MICROPROFILE™

Eclipse MicroProfile

whitepaper

ECLIPSE MICROPROFILE
WHITEPAPER 2019

Date of Issue :
22 / 05 / 2019

A Technical Focus

This whitepaper 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 appropriate. 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 there is lack of a centralized runtime in the design of any distributed system.

Lets do this!

"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."

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. MicroProfiles 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 eight 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 excellent described by Mike Croft in his blog post "The MicroProfile turns one".

Why Microservices?

The difficulties associated with developing, testing, fixing, and updating applications 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.

FIGURE DEMONSTRATING THE BENEFIT OF A DECOMPOSED SYSTEM A.K.A. MICROSERVICES

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 helpsmaintain 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). Image 1 illustrates the choreography of the checkout workflow.

ORCHESTRATION

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 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 2 illustrates the orchestration of the checkout workflow.

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 a 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!

Writing a single Microservice is simple and can easily be done with the help of the Java EE Standard, because Java EE has the APIs we need:

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 Java Enterprise Standard 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 the feedback of the end users. 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 solutionblocks - a.k.a. APIs - each of them addressing a specific topic.

IMAGE SHOWING THE EVOLUTION OF THE MICROPROFILE ITSELF & ITS APIS

1.3 MICROPROFILE CONFIGURATION

MicroProfile Config (also known as ConfigJSR), **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 it deployment, microservices ought to be possible without the need to repackage the whole application binary.

ConfigJSR 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;
```

Listing: Examples of MicroProfile Configuration usage.

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 include 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.

```
@GET
@Timeout(500)
@Fallback(fallbackMethod = "getAllProductsFallback")
public Response getAllProducts() throws InterruptedException {
    // retrieve products, e.g. via database access
    …
}

public Response getAllProductsFallback() {
    // retrieve products from alternative source, e.g. cached values
    …
}
```

Listing: Example of MicroProfile Fault Tolerance usage.

[QUESTION: insert image to demonstrate the "lifecycle" of a resilient microservice call including fallback?]

1.0 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 MicroProfile Health Check has been designed with Kubernetes, Cloud Foundry and Mesos in mind, so leveraging the provided orchestrator features is natural.

Listing: Example of MicroProfile Fault Tolerance usage.

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. MicroProfile Health Check discovery 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:

```
@Health
@ApplicationScoped
public class CustomerServiceConnectionPoolCheck 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() { … }

}
```

1.1 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:
an incrementally increasing or decreasing numeric value (e.g. total number of requests received or the total number of concurrently active HTTP sessions).

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

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).

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 OPENAPI

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.

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).

» Take the initial output from OpenAPI as a starting point to document APIs via Static OpenAPI files. It's worth mentioning that these static files can also be written before any code, which is an approach often adopted by enterprises that want to lock-in the contract of the API. In this case, we refer to the OpenAPI document as the „source of truth", by which the client and provider must abide.

» 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

[QUESTION: insert image equivalent to the architectural overview of the specification to demonstrate the flexibility of this mechanism?]

1.2 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.

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 Microprofile 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.

FOR RESTFUL BASED MICROSERVICES, SECURITY TOKENS OFFER A VERY LIGHTWEIGHT AND INTEROPERABLE WAY TO PROPAGATE IDENTITIES ACROSS DIFFERENT SERVICE WHERE A TOKEN FORMAT IS WELL KNOWN OR SERVICES CAN INVOKE A SEPARATE SERVICES SO THAT:

» Services don't need to store any state about clients or users.

» Services can verify the token validity.

» Services can identify the caller by introspecting the token.

» Services can enforce authorization policies based on any information within a security token

» Services need support both delegation and impersonation of identities.

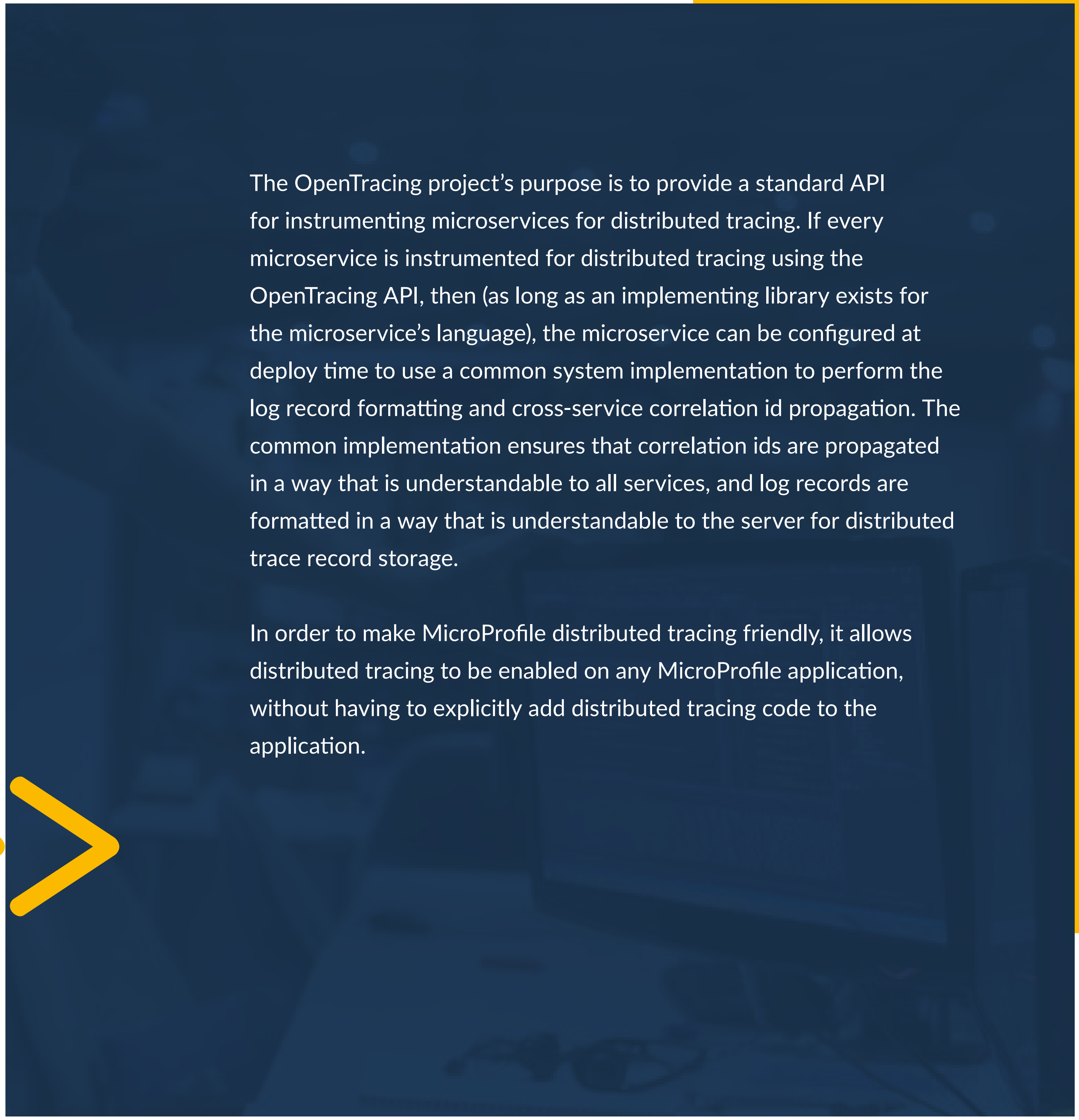
Today, the most common solutions involving RESTful and microservices security are based on OAuth2, OpenID Connect (OIDC) and JSON Web Tokens (JWT) standards.

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.

Lessons learned from real-world microservice adoption

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 Docker 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 application servers 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 gradually transform 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 works hand-in-hand with application servers. Another practice is to migrate certain functionalities to microservices. Because Microprofile microservices are based on standard Java EE, it is relatively easy to migrate existing code to microservices. Best practices for migration are well known and provide easy migration of REST and SOAP web services, CDI, JDBC, JPA, JMS and other common technologies. Only EJBs may require some code refactoring with best practice to migrate them to CDI.

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, resilien
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

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.

In January 2017, MicroProfile became an official Eclipse project. [or something similar, you tell me how can i help you here]-

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

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 Devoxx 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 Enterprise 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

Kumuluz

KumuluzEE

RedHat

Thorntail (née WildFly Swarm)

Red Hat OpenShift Application Runtimes

Innovation & Sandboxing

The current version 2.1 of 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 three 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 operator 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".

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 Linkered.

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 results 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.

MicroProfiles APIs builds an optimal bases for developing microservices-based applications by adopting a subset of the Java EE Standards and extending them with new ones that address common microservices patterns, including:

- 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 MICROPROILE”
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 off 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 & SAMPLES

To gain a deeper understanding of the different MicroProfile projects and how to use them correctly and effectively, developers can reference the MicroProfile Projects and the corresponding MicroProfiles Samples.