-jwt-auth/releases

JWT API $^{1/2}$

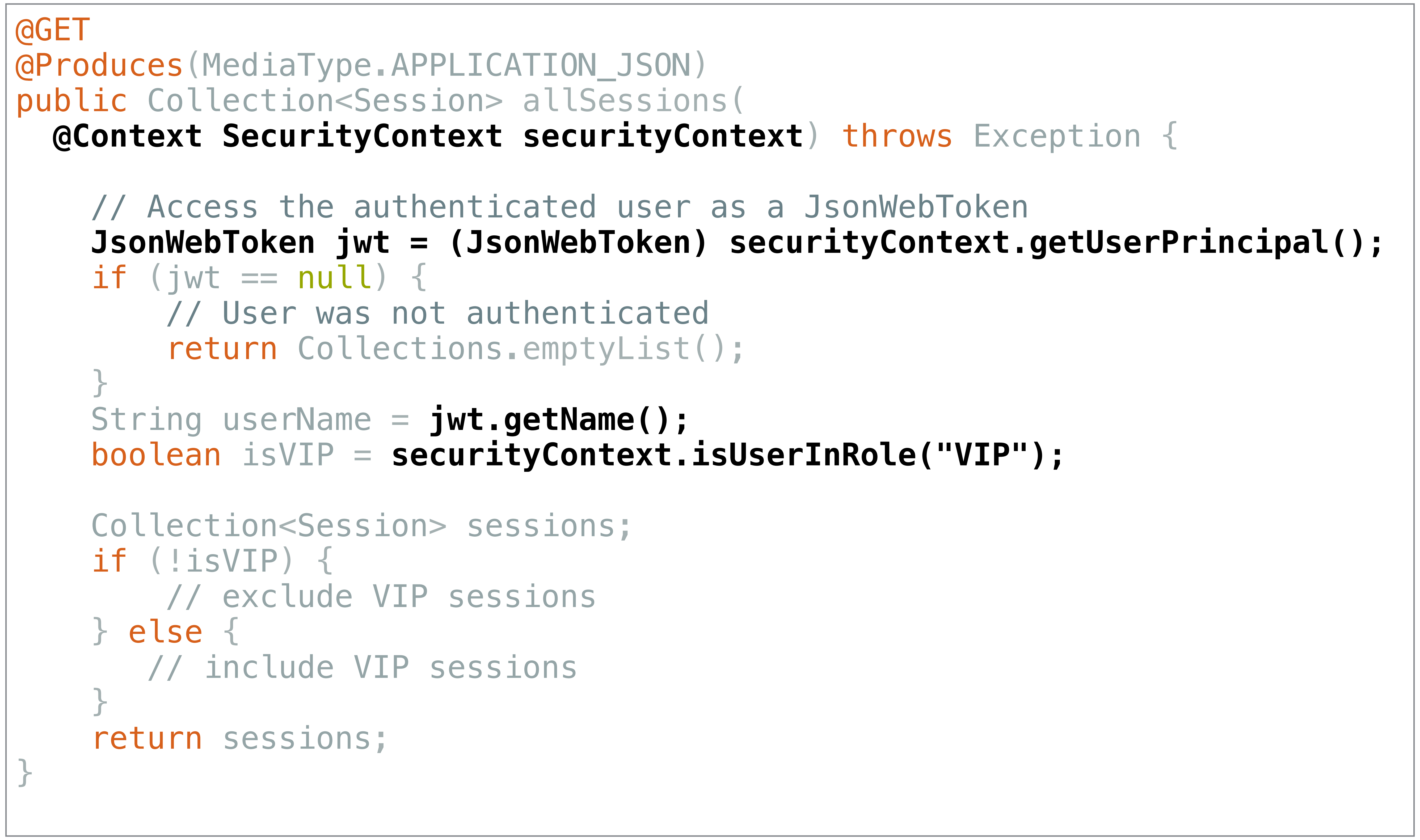
Securing REST endpoints



```
@LoginConfig(authMethod = "MP-JWT",  realmName = "TCK-MP-JWT")
@ApplicationPath("/api")
public class RESTApplication extends Application {
}
```

Within the bounds of MP specs

- *@LoginConfig* rather then web.xml
- Regular *SecurityContext* access
- Particular Principal implementation (*JsonWebToken*)
- Either explicit or declarative security enforcement (i.e. *@RolesAllowed*)



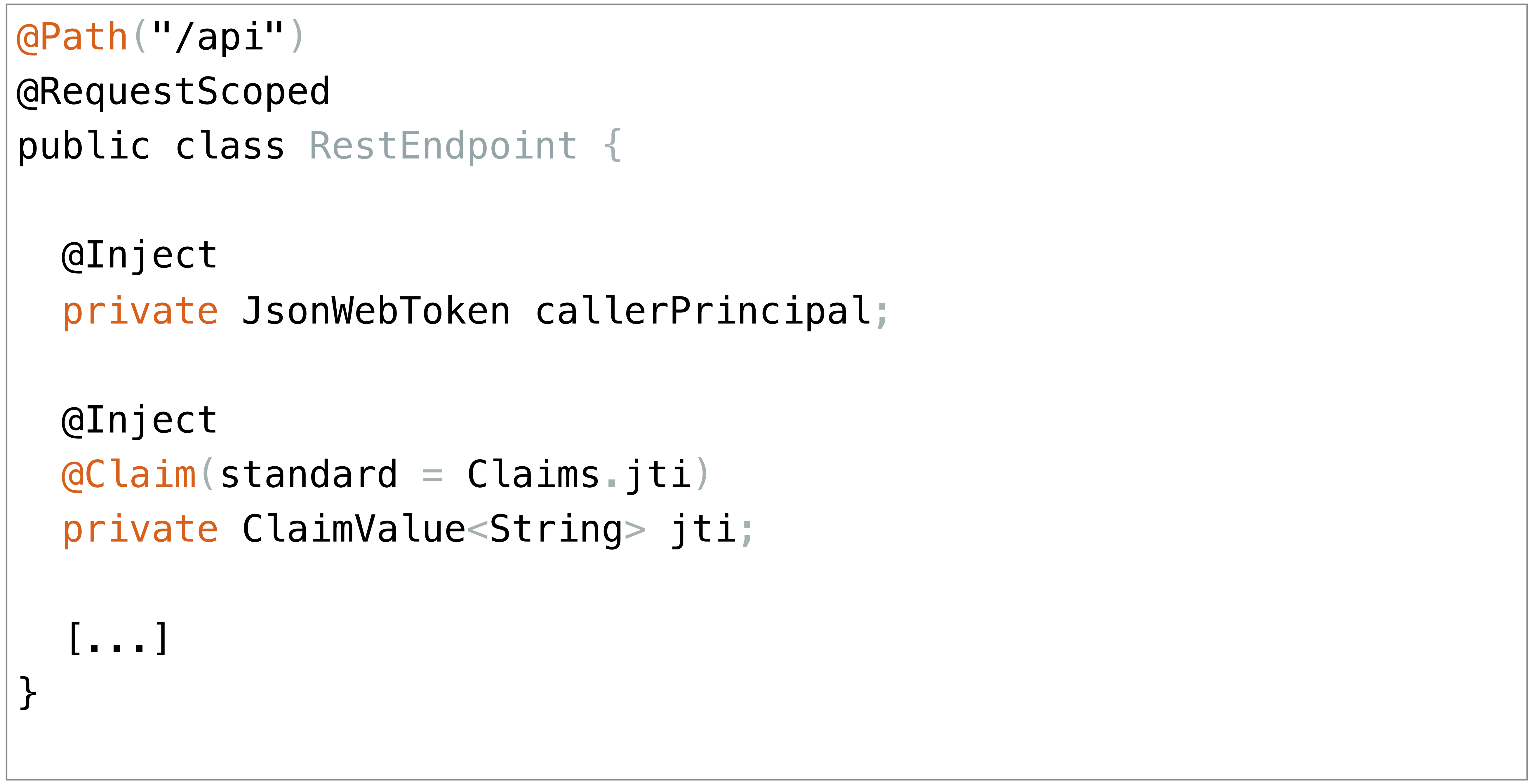
```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Collection<Session> allSessions(
  @Context SecurityContext securityContext) throws Exception {

    // Access the authenticated user as a JsonWebToken
    JsonWebToken jwt = (JsonWebToken) securityContext.getUserPrincipal();
    if (jwt == null) {
        // User was not authenticated
        return Collections.emptyList();
    }
    String userName = jwt.getName();
    boolean isVIP = securityContext.isUserInRole("VIP");

    Collection<Session> sessions;
    if (!isVIP) {
        // exclude VIP sessions
    } else {
       // include VIP sessions
    }
    return sessions;
}
```

JWT API 2/2

CDI based: @Inject @JsonWebToken

Request scope beans get access to:

- @Inject *JsonWebToken*
- @Claim’s
  - identifier
  - principal
  - groups
  - etc



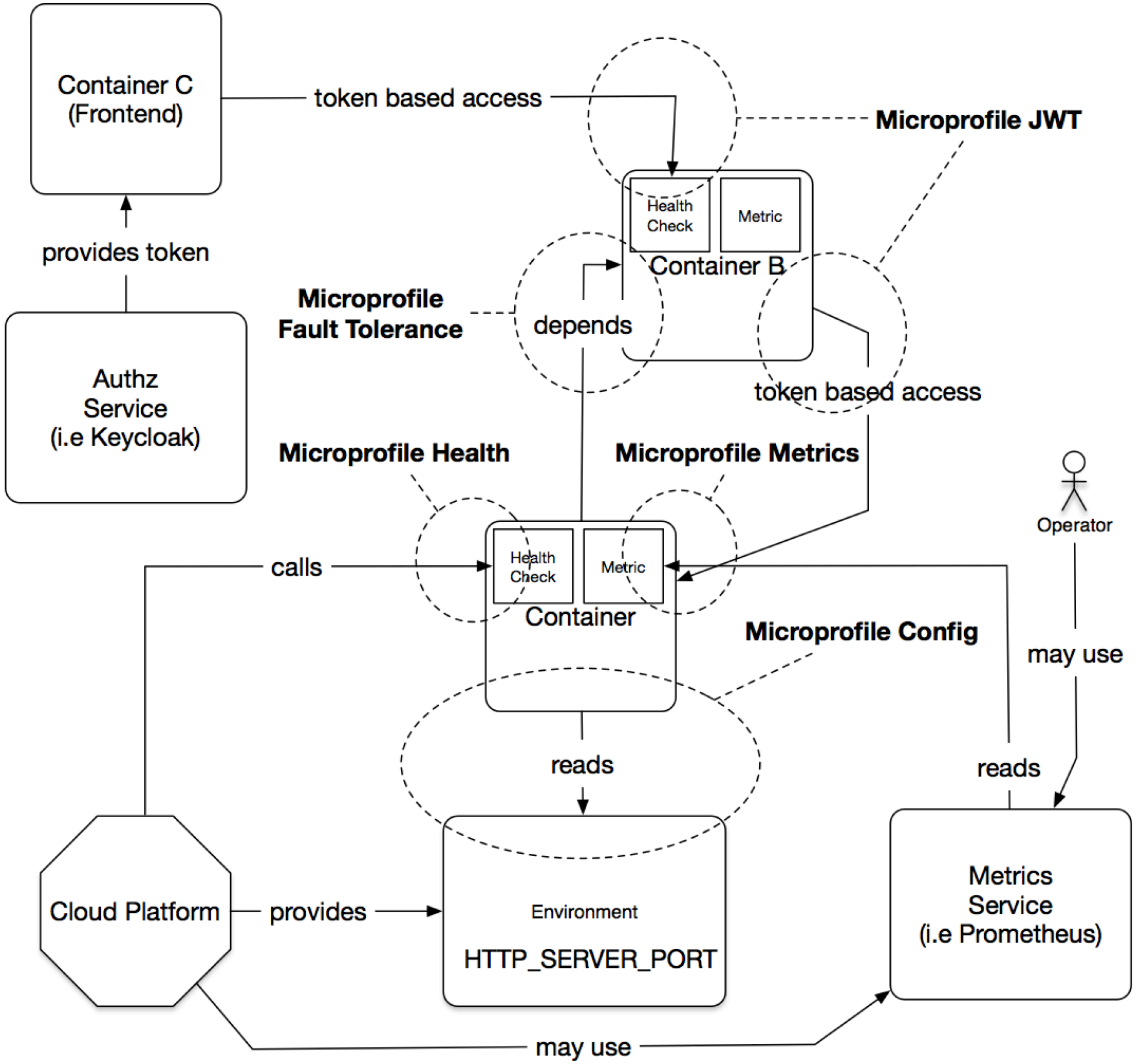
```
@Path("/api")
@RequestScoped
public class RestEndpoint {

  @Inject
  private JsonWebToken callerPrincipal;

  @Inject
  @Claim(standard = Claims.jti)
  private ClaimValue<String> jti;

  [...]
}
```

Cloud native fit for MicroProfile JWT

Security context propagation

Securing stateless services

- Token based approach
- Self-contained
- Depends on Authz/IDM service (i.e. Keycloak)
- Often used with OpenID authentication



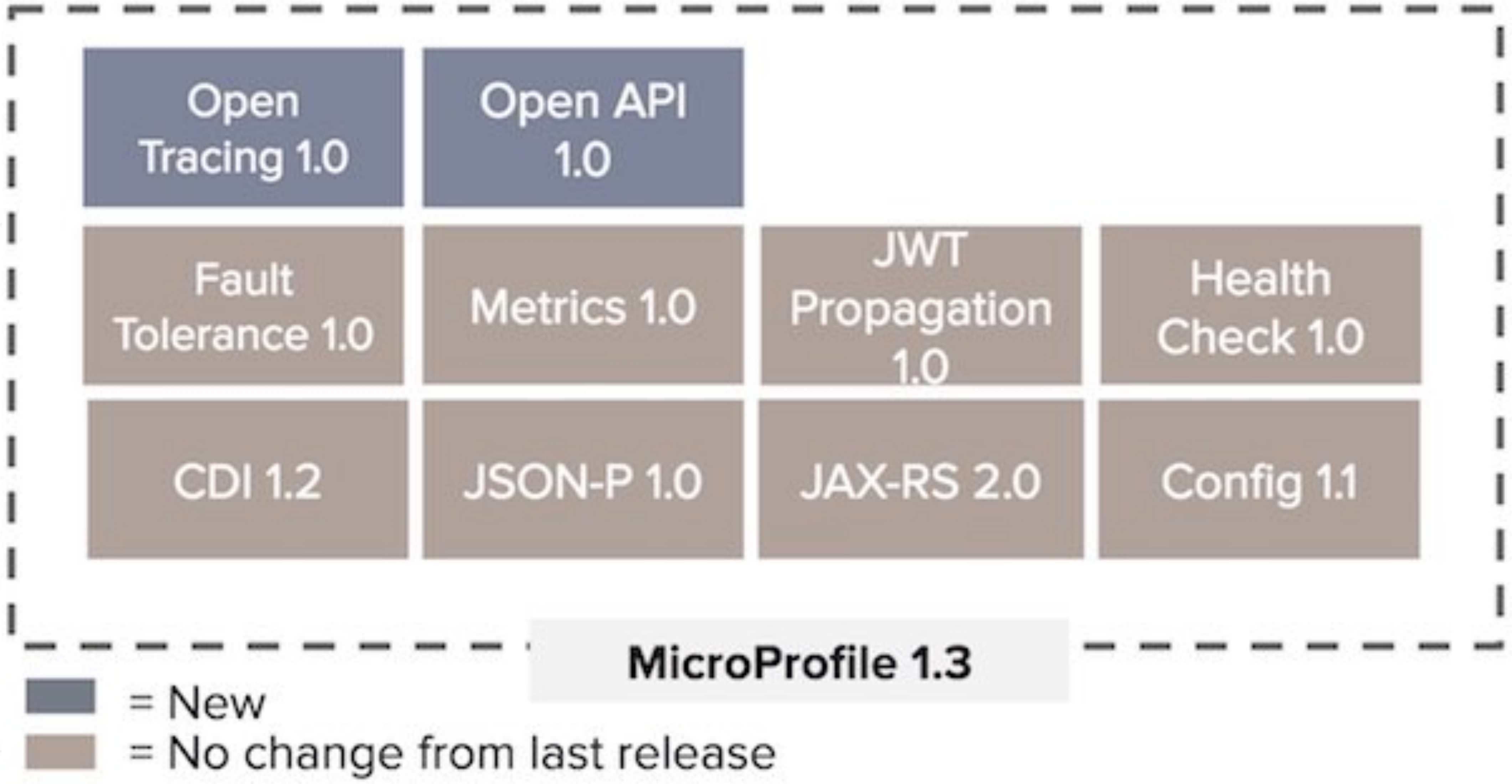
http://www.keycloak.org/, http://openid.net/connect/

What’s next?

RED HAT DEVELOPERS

MicroProfile.Next

The road ahead



Open Tracing 1.0 | Open API 1.0

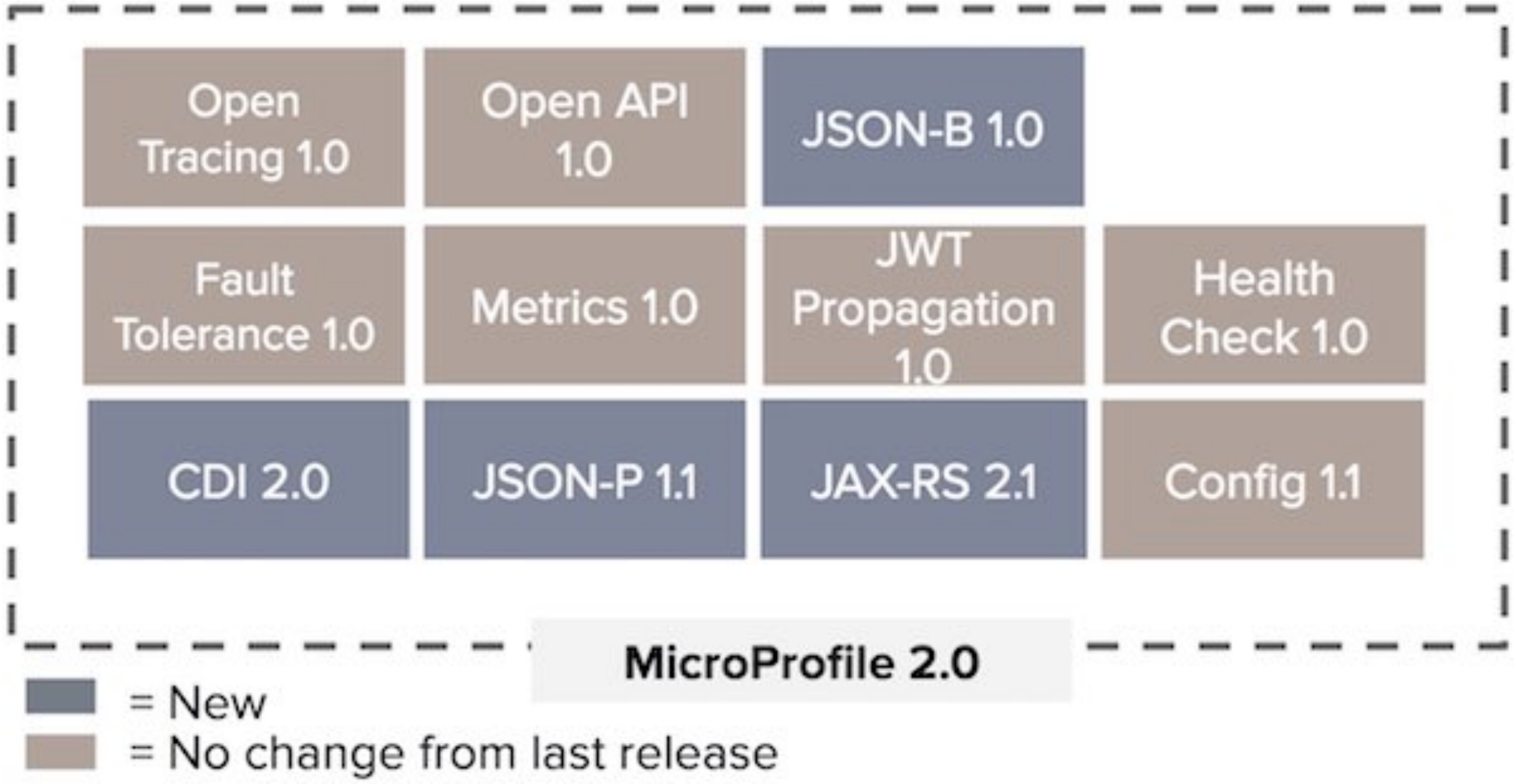
Fault Tolerance 1.0 | Metrics 1.0 | JWT Propagation 1.0 | Health Check 1.0

CDI 1.2 | JSON-P 1.0 | JAX-RS 2.0 | Config 1.1

MicroProfile 1.3

🟦 = New

🟫 = No change from last release



Open Tracing 1.0 | Open API 1.0 | JSON-B 1.0

Fault Tolerance 1.0 | Metrics 1.0 | JWT Propagation 1.0 | Health Check 1.0

CDI 2.0 | JSON-P 1.1 | JAX-RS 2.1 | Config 1.1

MicroProfile 2.0

🟦 = New

🟫 = No change from last release

More Information

Contribute & Collaborate

Main resources under http://microprofile.io

- All projects and specifications hosted under Eclipse
- Open, friendly community
- Chime in, participate, drive

https://groups.google.com/forum/#!forum/microprofile

Thanks!