



Eclipse MicroProfile™ white paper

2019

Pro bono by Lars Röwekamp





2019

2019



“ AN OPEN FORUM TO OPTIMIZE ENTERPRISE JAVA FOR A
 MICROSERVICES ARCHITECTURE BY INNOVATING ACROSS MULTIPLE
 IMPLEMENTATIONS & COLLABORATING ON COMMON AREAS OF
 INTEREST WITH A GOAL OF STANDARDIZATION.”

A Technical Focus

This White Paper discusses popular programming patterns that streamline microservices development. MicroProfile addresses those patterns by leveraging and extending a developer's existing skillset. For the sake of brevity, a developer will be referred to in its plural form: we.

Microservices is the most popular architecture when creating cloud-native applications. It significantly shortens the time to market of new application features by changing, testing and deploying each service, individually, without affecting other services. A well-designed and

right-sized microservices architecture can help engineer an application that is stable, scalable and fault tolerant.

Unfortunately, those benefits come at the expense of additional complexities for which typical Enterprise Computing Frameworks like Java EE, now Jakarta EE, are not sufficient. Although each microservice can be implemented easily, managing, securing, and monitoring a highly distributed collection of microservices is complex. This is among others mainly because of a lack of a centralized runtime in the design of any distributed system.

THE MISSION OF MICROPROFILE

MicroProfile enables Java EE developers to leverage their existing skill set while shifting their focus from traditional 3-tier applications to microservices. MicroProfile APIs establish an optimal foundation for developing microservices-based applications by adopting a subset of the Java EE standards and extending them to address common microservices patterns.

MicroProfile specifications include:

[MicroProfile Config](#)

[MicroProfile Fault Tolerance](#)

[MicroProfile Health Check](#)

[MicroProfile Metrics](#)

[MicroProfile Open API](#)

[MicroProfile Rest Client](#)

[MicroProfile JWT Authentication](#)

[MicroProfile Open Tracing API](#)

A benefit of adopting MicroProfile is that its eleven specifications (at the moment of writing) are community driven, free, and open which encourages and has resulted in multiple implementations. Multiple implementations eliminates the risk of vendor lock-in and maintains the developers' freedom of choice. In addition, MicroProfile continues to evolve delivering roughly three annual releases, offering an opportunity to include both new and updated specifications to keep pace with developer needs.

All this is possible because of the great engagement and the collaborative work of the permanent growing MicroProfile community as excellently described by Mike Croft in his blog post "MicroProfile Turns One".

Lets do this!

Why Microservices?

The difficulties associated with developing, testing, fixing, and updating an application has relegated big monolithic applications to the past. Today, application architecture must support agile and continuous development by decomposing systems into smaller services focused on specific business domains. These domain-specific services can then be developed and modified independently according to evolving business needs, without impacting the system as a whole.

Decomposing a monolith into independent microservices on the whole or only partially while leaving the remaining functionality unchanged has many advantages. For example, each microservice is easy to understand, develop and maintain. A microservice can be deployed, scaled and run independently. Local changes can be done without the risk of unanticipated side-effects and potential faults are isolated by definition thanks to the microservices boundaries. Such advantages help shorten the time to market by facilitating advanced agility.

Unfortunately, a single application that consists of tens or maybe hundreds of microservices can also have its drawbacks. As mentioned before managing, securing, and monitoring a highly distributed collection of microservices with no central runtime, management or monitoring instance is complex.



Example Use-Case: MicroProfile e-Commerce

Let's take a look at a simple microservices-based, web-application called "MicroProfile e-Commerce":

With the help of the e-Commerce solution, currently customers can search for products, add them to their virtual shopping cart, and check-out at any time.

TO IMPLEMENT ABOVE ACTIVITIES REQUIRES AT LEAST FOUR SERVICES:

- >> a **customer service** that allows customer registration and self-administration;
- >> a **catalogue service** that maintains the available products;
- >> a **cart service** that manages the customers shopping carts;
- >> a **payment service** that enables the customers to pay during check-out

IN ADDITION WE WILL NEED...

- >> an **inventory service** that helps maintain the products availability or inventory;
- >> a **shipping service** for the products' delivery;
- >> a **search service** for an optimized highperformance product search; and
- >> a **recommendation service** for personalized product recommendations.

LET'S TAKE A DEEPER LOOK AT ONE USE-CASE: SHOPPING CART CHECK-OUT*.

*(*The following use case description is simplified for a better understanding)*

During check-out, the application has to do several things in parallel.

1. Check the availability of the requested product via inventory service.
2. Fulfill the payment via payment service.
3. Trigger the physical shipment via shipment service.

Each service, detailed above, by itself seems to be quite easy to implement, test, deploy and monitor. However what happens when it come to real life use-cases where services interact?



The coordination of the above mentioned steps can be done in two different ways:

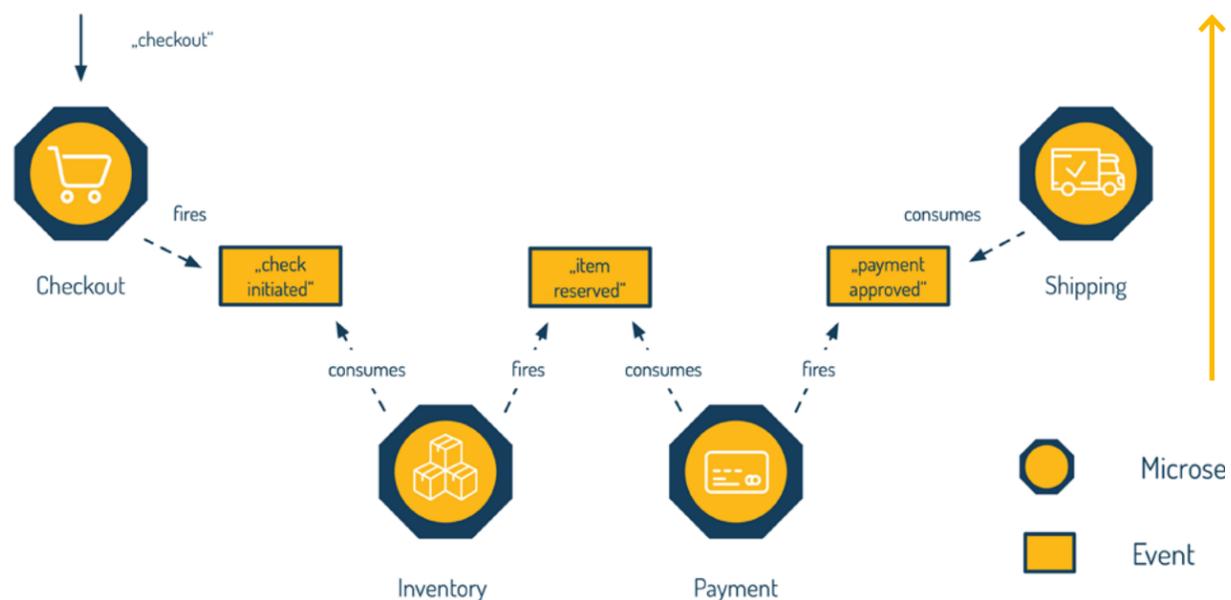
CHOREOGRAPHY

In a **choreography** based approach the workflow of the use-case is implicitly defined by a sequence of domain events and service actions. Each involved service signals a successful or faulty fulfilment of its part of the workflow via a corresponding domain event. Other services may listen to this domain event and start working on there part of the workflow after receiving it.

Let's assume that the cart service offers a convenient API to trigger the checkout workflow from the UI. After being called by

pressing the "checkout" button the service creates a domain event "checkout initiated" for customer Y and product X" which will be consumed by the inventory service. After checking the availability of the requested product, the inventory service will reserve the product and create a second domain event "product X reserved for customer Y" by itself. This domain event again is the signal for the payment service to fulfill the payment and create a third domain event "payment fulfilled for customer Y and product X" which triggers the physical shipment of the product to the customer (via shipment service). Figure 2 illustrates the choreography of the checkout workflow.

DISTRIBUTED CHECKOUT WORKFLOW VIA CHOREOGRAPHY



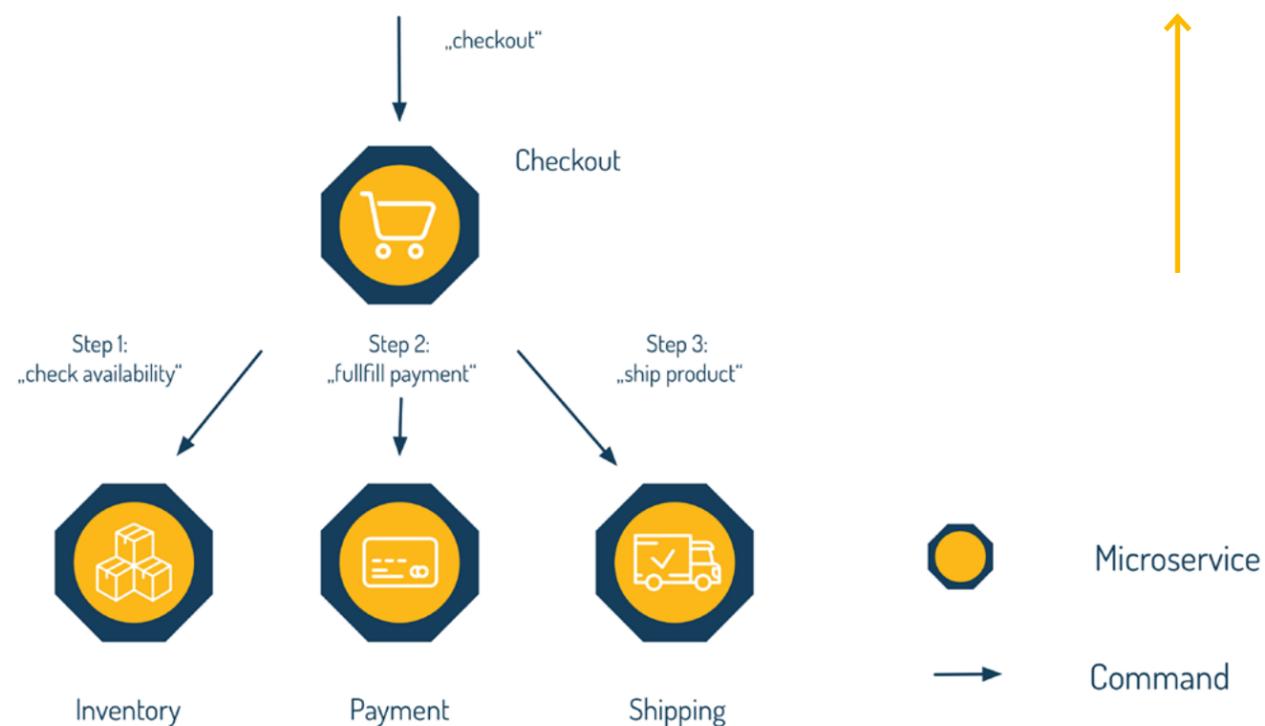
In an **orchestration** based approach the orchestrating service sends commands to the different services and receives their responses. This could be done synchronously via RESTful calls or asynchronously with the help of queues.

After being called from the UI by pressing the "checkout" button the cart service creates a command "reserve product X for customer Y" and sends it to the inventory service. The inventory service tries to reserve the product after receiving the request and sends back

ORCHESTRATION

a success response to the checkout service. After receiving the response the checkout service creates a second command to fulfill the workflow which is "fulfill payment for customer Y and product X" and sends it to the payment service. And so on. Figure 3 illustrates the orchestration of the checkout workflow.

DISTRIBUTED CHECKOUT WORKFLOW VIA ORCHESTRATION



Whether Choreography or Orchestration is used, there are still a lot of additional things to take care of in a microservices-based application. It is essential to make sure that each service by itself as well as the the system as a whole is up and running as expected from technical and functional point of view.

Cross-Cutting Concerns

Every system included in MicroProfile e-Commerce has a number of cross-cutting concerns that need to be addressed such as logging, monitoring of service health and metrics, fault tolerance, configuration, and security. In a microservices architecture, where there is no central point of control, these cross-cutting concerns are challenging.



LOGGING

Since MicroProfile e-Commerce is spread across multiple microservices, developers need a solution for distributed logging and tracing. Each service must be able to log and trace its specific part of the processing so that logging and tracing information across the system can be merged automatically and can be analyzed as a whole.



HEALTH CHECKS & METRICS

Due to the fact that we do not have a single service and therefore no single point of control, we also have to change the way we monitor our application state. For each microservice we need to know if it is healthy (health check) and performing as expected (metrics).



RESILIENCE & FAULT TOLERANCE

As the number of services grows, the odds of any one service failing also grows. If one of the involved services does not respond as expected, e.g. because of fragile network communication, we have to compensate for this exceptional situation. Therefore, we have to build up our microservices architecture to be resilient and fault tolerant by design. This means we must not only be able to detect but also to handle any issue automatically.



CONFIGURATION

The fact that there is no central runtime also implies that there is no single point of configuration. Let's assume that two of the involved microservices, e.g. catalogue service and search service, share a company cloud account. In that case, they may want to share a common configuration. Alternatively, it may be helpful that a microservice can access configurations from multiple sources in a homogeneous and transparent way.



AUTHENTICATION & AUTHORIZATION

During checkout, the customers of our MicroProfile e-Commerce solution do not want to log in repeatedly to every participating microservice. Instead, a mechanism to handle distributed authentication and authorization is required. Due to the stateless character of microservices the solution must offer security context propagation in a web-friendly way. In addition the security solution must be verifiable to make sure that the original service call is not forged in any way.



STANDARDIZED INTERFACE DOCUMENTATION

From the microservices developer's point of view, it is important to understand how to interact with any of the other microservices and how to test that the services APIs are still valid and backward compatible. Therefore standardized API documentation that can be also used for API testing is needed.



MicroProfile to the rescue

After pointing out the cross-cutting concerns in a microservices application, the big question is how do we deal with these concerns?

One option is to find a separate solution for each cross-cutting concern (e.g. logging, tracing, health checks, metrics, authentication, documentation). The problem with this approach is that we might end up in a bunch of incompatible solutions that may not have been tested together and are evolving without consideration to other services, making overall testing and version management more difficult. In large organizations, different teams may choose different

solutions for the same problem, making it more difficult for developers to switch projects.

Another approach is to find an “all-around package” that addresses many cross-cutting concerns. Yet an all-in-one solution could result in vendor lock-in constraining organizations to the release schedule and capabilities of a single vendor.

Up Up & Away

What we are looking for is a consistent and interoperable way to handle these challenges without the risk of vendor lock-in. This is the mission of MicroProfile. MicroProfile creates specifications that focus on portability and interoperability while remaining vendor agnostic: Write once, run everywhere!

MicroProfile utilizes a very small subset of Java EE APIs to define its core programming model:

- CDI 2.0 for Dependency Injection
- JAX-RS 2.1 for REST APIs
- JSON P 1.1 for JSON Parsing

JSON-B 1.0 for JSON Binding
To name a few...

MicroProfile’s mission is to close the gap between the Enterprise Java and the real-life challenges of microservices architectures.

The community has been using Enterprise Java technologies like Java EE, today re-branded as [Jakarta EE](#), within microservices architectures in innovative ways for quite a while (see [Jakarta EE Community Survey](#) for details)! While the community members have been innovating independently, MicroProfile now allows Open Source collaboration and innovation wherever there is a commonality. By leveraging these commonalities, such as the afore-mentioned cross-cutting concerns,

within a baseline platform, developers will continue to benefit from rapid innovation, application portability, and interoperability, with multiple vendors to choose from.

Vendors, the community, and open source projects bring their solutions to the real world while at the same time collecting end user feedback. In common areas, where there is enough stability and functionality individual solutions can result in standardization.



What is under the MicroProfile hood?

The [MicroProfile project](#) specifies several solution blocks - a.k.a. APIs - each of them addressing a specific topic.

APIS OF MICROPROFILE 3.0 APIS



MicroProfile 3.0

- = Updated
- = No change from last release (MicroProfile 2.2)

power



Eclipse
MicroProfile 3.0





2019

2019



1.3 MICROPROFILE CONFIGURATION

MicroProfile Config externalizes application configuration by providing microservices with the means to obtain configuration properties through several environment-aware sources. These sources can be accessed in parallel both internal and external to the application and made available through dependency injection or lookup in a unified way. MicroProfile Config also allows developers to implement and register their own configuration sources in a portable way, e.g. for reading configuration values from a shared database in an application cluster.

Microservices often contain functionality that behaves slightly differently depending on the deployment. This might be different REST endpoints (e.g. depending on the customer for whom a WAR is deployed). Or it might even be entire features that need to be switched on and off depending on the

installation. Regardless of its deployment, microservices ought to be possible without the need to repackage the whole application binary.

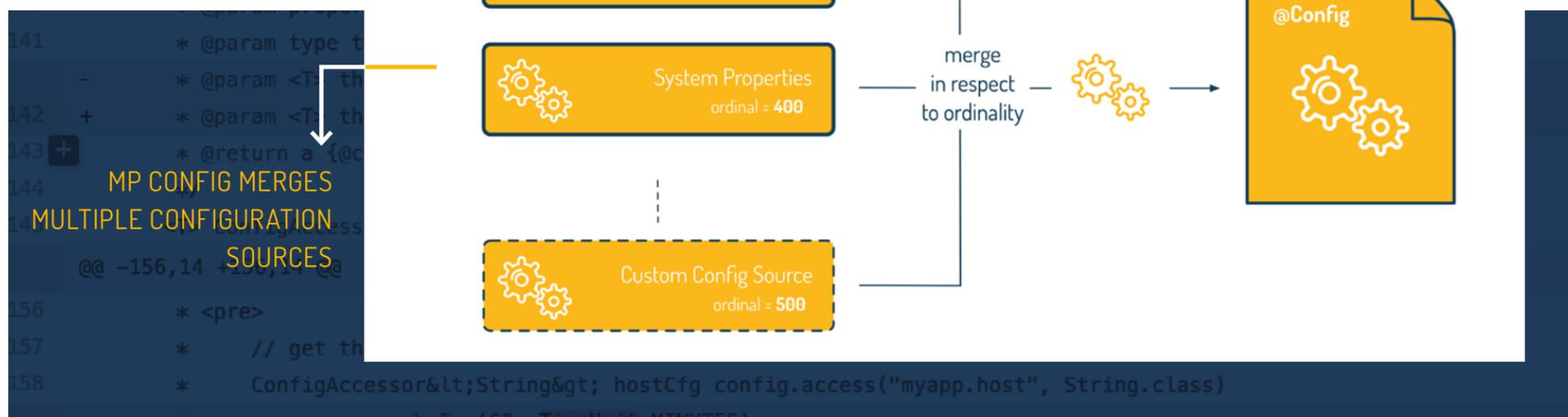
MicroProfile Config provides type-safe injection of configuration properties, no matter which configuration source they originate from. Developers can decide if the configuration properties should be read at service or application start or just-in-time with a given refresh rate. Default and custom converters take the responsibility for the type conversion needed to guarantee type-safeness.

The following example shows how to use Config via injection to configure the shopping cart service of our MicroProfile e-Commerce application:

```
// inject a boolean property
@Inject @ConfigProperty("ONE_CLICK_CHECKOUT_ALLOWED_KEY")
Boolean oneClickCheckoutAllowed;

// inject property with default value
@Inject @ConfigProperty("MAX_ITEM_COUNT_KEY", defaultValue="100")
Integer maxItemCount;

// inject a custom class property
@Inject @ConfigProperty("MINIMUM_AMOUNT_KEY")
Amount minimumAmount;
```





2.0 MICROPROFILE FAULT TOLERANCE

MicroProfile Fault Tolerance enables us to build resilient microservices by separating the execution logic from business logic. Key aspects of the Fault Tolerance API includes well known resilience patterns like TimeOut, RetryPolicy, Fallback, Bulkhead, and Circuit Breaker processing.

In a distributed system, like a microservices-based architecture, it is increasingly important to build in fault tolerance by default. Fault tolerance is about leveraging different strategies to guide the execution and result of some logic. Retry policies, bulkheads, and circuit breakers are popular concepts in this area. They dictate whether and when executions should take place, while fallbacks offer an alternative result when an execution does not complete successfully.

The main design of MP Fault Tolerance is to separate the execution logic from execution. The execution can be configured with fault tolerance policies, such as RetryPolicy, Fallback, Bulkhead and CircuitBreaker. The Fault Tolerance specification defines a standard API and approach for applications to follow in order to achieve resilience.

The following example shows a simple timeout and fallback scenario. If the originally called method `getAllProducts` could not handle the request within a given period of time the fallback method, `getAllProductsCallback`, is called automatically.

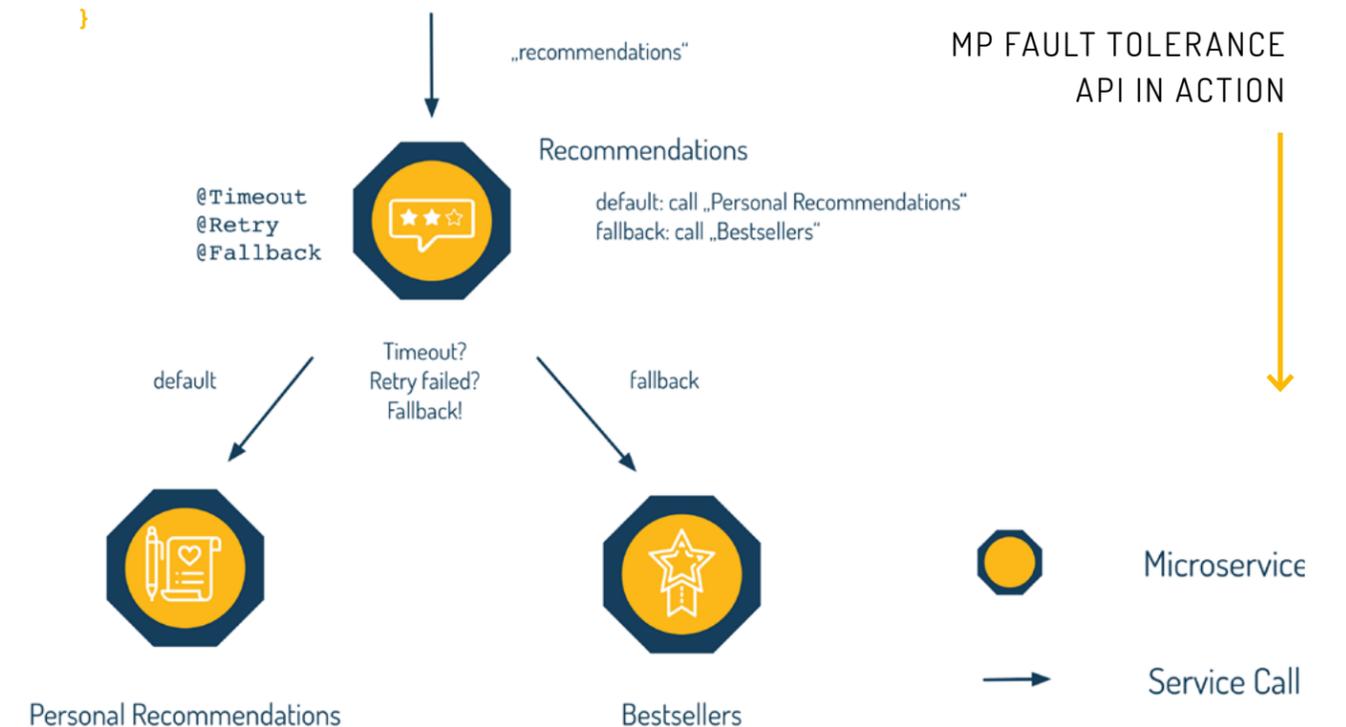
Listing: Example of MicroProfile Fault Tolerance usage.

```

@GET
@Timeout(500)
@Fallback(fallbackMethod = "getBestsellersFallback")
public Response getPersonalRecommendations() throws InterruptedException {
    // retrieve personal recommendations by delegating the call to
    // Personal Recommendations Microservice
    ...
}

public Response getBestsellersFallback() {
    // retrieve bestsellers instead of personal recommendations as fallback
    // by delegating the call to Bestsellers Service
    ...
}

```





2.0

MICROPROFILE HEALTH CHECK

[MicroProfile Health Check](#) lets developers define and expose a domain-specific microservices health status ("UP" or "DOWN") so unhealthy services can be restarted by the underlying environment. Health Check is a "machine-to-machine" API, where orchestration services like Kubernetes can automatically detect and restart unhealthy services without manual intervention.

Health checks are used to determine both the liveness and readiness of a service. Even though a service may be "live" and running it may not yet be ready to accept traffic, such as when a database connection is being established. This has a direct impact on the traffic routing, because traffic

should only be directed to a service that is both live and ready. The MP Health Check has been designed with Kubernetes, Cloud Foundry and Mesos in mind, so leveraging the provided orchestrator features is natural.

Determining the state of a service is often more complex than checking whether an HTTP endpoint is available. It can be composed by a set of verification procedures. Multiple domain-specific health checks can easily be added to a microservice by implementing the corresponding HealthCheck interface. MP Health Check discovers and compose these procedures to compute the overall outcome.

The following example demonstrates how to implement a simple readiness check to make sure that the connection pool of the customer service of our MicroProfile e-Commerce example is up and running as expected and therefore the microservice is ready to process requests.

Examples of MicroProfile Health Check usage.

```
@Readiness
@ApplicationScoped
public class ConnectionPoolCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        if (isConnectionPoolHealthy()) {
            return HealthCheckResponse("customer-cp")
                .up()
                .build();
        } else {
            return HealthCheckResponse("customer-cp")
                .down()
                .build();
        }
    }

    // check that there is no connection shortage
    private boolean isConnectionPoolHealthy() { ... }
}
```

As shown above a health check for readiness allows a third party service to know if a microservice is ready to process requests or not. In contrast a health check for liveness - indicated via @Liveness annotation - allows a third party service to determine if a microservice is running. This means that if this procedure fails the microservice can be discarded (terminated, shutdown) without any negative side-effects.





2.0

MICROPROFILE METRICS

MicroProfile Metrics delivers details about the microservices runtime behavior by providing a unified way for MicroProfile servers to export monitoring data to management agents. Metrics also provides a common Java API for exposing their telemetry data.

In order to run a service reliably developers need monitoring. There is already JMX as a standard to expose metrics, but remote-JMX is not easy to deal with and does not fit well in a polyglot environment where other services are not running in a JVM environment.

To enable monitoring in such a polyglot environment it is necessary that all MicroProfile implementations follow a certain standard with respect to the (base) API path, data types involved, always available metrics, and return codes used.

Metrics serves to pinpoint issues, providing long-term trend data for capacity planning, and pro-active discovery of issues (e.g. disk usage growing without bounds) via a simple to use RESTful API. Metrics can also help scheduling-systems decide when to scale the application to run on more or fewer machines.

METRICS SUPPORTS DIFFERENT DATA TYPES:

COUNTER:

a monotonically increasing numeric value (e.g. total number of requests received or the total number of concurrently active HTTP sessions).

METER:

a metric which tracks mean throughput and one-, five-, and fifteen-minute exponentially-weighted moving average throughput (e.g. number of calls a function processes).

HISTOGRAM:

a metric which calculates the distribution of a value (e.g. items per checkout in an e-commerce application).

TIMER:

a metric which aggregates timing durations and provides duration statistics, plus throughput statistics (e.g. response time or calculation time of complex logic).

GAUGE:

a metric that is sampled to obtain its value (e.g. CPU temperature or disk usage).

The following example demonstrates how to measure the duration of the check-out process of our example application MicroProfile e-Commerce:

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Timed(absolute = true,
      name = „microprofile.ecommerce.checkout“,
      displayName = „check-out time“,
      description = „time of check-out process in ns“,
      unit = MetricUnits.NANOSECONDS)
public Response checkOut(...) {
    // do some check-out specific business logic
    ...
    return Response.ok(...) build();
}
```

Listing: Examples of MicroProfile Metrics usage via annotations.

To be able to differentiate common metrics from vendor or application specific metrics, MP Metrics also supports three different sub-resources (aka scopes):

base: metrics that all MicroProfile vendors have to provide
 vendor: vendor specific metrics (optional)
 application: application-specific metrics (optional)





1.1 MICROPROFILE OPEN API

Exposing APIs has become an essential part of all modern applications. At the center of this revolution known as the API Economy, we find RESTful APIs, which transforms any application into language-agnostic services that can be called from anywhere: on-premises, private cloud, public cloud, etc.

For the clients and providers of these services to connect, there needs to be a clear and complete contract. Similar to the WSDL contract for legacy Web Services, the [OpenAPI Specification](#) (OAS) is the contract for RESTful Services.

OAS defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code,

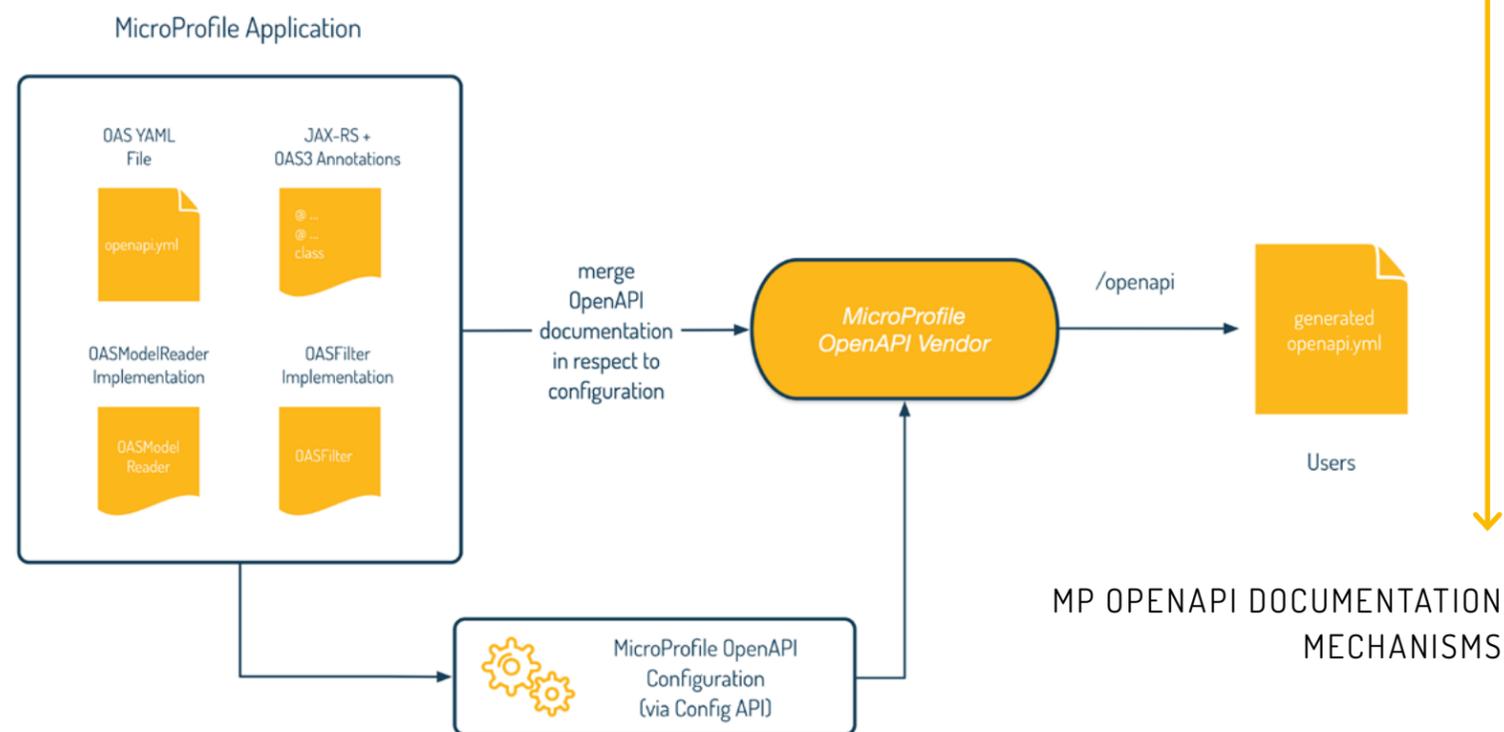
additional documentation, or inspection of network traffic. The description can be understood as a clear and complete contract.

The [MicroProfile Open API](#) specification provides a set of Java interfaces and programming models that allow developers to natively produce OpenAPI v3 documents.

```
@OpenAPIDefinition(
  servers = {
    @Server(
      description = "definition server 1",
      url = "http://{var1}.definition1/{var2}",
      variables = {
        @ServerVariable(
          name = "var1",
          description = "var 1",
          defaultValue = "1",
          allowedValues = {"1", "2"}
        )
      }
    )
  }
)
class Server {
  description = "class server 1",
  url = "http://{var1}.class1/{var2}",
  variables = {
    @ServerVariable(
      name = "var1",
      description = "var 1",
      defaultValue = "1",
      allowedValues = {"1", "2"}
    )
  }
}
```

THE APPLICATION DEVELOPERS HAVE A FEW CHOICES:

- » Augment existing JAX-RS annotations with the OpenAPI Annotations. Using annotations means developers don't have to re-write the portions of the OpenAPI document that are already covered by the JAX-RS framework (e.g. the HTTP method of an operation).
- » Use the Programming model to provide a bootstrap (or complete) OpenAPI model tree. Additionally, a Filter is described which can update the OpenAPI model after it has been built from.





1.3

MICROPROFILE REST CLIENT

MicroProfile Rest Client **simplifies building REST Clients** by providing a type-safe approach for invoking RESTful services over HTTP. The MicroProfile Rest Client builds upon the [JAX-RS 2.1 APIs](#) for consistency and ease-of-use.

Using MicroProfile Rest Client to write client applications allows for a more natural and model-centric coding style, while the underlying implementation handles the communication between client and service by automatically making the HTTP connection and serializing the model object to JSON (or XML etc.), so that the remote service can process it.

Rest Client allows developers use plain Java interfaces to invoke a RESTful service. It specifies the mapping between a method on an interface and a REST request using existing JAX-RS annotations. In many cases, an annotated client interface looks very much like the corresponding REST resource. It's easy to reuse the same REST client to avoid code duplication. A REST Client interface is also a very natural way to specify and document the contract between the client and the RESTful service. Using Rest Client thus reduces a lot of boilerplate constructs and leads to a cleaner code.

For example, to invoke the checkout process from another service, the service only needs to define a Java interface with appropriate annotations. Then a container or a builder will create a proxy capable of mapping between method invocations and REST requests. Knowing the JAX-RS API, this is very straightforward:

```
@RegisterRestClient
@Path("/shopping")
public interface ShoppingServiceClient {

    @POST
    @Path("/checkout")
    @Produces(MediaType.APPLICATION_JSON)
    public Response checkOut(...);
}
```

Interfaces annotated with `@RegisterRestClient` can be simply injected into any CDI bean with the `@RestClient` qualifier annotation, for example:

```
@Inject @RestClient
ShoppingServiceClient shopping;
```

Location of the RESTful service depends on the environment. It needs to be provided as a MP Config configuration property. The name of the property is derived from the full class name of the interface and the property specifies the location for all injected proxies.

If needed, a proxy for the interface can be created programmatically with a builder. This also allows specifying the location for each proxy separately:

```
ShoppingServiceClient shopping = RestClientBuilder.newBuilder()
    .baseUri(new URI("http://localhost:8080/"))
    .build(ShoppingServiceClient.class);
```



1.1 MICROPROFILE JWT

MicroProfile JWT Authentication defines a format of **JSON Web Token (JWT)** used as the basis for **interoperable authentication and authorization** by providing role-based access control (RBAC) microservice endpoints using OpenID Connect (OIDC).

The security requirements that involve microservice architectures are strongly related with RESTful Security. In a RESTful architecture style, services are usually stateless and any security state associated with a client is sent to the target service on every request. This allows services to re-create a security context for the caller and perform both authentication and authorization checks.

One of the main strategies to propagate the security state from clients to services, or even from services to services, involves the use of security tokens. In fact, the main security protocols in use today are based on security tokens such as OAuth2, OpenID Connect, SAML, WS-Trust, WSFederation and others. While some of these standards are oriented to identity federation, they all share a common concept regarding security tokens and token-based authentication.

```

23 - <version>1.1.1</version>
24 + <version>1.2-SNAPSHOT</version>
25 FOR RESTFUL BASED MICROSERVICES, SECURITY
26 TOKENS OFFER A VERY LIGHTWEIGHT AND
27 INTEROPERABLE WAY TO PROPAGATE IDENTITIES
28 ACROSS DIFFERENT SERVICE WHERE A TOKEN
29 FORMAT IS WELL KNOWN OR SERVICES CAN INVOKE
30 A SEPARATE SERVICES SO THAT:
31
32 » Services don't need to store any state about clients or users.
33 » Services can enforce authorization policies based on any information within a security token.
34
35 » Services can verify the token validity.
36
37 » Services need support both delegation and impersonation of identities.
38
39 » Services can identify the caller by introspecting the token.
40
41 <groupId>org.eclipse.microprofile.jwt</groupId>
42 <artifactId>microprofile-jwt-auth-parent</artifactId>
43 <version>1.1.1</version>

```

Today, the most common solutions involving RESTful and microservices security are based on OAuth2, OpenID Connect (OIDC) and JSON Web Tokens (JWT) standards.

MP JWT heavily relies on the above standards, therefore it offers an API making it very easy to access, verify and propagate JSON Web Tokens in a Java Security compatible way.



1.3

MICROPROFILE OPENTRACING

[MicroProfile OpenTracing](#) enables services to easily **participate in a distributed tracing environment** by defining an API.

Distributed tracing is particularly important in a microservices environment where a request typically flows through multiple services. To accomplish distributed tracing, each service must be instrumented to log messages with a correlation id that may have been propagated from an upstream service. A common companion to distributed trace logging is a service where the distributed trace records can be stored (see also examples on [opentracing.io](#)). The storage service for distributed trace records can provide cross service trace records associated with particular request flows.

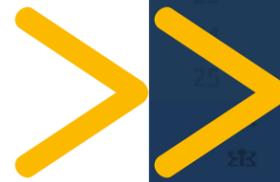
The OpenTracing project's purpose is to provide a standard API for instrumenting microservices for distributed tracing. If every microservice is instrumented for distributed tracing using the OpenTracing API, then (as long as an implementing library exists for the microservice's language), the microservice can be configured at deploy time to use a common system implementation to perform the log record formatting and cross-service correlation id propagation. The common implementation ensures that correlation ids are propagated in a way that is understandable to all services, and log records are formatted in a way that is understandable to the server for distributed trace record storage.

In order to make MicroProfile distributed tracing friendly, it allows distributed tracing to be enabled on any MicroProfile application, without having to explicitly add distributed tracing code to the application.

```

21 21      <!-- This is just for now and will not work if the API has a separ
22 22      <groupId>org.eclipse.microprofile.opentracing</groupId>
23 23      <artifactId>microprofile-opentracing-parent</artifactId>
24 24      <version>1.3.1</version>
25 25      <version>1.3.2-SNAPSHOT</version>
26 26
27 27
@@ -21,7 +21,7 @@
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29
30 30
31 31
32 32
33 33
34 34
35 35
36 36
37 37
38 38
39 39
40 40
41 41
42 42
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
51 51
52 52
53 53
54 54
55 55
56 56
57 57
58 58
59 59
60 60
61 61
62 62
63 63
64 64
65 65
66 66
67 67
68 68
69 69
70 70
71 71
72 72
73 73
74 74
75 75
76 76
77 77
78 78
79 79
80 80
81 81
82 82
83 83
84 84
85 85
86 86
87 87
88 88
89 89
90 90
91 91
92 92
93 93
94 94
95 95
96 96
97 97
98 98
99 99
100 100
101 101
102 102
103 103
104 104
105 105
106 106
107 107
108 108
109 109
110 110
111 111
112 112
113 113
114 114
115 115
116 116
117 117
118 118
119 119
120 120
121 121
122 122
123 123
124 124
125 125
126 126
127 127
128 128
129 129
130 130
131 131
132 132
133 133
134 134
135 135
136 136
137 137
138 138
139 139
140 140
141 141
142 142
143 143
144 144
145 145
146 146
147 147
148 148
149 149
150 150
151 151
152 152
153 153
154 154
155 155
156 156
157 157
158 158
159 159
160 160
161 161
162 162
163 163
164 164
165 165
166 166
167 167
168 168
169 169
170 170
171 171
172 172
173 173
174 174
175 175
176 176
177 177
178 178
179 179
180 180
181 181
182 182
183 183
184 184
185 185
186 186
187 187
188 188
189 189
190 190
191 191
192 192
193 193
194 194
195 195
196 196
197 197
198 198
199 199
200 200
201 201
202 202
203 203
204 204
205 205
206 206
207 207
208 208
209 209
210 210
211 211
212 212
213 213
214 214
215 215
216 216
217 217
218 218
219 219
220 220
221 221
222 222
223 223
224 224
225 225
226 226
227 227
228 228
229 229
230 230
231 231
232 232
233 233
234 234
235 235
236 236
237 237
238 238
239 239
240 240
241 241
242 242
243 243
244 244
245 245
246 246
247 247
248 248
249 249
250 250
251 251
252 252
253 253
254 254
255 255
256 256
257 257
258 258
259 259
260 260
261 261
262 262
263 263
264 264
265 265
266 266
267 267
268 268
269 269
270 270
271 271
272 272
273 273
274 274
275 275
276 276
277 277
278 278
279 279
280 280
281 281
282 282
283 283
284 284
285 285
286 286
287 287
288 288
289 289
290 290
291 291
292 292
293 293
294 294
295 295
296 296
297 297
298 298
299 299
300 300
301 301
302 302
303 303
304 304
305 305
306 306
307 307
308 308
309 309
310 310
311 311
312 312
313 313
314 314
315 315
316 316
317 317
318 318
319 319
320 320
321 321
322 322
323 323
324 324
325 325
326 326
327 327
328 328
329 329
330 330
331 331
332 332
333 333
334 334
335 335
336 336
337 337
338 338
339 339
340 340
341 341
342 342
343 343
344 344
345 345
346 346
347 347
348 348
349 349
350 350
351 351
352 352
353 353
354 354
355 355
356 356
357 357
358 358
359 359
360 360
361 361
362 362
363 363
364 364
365 365
366 366
367 367
368 368
369 369
370 370
371 371
372 372
373 373
374 374
375 375
376 376
377 377
378 378
379 379
380 380
381 381
382 382
383 383
384 384
385 385
386 386
387 387
388 388
389 389
390 390
391 391
392 392
393 393
394 394
395 395
396 396
397 397
398 398
399 399
400 400
401 401
402 402
403 403
404 404
405 405
406 406
407 407
408 408
409 409
410 410
411 411
412 412
413 413
414 414
415 415
416 416
417 417
418 418
419 419
420 420
421 421
422 422
423 423
424 424
425 425
426 426
427 427
428 428
429 429
430 430
431 431
432 432
433 433
434 434
435 435
436 436
437 437
438 438
439 439
440 440
441 441
442 442
443 443
444 444
445 445
446 446
447 447
448 448
449 449
450 450
451 451
452 452
453 453
454 454
455 455
456 456
457 457
458 458
459 459
460 460
461 461
462 462
463 463
464 464
465 465
466 466
467 467
468 468
469 469
470 470
471 471
472 472
473 473
474 474
475 475
476 476
477 477
478 478
479 479
480 480
481 481
482 482
483 483
484 484
485 485
486 486
487 487
488 488
489 489
490 490
491 491
492 492
493 493
494 494
495 495
496 496
497 497
498 498
499 499
500 500
501 501
502 502
503 503
504 504
505 505
506 506
507 507
508 508
509 509
510 510
511 511
512 512
513 513
514 514
515 515
516 516
517 517
518 518
519 519
520 520
521 521
522 522
523 523
524 524
525 525
526 526
527 527
528 528
529 529
530 530
531 531
532 532
533 533
534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
601 601
602 602
603 603
604 604
605 605
606 606
607 607
608 608
609 609
610 610
611 611
612 612
613 613
614 614
615 615
616 616
617 617
618 618
619 619
620 620
621 621
622 622
623 623
624 624
625 625
626 626
627 627
628 628
629 629
630 630
631 631
632 632
633 633
634 634
635 635
636 636
637 637
638 638
639 639
640 640
641 641
642 642
643 643
644 644
645 645
646 646
647 647
648 648
649 649
650 650
651 651
652 652
653 653
654 654
655 655
656 656
657 657
658 658
659 659
660 660
661 661
662 662
663 663
664 664
665 665
666 666
667 667
668 668
669 669
670 670
671 671
672 672
673 673
674 674
675 675
676 676
677 677
678 678
679 679
680 680
681 681
682 682
683 683
684 684
685 685
686 686
687 687
688 688
689 689
690 690
691 691
692 692
693 693
694 694
695 695
696 696
697 697
698 698
699 699
700 700
701 701
702 702
703 703
704 704
705 705
706 706
707 707
708 708
709 709
710 710
711 711
712 712
713 713
714 714
715 715
716 716
717 717
718 718
719 719
720 720
721 721
722 722
723 723
724 724
725 725
726 726
727 727
728 728
729 729
730 730
731 731
732 732
733 733
734 734
735 735
736 736
737 737
738 738
739 739
740 740
741 741
742 742
743 743
744 744
745 745
746 746
747 747
748 748
749 749
750 750
751 751
752 752
753 753
754 754
755 755
756 756
757 757
758 758
759 759
760 760
761 761
762 762
763 763
764 764
765 765
766 766
767 767
768 768
769 769
770 770
771 771
772 772
773 773
774 774
775 775
776 776
777 777
778 778
779 779
780 780
781 781
782 782
783 783
784 784
785 785
786 786
787 787
788 788
789 789
790 790
791 791
792 792
793 793
794 794
795 795
796 796
797 797
798 798
799 799
800 800
801 801
802 802
803 803
804 804
805 805
806 806
807 807
808 808
809 809
810 810
811 811
812 812
813 813
814 814
815 815
816 816
817 817
818 818
819 819
820 820
821 821
822 822
823 823
824 824
825 825
826 826
827 827
828 828
829 829
830 830
831 831
832 832
833 833
834 834
835 835
836 836
837 837
838 838
839 839
840 840
841 841
842 842
843 843
844 844
845 845
846 846
847 847
848 848
849 849
850 850
851 851
852 852
853 853
854 854
855 855
856 856
857 857
858 858
859 859
860 860
861 861
862 862
863 863
864 864
865 865
866 866
867 867
868 868
869 869
870 870
871 871
872 872
873 873
874 874
875 875
876 876
877 877
878 878
879 879
880 880
881 881
882 882
883 883
884 884
885 885
886 886
887 887
888 888
889 889
890 890
891 891
892 892
893 893
894 894
895 895
896 896
897 897
898 898
899 899
900 900
901 901
902 902
903 903
904 904
905 905
906 906
907 907
908 908
909 909
910 910
911 911
912 912
913 913
914 914
915 915
916 916
917 917
918 918
919 919
920 920
921 921
922 922
923 923
924 924
925 925
926 926
927 927
928 928
929 929
930 930
931 931
932 932
933 933
934 934
935 935
936 936
937 937
938 938
939 939
940 940
941 941
942 942
943 943
944 944
945 945
946 946
947 947
948 948
949 949
950 950
951 951
952 952
953 953
954 954
955 955
956 956
957 957
958 958
959 959
960 960
961 961
962 962
963 963
964 964
965 965
966 966
967 967
968 968
969 969
970 970
971 971
972 972
973 973
974 974
975 975
976 976
977 977
978 978
979 979
980 980
981 981
982 982
983 983
984 984
985 985
986 986
987 987
988 988
989 989
990 990
991 991
992 992
993 993
994 994
995 995
996 996
997 997
998 998
999 999
1000 1000

```





2019

2019




Lessons learned from real-world microservice adoption

By [Matjaz B. Juric](#)

The companies that are using Java microservices for their production and even mission critical systems come from different industries, including energy, banking, insurance, manufacturing, transport and logistics, public administration, and others.

Companies are mainly using microservices for two kinds of projects, development of new digital

services and applications for digital channels, and gradual migration of existing applications from application servers to microservices. With each and every company that has adopted microservices, we, at Kumuluz, have observed one common thing – developers simply love microservices. In this section, we will share those findings.



The first common use case is digital services and applications for mobile and web applications that offer customers superb digital user experience, customer engagement, social and IoT integration, and other aspects of digital transformation. Such solutions typically require APIs on the backend, REST APIs or event streaming. MicroProfile microservices are the perfect choice for implementing back-end for such applications. They are fast, lightweight and easy to scale as they work perfectly with containers and can be deployed on Kubernetes-like environments, either on premise or hosted in the cloud. With microservices, it is easier to achieve scalability, high availability and resilience.

The major advantage for companies that have traditionally based their developments on Java and Java EE (now Jakarta EE), is that the transition from classic application servers to MicroProfile is straightforward. Developers skilled in Java/Java EE can start using MicroProfile in relatively short time, without having to learn new technologies, languages or APIs.

This brings us to the second common use case, the gradual migration from monolithic applications to microservices. Typical Java EE applications are large and complex. They have been developed over longer periods of time and often they are mission critical. Therefore companies cannot afford to rewrite them for microservices. Migrating to MicroProfile allows to gradual transformation from a monolithic, application server based application to microservices. A common practice is to develop major changes or new functionalities that need to be added to such applications as microservices (for example, common new functionalities for banking applications in EU are PSD2 (Payment Service Directive 2) APIs), leaving the rest as-is on the application server. MicroProfile can work hand-in-hand with application servers. Another practice is to migrate certain functionalities to microservices. Because MicroProfile microservices use the core subset of Java EE technologies,, it is relatively easy to migrate existing code to microservices.



Microservices also allow companies to easily upgrade the version of Java. With new versions of Java scheduled every six months, developers can choose a new version of Java only for selected microservices and not migrate everything at once, which is a common practice with application servers. Finally, adding other languages, such as Node.js or Go, for selected functionalities is straightforward with microservices.

THE BENEFITS OF MICROSERVICES ARE NOT LIMITED TO TECHNOLOGY RELATED ADVANTAGES ONLY. THEY ALSO BRING IMPORTANT ORGANIZATION BENEFITS.

Companies using MicroProfile today have considerably improved their agility of the development and shortened time to market. This is achieved with the appropriate organization of their teams, organized around microservices allowing parallelization of development – and deployment. In other words, teams work in parallel and develop, deploy and test microservices independent of each other.

MICROSERVICE APPROACH BRINGS CONSIDERABLE TIMESAVING AND IMPROVES AGILITY AND TIME-TO-MARKET BY ORDER OF MAGNITUDE. THIS IS SOMETHING BUSINESS PEOPLE LOVE TO SEE.

The experience of companies that have adopted microservices also shows that microservices require automation of infrastructure and DevOps. It is almost impossible to gain benefits of microservices if companies do not have DevOps in place and if their Continuous Delivery cycle is not fully automated.

ADVANTAGES OF MICROSERVICE ARCHITECTURE

Business related advantages Technology related advantages

Increased business agility and flexibility

Fast, lightweight

Order of magnitude shorter time-to-market

Scalable

Better fulfilment of business requirements

Highly available, fault tolerant, resilient

Faster digital transformation

Docker and Kubernetes ready

Parallel development and automation

Deployable on premise or cloud hosted

Improved Business-IT alignment and lower technical debt

Easy migration from Java/Java EE to microservices

Preserved existing investment in technology and know-how

Should be combined with DevOps, Continuous Delivery and automated testing

In conclusion, the experiences of companies that are using MicroProfile in the production for their mission critical systems, are mostly positive. The most important lesson is that companies have to address all aspects of the microservice architecture. As more developers are encouraged by their employers to do so, companies will definitely see the advantages in both technology and business.



Partner Network

By [Markus Eisele](#)

Enterprise Java technologies like Java EE have evolved with the industry for nearly two decades to support distributed application architectures based on RMI/IIOP, Web Services, and REST. MicroProfile is the next step in that evolution.

MicroProfile is possible because of the broad support from all its active members and the extensive partner network. The partner network gives this project a competitive advantage and the opportunity to access a broader range of resources and existing expertise.

In January 2017, MicroProfile became a member of the Eclipse Incubation Projects group.

INDIVIDUALS, ORGANIZATIONS, VENDORS

The original MicroProfile efforts was kicked off and announced in late June 2016 by Red Hat, IBM, Tomitribe, Payara and the London Java Community. A lot has happened since then. MicroProfile v 1.0 was released on September 19, 2016. Its implementation interoperability was demonstrated on November 2016 at Devovx Antwerp, where Red Hat, IBM, Tomitribe, and Payara demonstrated a unified web application with underlying microservices which had been developed separately by each vendor using MicroProfile. In addition, MicroProfile became part of the Eclipse Foundation as an incubation project back in December 14, 2016. New members have joined MicroProfile, such as SOUJava, Hazelcast, Fujitsu, Lightbend, Microsoft, Oracle, Hammock, and KumuluzEE.



CURRENTLY AVAILABLE IMPLEMENTATIONS

Unlike the Java EE standard, MicroProfile does not have a reference implementation but only TCKs (test compatibility kits) to pass. Each runtime provider is self-motivated to implement new APIs as soon as possible and pass the corresponding TCK. At the moment of writing this paper, eight vendors provide full or partial implementations of MicroProfile.

Payara

[Payara Server](#)

[Payara Micro](#)

Tomitribe

[Apache TomEE](#)

IBM

[Open Liberty](#)

[WebSphere Liberty](#)

Fujitsu

[Launcher](#)

Hammock

[Hammock](#)

Red Hat

[JBoss EAP](#)

[Thorntail](#)

[WildFly](#)

[Red Hat Application Runtimes](#)

SmallRye

[SmallRye](#)

Kumuluz

[KumuluzEE](#)

Innovation & Sandboxing



MicroProfile has many serviceable features, enabling it to address most of the challenges of a microservices architecture. But there is more to come.

As mentioned before the MicroProfile approach allows interested parties and partners to continue to innovate quickly and efficiency while collaborating where there is a commonality. Therefore MicroProfile supports its community by offering a [sandbox](#) for incubating ideas and code examples that eventually turn into new specifications in the near future. The sandbox is meant to experiment and share code and documents and allow discussions with the rest of the MicroProfile community.

After having reached an adequate degree of maturity, the initiator of the sandbox project can ask for a MicroProfile repository and in this way start the MicroProfile [feature init process](#). Of course experienced MicroProfile project committers will assist if desired.

The MicroProfile process by definition facilitates the engagement of the community while at the same time preventing any kind of vendor lock-in.

Just to give an example, at the time of writing, there are four sandbox projects available and up for discussion for the MicroProfile community.

SANDBOX PROJECT “LONG RUNNING ACTIONS”

The proposal for [Long Running Actions](#) introduces APIs for services to coordinate activities. The main thrust of the proposal introduces an API for loosely coupled services to coordinate long running activities in such a way as to guarantee a globally consistent outcome without the need to take locks on data.

SANDBOX PROJECT “REACTIVE”

The Reactive working group is defining several specification to make MicroProfile applications more event-driven, asynchronous and efficient.

The [Reactive Streams Operators](#) specification propose a set of operators for Reactive Streams. By mapping Java Streams API but for Reactive Streams, it provide a natural API to deal with stream of data, enforcing error propagation, completion signals, and back-pressure.

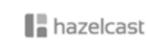
The [Reactive Messaging](#) proposal explores the question „what if Java offered a new API for handling streams of messages - either point to point or from a message broker - based on the JDK 9 Flow API or alternatively on the JDK8 compatible Reactive Streams API - that was lighter weight and easier to use than JMS/MDBs“.





2019

2019



SANDBOX PROJECT “GRAPHQL”

The proposal for [GraphQL](#) is an API specification making GraphQL available for microservice development.

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. It provides an alternative, though not necessarily a replacement for REST.

GraphQL is already widely used in Microservices architectures as an alternative API endpoint to REST. There are several Java-based GraphQL libraries available, but none with the reach of the MicroProfile community.



SANDBOX PROJECT “CONCURRENCY”

The proposal [Concurrency](#) introduces APIs - fully compatible with the EE Concurrency specification - for obtaining `CompletableFutures` that are backed by managed threads, with the ability to capture context from the thread that creates the `CompletableFuture` and apply it when running the `CompletionStage` action.



SERVICE MESH

Beneath the above described sandbox project there is an ongoing discussion if and how MicroProfile could support [Service Mesh](#) solutions like [Istio](#) or [Linkerd](#).

Cloud Native microservices developed with MicroProfile can take advantage of a Service Mesh by extracting many concerns away from the development of the microservice itself and delegate it to the Service Mesh which is a dedicated infrastructure layer for making service-to-service communication safe, fast and reliable.



Summary

Microservices can significantly shorten the time to market. In addition, they can help to build an application that is more stable, scalable and fault tolerant.

These benefits do not come for free. Splitting up an application into tens or hundreds of microservices will result in a more complex operation scenario due to the absence of a central runtime. New challenges arise for which typical Enterprise Computing Frameworks, like Java EE, are not appropriate.

MicroProfile defines a programming model for developing cloud-native microservices-based applications and enables Java EE developers to leverage their existing skill set while shifting their focus from traditional 3-tier applications to microservices.

Check out the MicroProfile APIs, listed below:

- [MicroProfile Config](#)
- [MicroProfile Fault Tolerance](#)
- [MicroProfile Health Check](#)
- [MicroProfile Metrics](#)
- [MicroProfile Open API](#)
- [MicroProfile Rest Client](#)
- [MicroProfile JWT Authentication](#)
- [MicroProfile Open Tracing API](#)

“ IF YOU ASK YOURSELF, WHEN IS THE BEST TIME TO START DEVELOPING YOUR MICROSERVICES-BASED APPLICATIONS USING MICROPROFILE ” THE ANSWER IS NOW. BECOME PART OF AN OUTSTANDING COMMUNITY & GIVE MICROPROFILE A TRY.

BEST TIME TO START IS NOW!

MicroProfile enables developers to substantially increase the productivity and the quality of microservices-based applications. This is why MicroProfile was chosen as the winner of one of the most important community awards in 2018, the Duke's Choice Award.

To ensure that getting started with MicroProfile is as easy as possible, the MicroProfile community has developed very helpful content, like tutorials, example applications, blog posts or presentations.

BLOG

Newbies to MicroProfile will find “getting started” tutorials for nearly every aspect of the MicroProfile usage inside the [MicroProfile Blog](#).

FAQ

In addition, answers frequently asked questions while starting to work with the MicroProfile APIs can be found at the [MicroProfile FAQs](#).

PROJECTS & DOCUMENTATION

To gain a deeper understanding of the different MicroProfile projects and how to use them correctly and effectively, developers can reference the [MicroProfile Projects & Documentation](#)



Wonderful MicroProfile White Paper Contributors

GENERAL REVIEW:

[Emily Jiang](#), [Heiko Rupp](#), [Jean-Louis Monteiro](#), [Cesar Saavedra](#), [Kevin Sutter](#),
[Mark Struberg](#), [Ken Finnigan](#), [Clement Escoffier](#), [David Blevins](#), [John Clingan](#),
[Antoine Sabot-Durand](#), [Scott M. Stark](#), [Ondro Mihalyi](#).

CONTRIBUTED SECTIONS:

>> Lessons learned from the Real World Microservice Adoption by [Matjaz B. Juric](#)

>> Partner Network by [Markus Eisele](#)

STEWARDS:

[Richard Monson-Haefel](#) (Continuity of Content), [Peggy Blevins](#) (Copy/Edit),
[Ryan St. James](#) (Design) & [Amelia Eiras](#) (360 Steward)

OSS PROCESS:

<https://github.com/eclipse/microprofile-marketing/issues/3>



ABOUT LARS RÖWEKAMP

Lars Röwekamp is the founder and CIO New Technologies of [open knowledge GmbH](#).

He has been advising customers on international projects for more than twenty years around the theme of enterprise and mobile computing. In his daily job Lars Röwekamp is responsible for the evaluation and rating of new technologies in context of marketability, sustainability and future viability.

You can find him on Twitter at [@mobileLarson](#).

