Current Status

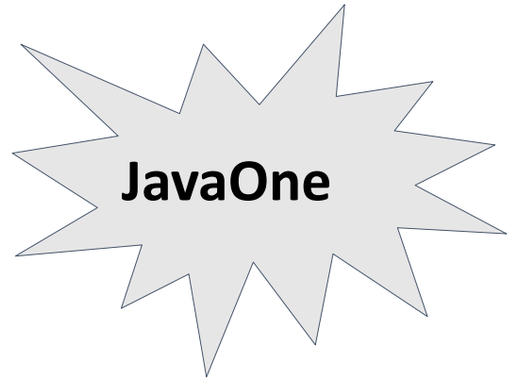
Eclipse Incubator

- https://projects.eclipse.org/projects/technology.microprofile - ASLv2 License
- http://microprofile.io/ - Home Page
- https://github.com/eclipse - Eclipse Foundation GitHub Organization
    - microprofile
        - Documentation
    - microprofile-evolution-process
        - Specification proposals
- microprofile-config
    - Configuration API
- microprofile-health
    - Health check API and support
- microprofile-fault-tolerance
    - Standardizing fault tolerance policy API and integration
- microprofile-jwt-auth
    - Standardizing JSON web token claims to support RBAC

MicroProfile Roadmap

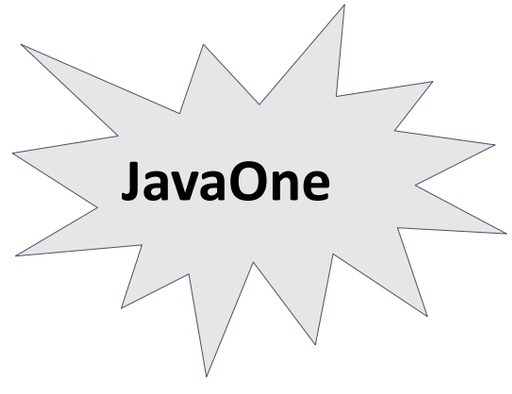
- MicroProfile 1.1 (2Q 2017)
  - Prioritized list of features
    - Config 1.0
      - Planning an official Eclipse release of Config 1.0 by Devoxx UK (early May)
    - Fault Tolerance 1.0
      - Emily working closely with community to nail down programming models
        - CDI-based vs standalone API
      - Integration with (Hystrix) dashboard might get pushed out to FT 1.1 (MP 1.2)
      - Planning an official Eclipse release of FT 1.0 by end of May/ early June
- MicroProfile project needs an official MicroProfile 1.1 content and release

MicroProfile Roadmap (cont)



- MicroProfile 1.2 (3Q 2017)
  - Prioritized list of features
    - Fault Tolerance 1.1 w/ (Hystrix) Dashboard
    - Security (JWT Propagation)
    - Health Check / Metrics / Monitor (single feature or multiple?)
    - OpenTracing (Relatively new community request, stretch goal)
- We would really like to push this for an August delivery to allow a bigger splash at JavaOne, like we did last year. Vendors would have some time to develop real code for demonstrations and presentations at JavaOne. Another stretch goal.

MicroProfile Roadmap (cont)



- MicroProfile 2.0 (3Q 2017)
  - Update to latest Java EE 8 specs
    - CDI 2.0
    - JAX-RS 2.1
    - JSON-P 1.1
    - JSON-B 1.0 (???)

- Based on MP 1.0, 1.1, or 1.2 content?
  - If MP 1.2 is “announced”, then MP 2.0 would contain the updated Java EE 8 specs plus the MP 1.2 content

MP Configuration

- GOAL: Standardize Configuration API so microservices can run in different environments seamlessly

- APIs/SPIs for:
  - Configuration providers to introduce configuration sources with a priority for ordering, and type converters
  - Accessing configuration information and values
  - Injecting configuration values

Simple Programmatic Example

```
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);
        Integer serverPort = config.getValue("acme.myprj.some.port", Integer.class);


        callToServer(serverUrl, serverPort);
    }
}
```

Simple DI Style Example

```
@ApplicationScoped
public class InjectedConfigUsageSample {
    @Inject
    private Config config; // Access full Config object

    @Inject                                                          1
@ConfigProperty(name=“myprj.some.url") // Required to exist or DeploymentException
private String someUrl;
    @Inject                                                          2
@ConfigProperty(name=“myprj.some.port") // Not required
private Optional<Integer> somePort;
    @Inject                                                          3
@ConfigProperty(name="myprj.some.dynamic.timeout", defaultValue="100")
private javax.inject.Provider<Long> timeout; // Dynamically resolved on each Provider#get()
```

MP Fault Tolerance

- Effort to standardize APIs for dealing with latency and failures
  - Take popular concepts from libraries such as Hystrix and Failsafe
- Initial focus
  - TimeOut: Define a duration for timeout
  - RetryPolicy: Define a criteria on when to retry
  - Fallback: provide an alternative solution for a failed execution.
  - Bulkhead: isolate failures in part of the system while the rest of the system can still function.
  - CircuitBreaker: offer a way of fail fast by automatically failing execution to prevent the system overloading and indefinite wait or timeout by the clients.

Requirements

- Loose coupling: Execution logic should not know anything about the execution status or fault tolerance.
- Failure handling strategy should be configured when the execution takes place.
- Support for synchronous and asynchronous execution
- Integration with 3rd party asynchronous APIs. This is necessary to handle executions that are completed at some time in the future, where retries will need to be explicitly scheduled from within the asynchronous execution. This is common when working with various 3rd party asynchronous tools such as Netty, RxJava, Vert.x, etc.
- Require immutable failure handling policy configuration
- Some Failure policy configurations, e.g. CircuitBreaker, RetryPolicy, can be used stand alone. For example, it has been very useful for circuit breakers to be standalone constructs which can be plugged into and intentionally shared across multiple executions. Likewise for retry policies. Additionally, an Execution construct can be offered that allows retry policies to be applied to some logic in a standalone, manually controlled way.

Possible CDI

```
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class FaultToleranceBean {
    int i = 0;
    @Retry(maxRetries = 2)
    public Runnable doWork() {
        // This unreliable service sometimes succeeds but
        // sometimes throws a RuntimeException
        Runnable mainService = () -> serviceA();
        return mainService;
    }
}
```

Possible CDI

```
// When `TimeOutException` was received, delay 2 seconds
and then retry 2 more times.

RetryPolicy rp = FaultToleranceFactory
        .getInstance(RetryPolicy.class)
        .retryOn(TimeOutException.class)
        .withDelay(2, TimeUnit.SECONDS)
        .withMaxRetries(2);
SyncExecutor se = Executor.with(rp);
se.get(() -> someCallable());
```

MP Health Check

- Goals
  - MUST be compatibility with Kubernetes health checks: http://kubernetes.io/docs/user-guide/liveness/
  - MUST be appropriate for machine-to-machine communication
  - SHOULD give enough information for a human administrator

MP Health Check

- Goals
  - MUST be compatibility with Kubernetes health checks: http://kubernetes.io/docs/user-guide/liveness/
  - MUST be appropriate for machine-to-machine communication
  - SHOULD give enough information for a human administrator

Concepts

- Producers (services, endpoints) expose at least a REST/HTTP /health endpoint
  - Other protocols may be supported
- Producers have zero or more health check procedures configured
  - Represent logical services whose health the producer depends on
    - memory, disk space, bandwidth, database access, …
    - Must have an id, “UP” or “DOWN” status as well as procedure specific status properties
- Invocation of the /health endpoint results in the producer evaluating the configured health check procedures and returning a status and possibly a JSON payload representing the health check procedure information
  - Overall health is logical combination of all procedures

Request Outcomes

Request	HTTP Status	JSON Payload	State	Comment
/health	200	Yes	UP	Check with payload
/health	204	No	UP	Check without procedures installed
/health	503	Yes	Down	Check failed
/health	500	No	No	Request processing failed (i.e. error in procedure

Request Examples

Status 200:

```
{
  "outcome": "UP",
  "checks": [
    { "id": "myCheck",
      "result": "UP",
      "data": {
        "key": "value",
        "foo": "bar"
      }
    }, { “id”: “memory”,
      “result”: “UP”,
      “data”: {
        “available”: 32768
        “free”: 4096,
        “units”: “Mb”
      }
    }
  ]
}
```

Status 503:

```
{
  "outcome": "DOWN",
  "checks": [
    { "id": "myCheck",
      "result": "DOWN",
      "data": {
        "key": "value",
        "foo": "bar"
      }
    }, { “id”: “memory”,
      “result”: “UP”,
      “data”: {
        “available”: 32768
        “free”: 4096,
        “units”: “Mb”
      }
    }
  ]
}
```

Possible JAX-RS


```
@Path("/app")
public class HealthCheckResource {

    @GET
    @Path("/diskspace")
    @Health
    public HealthStatus checkDiskspace() {
        [...]
    }

    @GET
    @Path("/something-else")
    @Health
    public HealthStatus checkSomethingElse() {
        [...]
    }
}
```

Possible CDI

```
@ApplicationScoped()
public class HealthChecks {

    @Produces
    @Health
    public HealthStatus checkDiskspace() {
       [...]
    }

    @Produces
    @Health
    public HealthStatus checkSomethingElse() {
       [...]
    }
}
```

MP JWT RBAC

- Effort to standardize identity and access grants in a JSON Web Token (JWT, RFC 7519).
- Builds on the JWT minimal set of claims to define claims for:
    - caller identity
    - caller roles
        - Independent of services
        - Specific to a service
- Requires use of JSON Web Signature (JWS, RFC 7515)
    - With header alg="RS256"; RSA public key signature with SHA-256 hash algorithm

Example Token

```
Header:
{
  "alg": "RS256"
  "typ": "JWT"
}
```

```
Payload:
{
  "jti": "e64a4894-b181-4646-ab14-6d415a473749",
   "iss": "https://server.example.com",
   "sub": "24400320",
   "aud": "s6BhdRkqt3",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "iat": 1311280970,
   "auth_time": 1311280969,
   "preferred_username": “alice”,
   "realm_access": {
         "roles": ["role-in-realm", "user", “shared-service-role“]
    },
    "resource_access": {
         "my-service": {
           "roles": ["role-in-my-service"]
         },
         "my-service2": {
           "roles": [“role-in-my-service2"]
         }
    },
}
```

1

2

3

Proposed Principal

```
package org.eclipse.microprofile.jwt;

import java.security.Principal;
import java.util.Set;

public interface JWTPrincipal extends Principal {
    // Various standard claim accessors...
    String getIssuer();
    String getSubject();
    long getExpiration();
    long getIssuedAt();
    long getNotBefore();
    //...
    // Extension claims
    String getPrincipalName();
    Set<Principal> getRealmRoles();
    Set<String> getServiceNames();
    Set<Principal> getRolesForService(String serviceName);
}
```

JWTPrincipal

- Returned by various getCallerPrincipal, getUserPrincipal container and security context methods
- Available for injection

MicroProfile

Community

redhat.

IBM

FUJITSU

kumuluzEE

Tomitribe

payara

LJC

hazelcast

SOUJava
sociedade de usuários java