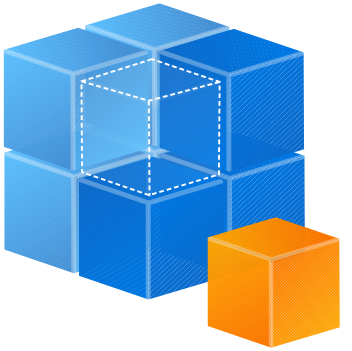
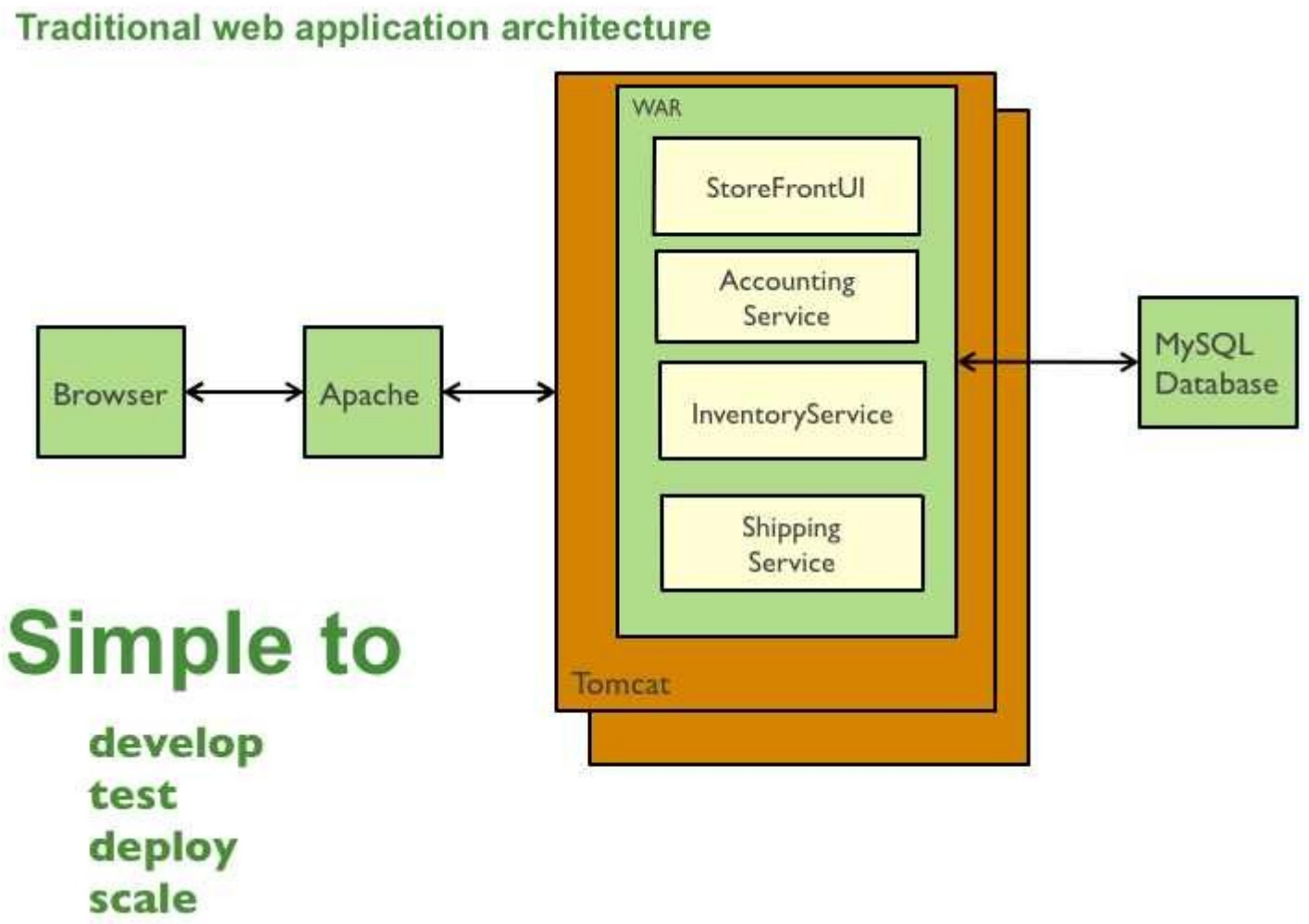
Java Technologies
Microservices

The Context



- **Monolith**
  - *big* application requiring a *big* server
  - contains all the required components
  - deployed as a single package (for example, a .war)
  - changing a component → redeployment
  - uses (mostly) the same programming language and the same technological stack
  - scalability is obtained using server clustering
- How to create **a more flexible architecture**?

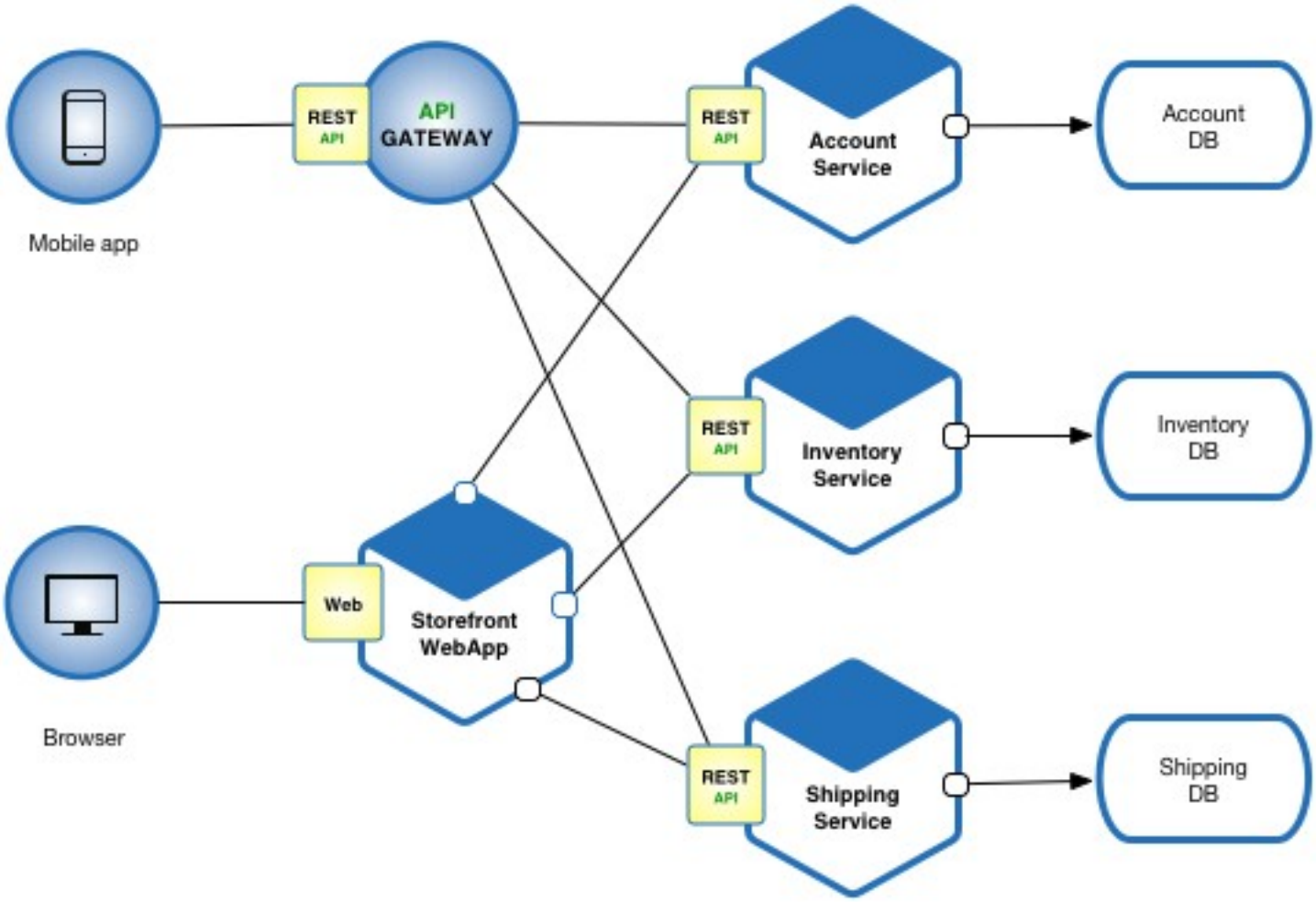
Traditional Web App



Microservices

- An **architectural style** that aims at breaking the monolith in multiple parts (services)
- Each service must be:
  - **small**, **isolated**, **independent**ly managed
  - **cohesive**, focusing on a single responsibility
  - responsible for **its own data**
  - exposing a **standard interface**
  - easy to start (not requiring a *big* server)
- Trade-off: more flexibility and *auto*-scalability vs. communication overhead and higher complexity

Microservices Based App



The Five Rules

- Deploy applications as sets of small, independent services
- Optimize services for a single function (Single Responsability Pinciple)
- Communicate via REST API and message brokers
- Apply Per-service CI/CD
- Apply Per-service HA/clustering decisions

Brown, Kyle, and Bobby Woolf. "Implementation patterns for microservices architectures."

Eclipse MicroProfile

- Optimizing Enterprise Java
  for a Microservices Architecture
- A collection of Java EE APIs and technologies which together form a core baseline microservice that aims to deliver application portability across multiple runtimes.
- Based on a subset of Java EE WebProfile: REST, JSON, JAX-RS, CDI
- Specifications first
  - Avoiding vendor lock-in

MicroProfile APIs

MicroProfile Runtimes

https://wiki.eclipse.org/MicroProfile/Implementation

- PayaraMicro

  Payara Micro is the lightweight middleware platform of choice for containerized Jakarta EE application deployments. Less than 70MB, Payara Micro requires no installation, configuration, or code rewrites – so you can build and deploy a fully working app within minutes.
- OpenLiberty

  Build cloud-native apps and microservices while running only what you need. Open Liberty is the most flexible server runtime available to Java developers in this solar system.
- WildFly

  WildFly is a powerful, modular, & lightweight application server that helps you build amazing applications.
- Quarkus

  Supersonic Subatomic Java. A Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards.
- KumuluzEE

  Develop microservices with Java EE / Jakarta EE technologies and extend them with Node.js, Go and other languages. Migrate existing Java EE applications to microservices and cloud-native architecture.
- Red Hat Thorntail, TomEE, etc.
- *Non-standard alternatives:* Spring Boot, Micronaut, Dropwizard, etc.

Hello World

https://dzone.com/articles/microservices-for-java-ee-developers

- **Download a MicroProfile Runtime** (for example, PayaraMicro → payara-micro-5.jar)
- **Create a Maven Web Project** – Do not select a EE server.

  As an alternative, you can use the support offered by IDE – Create Payara Micro Application,

  or use https://start.microprofile.io/
- Configure the **pom.xml**
- Test your application (there is a predefined index.html)

```
java -jar payara-micro-5.jar
   --deploy HelloWorld.war
   --autoBindHttp
   --name HelloService
   --contextRoot /demo
```

pom.xml

- The bear necesity

```
<dependencies>
    <dependency>
        <groupId>org.eclipse.microprofile</groupId>
        <artifactId>microprofile</artifactId>
        <version>1.2</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

- Addition configurations, available via payara-micro-maven-plugin

Create an Endpoint

- The Application: *DemoRestApplication*

```
@ApplicationPath("/service")
@ApplicationScoped
public class DemoRestApplication extends Application {
  //Nada
}}
```

- The Service: *HelloControler*

```
@Path("/hello")
@Singleton
public class HelloController {
    @GET
    public String sayHello() {
        return "Hello World";
    }
}
```

```
http://localhost:8080/demo/service/hello
```

Using OpenLiberty

https://openliberty.io/guides/getting-started.html

- Create a project https://start.microprofile.io/
- Make sure you have maven installed
- Go to the project directory
- **`mvn liberty:run`**
  - starts an Open Liberty server instance in the foreground; an Open Liberty server runtime is downloaded and installed into the target/liberty/wlp directory, a server instance is created and configured in the target/liberty/wlp/usr/servers/defaultServer directory, and the application is installed into that server via loose config.
- **`http://localhost:9080/`**
- **`mvn liberty:dev`**
  - Updating the server configuration without restarting the server

Microservices Challenges

- Infrastructure (chassis) for creating / running
- Decomposition of the monolith
- Communication and Discovery
- Observability and Reliability
- Data Management
- Security
- Testing and Documenting
- Deployment
- Scalability

Decomposition of the Monolith

- **Stable** architecture
- **Cohesive** services A service should implement a small set of strongly related functions.
- **Common Closure Principle** Things that change together should be packaged together
- **Loosely coupled** Each service encapsulates its implementation
- **Testable**
- **Small enough** services (“two pizza” team, 6-10 people)
- **Autonomous teams** Minimal collaboration with other teams
- Strategies
  - By **business** capability (fine grained)
  - By **subdomain** (coarse grained)

Microprofile Architecture (1)

- **MicroProfile Rest Client, JAX-RS**

  A type-safe approach for invoking RESTful services over HTTP. The MicroProfile Rest Client builds upon the JAX-RS APIs for consistency and ease-of-use.

- **Context and Dependency Injection (CDI)**

  The base for a growing number of APIs included in MicroProfile

- **Common Annotations**

  Annotations for common semantic concepts across a variety of individual technologies

- **JSON-B**

  Standard APIs for binding JSON documents to Java code

- **JSON-P**

  Standard APIs for processing JSON documents

Microprofile Architecture (2)

- **MicroProfile Config**

  Obtain configuration properties through several environment-aware sources both internal and external to the application and made available through dependency injection or lookup.

- **MicroProfile Fault Tolerance**

  Separate execution logic from business logic. Key aspects of the API include TimeOut, RetryPolicy, Fallback, Bulkhead, and Circuit Breaker processing.

- **MicroProfile Health**

  Probe the state of a computing node from another machine (e.g. kubernetes service controller).

- **MicroProfile JWT Authentication**

  Provides role based access control (RBAC) microservice endpoints using OpenID Connect (OIDC) and JSON Web Tokens (JWT)

Microprofile Architecture (3)

- **MicroProfile Metrics**

  A unified way for MicroProfile servers to export monitoring data to management agents. Metrics will also provide a common Java API for exposing their telemetry data.

- **MicroProfile OpenAPI**

  A unified Java API for the OpenAPI specification (OAS) that all application developers can use to expose their API documentation.

- **MicroProfile OpenTracing**
  - An API and associated behaviors that allow services to easily participate in a distributed tracing environment

- *Extensions for Microprofile (community)*

Configuration

- The majority of applications need to be configured based on a running environment. It must be possible to modify configuration data from outside an application so that the application itself does not need to be repackaged.
- *resources/META-INF/microprofile-config.properties*

```
message = Hello World;
```

- In the definition of a service

```
@Inject
@ConfigProperty(name = "message")
private String value;
```

- Or:

```
Config config = ConfigProvider.getConfig();
String value = config.getValue("message", String.class);
```

- Environment variables, System properties, Server external configuration files (server.xml), *ConfigSources*

Communication

- **Synchronous**: request / response
  - ➔ Programatic, HTTP oriented: *ClientBuilder* (JAX-RS)
  - ➔ Declarative (CDI), RMI style: *RestClient* (MP)
- **Asynchronous** (using messages)
  - ✔ Request / response
  - ✔ Notifications
  - ✔ Request / asynchronous response
  - ✔ Publish / subscribe
  - ✔ Publish / asynchronous response
  - ➔ *Kafka, RabbitMQ, ActiveMQ, Debezium, OpenMQ, etc.*
- **Domain specific**: email protocols such as SMTP and IMAP, media streaming protocols such as RTMP , HLS and HDS.

Real Time Media Protocol, HTTP Live/Dynamic Streaming

JAX-RS *ClientBuilder*

Distributed object communication

Programatic, HTTP

```
String uriOne = "http://localhost:8080/resources/products/1";
Product product =
   ClientBuilder.newClient()
    .target(uriOne)
    .request(MediaType.APPLICATION_JSON)
    .get(Product.class);
```

```
String uriAll = "http://localhost:8080/resources/products";
List<Product> list =
   ClientBuilder.newClient()
    .target(uriAll)
    .request(MediaType.APPLICATION_JSON)
    .get(new GenericType<ArrayList<Product>>() {});
```

MicroProfile *RestClient*

Declarative, Type-safe, CDI

```
@RegisterRestClient(baseUri="http://localhost:8081/resources")
@ApplicationScoped
public interface Service {
    @GET
    @Path("/products")
    List<Product> getProducts();
}
```

May be defined in microprofile-config.properties

```
@Path("/client")
@ApplicationScoped
public class ClientController {

    @Inject
    @RestClient
    private Service service;

    @GET
    @Path("/test")
    public List<Product> onClientSide() {
        return service.getProducts();
    }
}
```

MicroProfile *RestClientBuilder*

- Get an instance of the client and also allow you to add configurations dynamically.
- Example

```
    @GET
    @Path("/test")
    public List<Product> onClientSide() {
      Service service = RestClientBuilder.newBuilder()
        .build(Service.class);
      return service.getProducts();
    }
}
```

Synchronous and Asynchronous REST Clients

- **Synchronous client:** constructs an HTTP structure, sends a request, and waits for a response.
  - Preferred in situations where service availability is high and low latency is a priority.
- **Asynchronous client:** constructs an HTTP structure, sends a request, and moves on.
  - Valuable in cases where service availability is low or overloaded with demand

Asynchronous *RestClient*

```
@RegisterRestClient
@ApplicationScoped
public interface Service {
    @GET
    String sayHello();

    @GET
    CompletionStage<String> sayHelloAsync();
}
```

```
public class ClientController {
    ...
    @GET
    @Path("/test")
    public String invokeSync() {
        var res = service.sayHelloAsync();
        //do stuff while invoking the remote service
        try {
            return res.toCompletableFuture().get();
        } catch (InterruptedException | ExecutionException ex) {
            return ex.toString();
        }
    }
}
```

java.util.concurrent.Future

- A *Future* represents the result of an asynchronous computation.
- Methods are provided to:
  - check if the computation is complete,
  - wait for its completion
  - retrieve the result of the computation.
- The result can only be retrieved using method *get* when the computation has completed, blocking if necessary until it is ready.
- Cancellation is performed by the *cancel* method. Additional methods are provided to determine if the task completed normally or was cancelled.

CompletionStage

- A stage of a possibly **asynchronous computation**, that performs an action or computes a value when another *CompletionStage* completes.
- A stage completes upon termination of its computation, but this may in turn trigger other dependent stages.
- Example:

```
stage.thenApply(x -> square(x))
     .thenAccept(x -> System.out.print(x))
     .thenRun(() -> System.out.println());
```

CompletableFuture

Monadic pattern

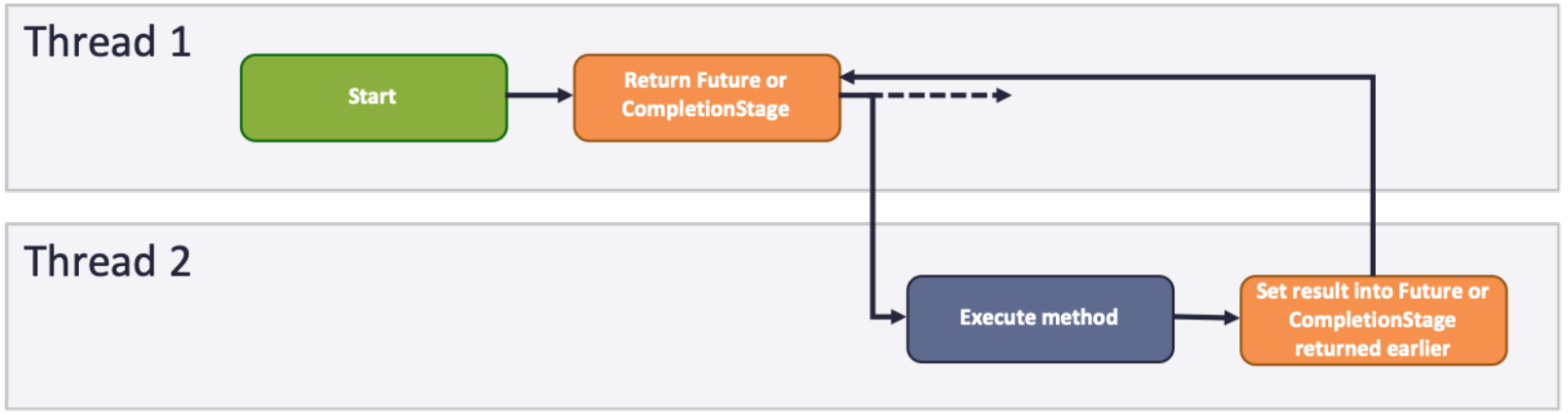
- Implements *Future, CompletionStage*
- A *Future* that may be explicitly completed (setting its value and status), and may be used as a *CompletionStage*, supporting dependent functions and actions that trigger upon its completion.

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future = completableFuture
  .thenApply(s -> s + " World");

assertEquals("Hello World", future.get());
```

Sever-side *Asynchronous*

- Runs the logic of a service in a new thread.
- Useful for *fault tolerance* techniques and *throttling* access to other resources (such as a database).

```
@Asynchronous
public CompletionStage<String> serviceMethod() {
  return CompletableFuture.completedFuture("Hello");
}
```



Sever-side *Asynchronous*

- When a method marked with this annotation is called from one thread (which we will call Thread A), the method call is intercepted, and execution of the method is submitted to run asynchronously on another thread (which we will call Thread B).
- On Thread A, a *Future* or *CompletionStage* is returned immediately and can be used to get the result of the execution taking place on Thread B, once it is complete.
- Before the execution on Thread B completes, the *Future* or *CompletionStage* returned in Thread A will report itself as incomplete. At this point, Future.cancel(boolean) can be used to abort the execution.
- Once the execution on Thread B is complete, the *Future* or *CompletionStage* returned in Thread A will hold the response from Thread B, or the exception (if any).

Resilience and High Availability

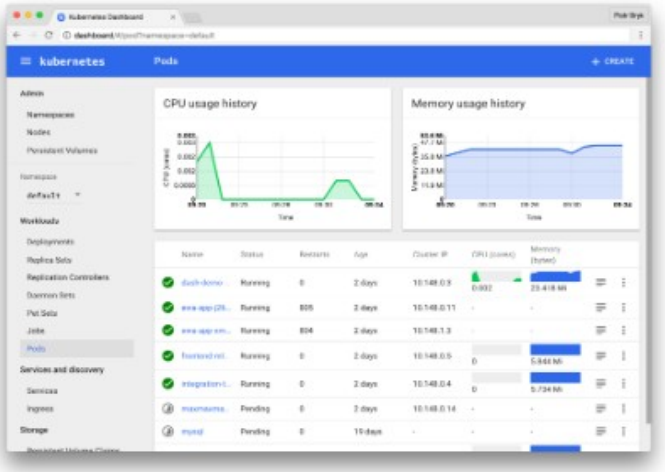
- We are in the context of a dstributed system with a large number of communicating nodes (machines)
- How to deal with unexpected failures?
- A microservice must report its **health** and **diagnostics**.
- A microservice needs to be able to **restart** often on another machine for availability.
- Resilience can be:
  - in the **compute** capability (the process can restart at any time)
  - in the **state or data** (no data loss, and the data remains consistent).
- Client apps or client services must have a strategy to retry sending messages or to retry requests, since in many cases failures in the cloud are partial.

Metrics

Metric = a system or standard of measurement

- This specification aims at providing a unified way for Microprofile servers to export Monitoring data ("Telemetry") to management agents and also a unified Java API, that all (application) programmers can use to expose telemetry data.
- Examples:

```
public class MetricController {
    @Inject
    @Metric(name = "endpoint_counter")
     private Counter counter;

     @Path("timed")
     @Timed(name = "timed-request")
     @GET
     public String timedRequest() { … }

    @Gauge(name = "counter_gauge", unit = MetricUnits.NONE)
    private long getCustomerCount() {
        return counter.getCount();
    }
}
```



http://localhost:8080/metrics
/base, /vendor, /application

MicroProfile Metrics API

- *@Metric*: A meter of the field's type will be created and injected into managed objects. It will be up to the user to interact with the metric. This annotation can be used on fields of type (interfaces):
  - *Meter, Timer, SimpleTimer, Counter, and Histogram*.
- *@Counted*: An annotation for marking a method, constructor, or class as counted. It counts the total invocations of the annotation method (monotonic=true), or the amount of parallel invoked methods at any time: (monotonic=false).
- *@Gauge* exposes the return value of the annotated method as a metric. It is used to expose metrics with dedicated methods. The values have to be provided by the developer (e.g. number of orders in the DB) and the method is going to be invoked by the metrics infrastructure.
- *@ConcurrentGauge:* same as *@Counted(monotonic=false)*

Fault Tolerance API

Fault tolerance is about leveraging different strategies to guide the execution and result of some logic:

- **Timeout**: Define a maximum duration for execution
- **Retry**: Attempt execution again if it fails
- **Bulkhead**: Limit concurrent execution so that failures in that area can’t overload the whole system
- **CircuitBreaker**: Automatically fail fast when execution repeatedly fails
- **Fallback**: Provide an alternative solution when execution fails
- **Asynchronous**: invoke the operation asynchronously.

The Timeout Pattern

- Assuming a service was reached and a request was made, what happens if the service doesn't respond?
  - wait indefinitely or
  - leverage some sort of timeout value.
- What happened on the service side?
  - It never got the request.
  - It got the request, processed it, but the response didn't get to the client.
  - The service crashed.
- What next?
  - Ignore it :) or use a default (cached) value or
  - Retry the invocation, if it's safe(!), i.e.idempotent service

Fallback Timeout

```
@Path("/resilience")
@ApplicationScoped
public class FaultToleranceController {

    @Timeout(500)
    @Fallback(fallbackMethod = "fallback")
    @GET
    public String doWork() {
      try { Thread.sleep(700L); }
      catch (InterruptedException e) { }
      return "Response from normal processing";
    }

    private String fallback() {
       return "Fallback answer due to timeout";
    }
}
```

applyOn
The list of exception types which should trigger Fallback

skipOn
The list of exception types which should not trigger Fallback

The Retry Pattern

- Services can fail due to transient faults, such as:
  - slow network connections,
  - timeouts, or
  - the resources being overcommitted or temporarily unavailable.
- These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them.
- For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased.

Fallback Retry

```
@Path("/resilience")
@ApplicationScoped
public class FaultToleranceController {

    @Retry(maxRetries = 2, delay = 200, jitter = 50)
    @Fallback(fallbackMethod = "fallback")
    public Result doWork() {
        return callAnotherService();
        // This service usually works...
    }

    private Result fallback() {
        return Result.emptyResult();
    }
}

//using config:
mypackage.FaultToleranceController/doWork/Retry/maxRetries=6
```

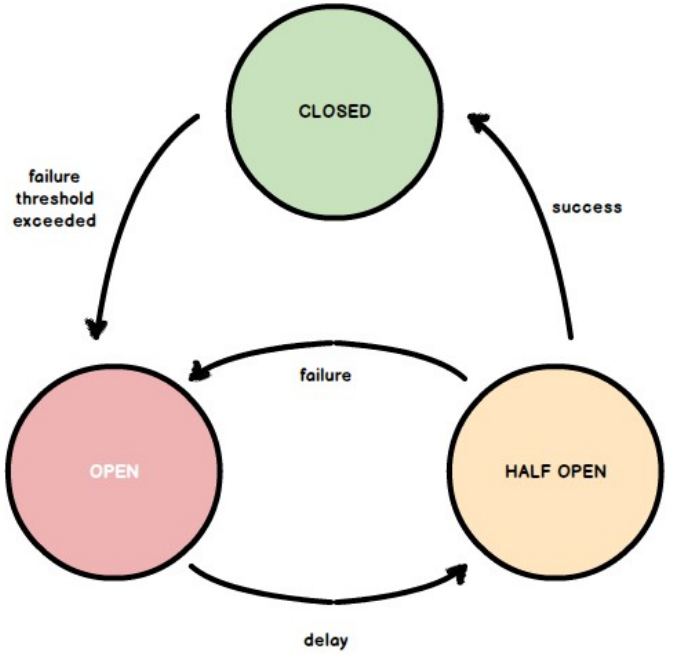
If the method returns normally (doesn't throw):
→ the result is simply returned.
If the thrown object is assignable to any value in the *abortOn*() parameter:
→ the thrown object is rethrown.
If the thrown object is assignable to any value in the *retryOn*() parameter:
→ the method call is retried.
Otherwise the thrown object is rethrown.

The Circuit Breaker Pattern

- A **circuit breaker** aims to prevent further damage by not executing functionality that is doomed to fail. After a failure situation has been detected, circuit breakers prevent methods from being executed and instead throw exceptions immediately. After a certain delay or wait time, the functionality is attempted to be executed again.
- A circuit breaker can be in one of the following states:
  - **Closed**: In normal operation, the circuit is closed. If a failure occurs → the circuit will be opened.
  - **Open**: When the circuit is open, calls to the service operating under the circuit breaker will fail immediately. After a specified delay, the circuit transitions to half-open state.
  - **Half-open**: In half-open state, trial executions of the service are allowed. If these calls fail, the circuit will return to open state, otherwise the circuit will be closed.

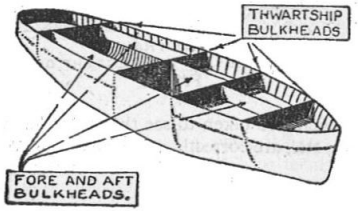
CircuitBreaker Example

```
@CircuitBreaker(
    successThreshold = 10,
    requestVolumeThreshold = 4,
    failureRatio=0.75,
    delay = 1000)
public Connection getConnection() {
      //...
       return con;
}
```



The above code-snippet means the method *getConnection* applies the CircuitBreaker policy, which is to open the circuit once 3 (4x0.75) failures occur among the rolling window of 4 consecutive invocations. The circuit will stay open for 1000ms and then back to half open. After 10 consecutive successful invocations, the circuit will be back to close again. When a circuit is open, a CircuitBreakerOpenException will be thrown.

The Bulkhead Pattern



- The **Bulkhead** pattern is to prevent faults in one part of the system from cascading to the entire system, which might bring down the whole system.
- Context: Excessive load or failure in a service will impact all consumers of the service. A consumer may send requests to a faulty service, using resources for each request. Other consumers are no longer able to consume the service, causing a cascading failure effect.
- To prevent this, elements of an application should be **isolated** so that if one fails, the others will continue to function.
- The implementation is **to limit the number of concurrent requests** accessing to an instance.

Bulkead Implementations

There are two different approaches to the bulkhead:

- **Semaphore** isolation: execution happens on the calling thread and the concurrent requests are constrained by the semaphore count.

```
@Bulkhead(5) // maximum 5 concurrent requests allowed
public Connection service() {...}
```

- **Thread pool** isolation: execution happens on a separate thread and the concurrent requests are confined in a fixed number of a thread pool.

```
@Asynchronous
@Bulkhead(value = 5, waitingTaskQueue = 8)
// maximum 5 concurrent requests allowed,
// maximum 8 requests allowed in the waiting queue
public Future<Connection> service() {
  return CompletableFuture.completedFuture(...);
}
```

Health Check

- Used to probe the state of a computing node from another machine (i.e. kubernetes service controller) with the primary target being cloud infrastructure environments where automated processes maintain the state of computing nodes.
- Health checks are used to determine if a computing node needs to be discarded (terminated, shutdown) and eventually replaced by another (healthy) instance. It is not intended (although could be used) as a monitoring solution for human operators.
- **Readiness**: allows third party services to know if the application is ready to process requests or not.
- **Liveness**: allows third party services to determine if the application is running. This means that if this procedure fails the application can be discarded (terminated, shutdown).

Readiness and Liveness

```
@Readiness //or @Liveness or both
@ApplicationScoped
public class ServiceReadyHealthCheck implements HealthCheck {
    @Override
    public HealthCheckResponse call() {
       return HealthCheckResponse
         .named(ServiceReadyHealthCheck.class.getSimpleName())
         .withData("ready",true)
         .up() //or .status(getMemUsage() < 0.9)
         .build();
    }
}
```

HealthCheck using CDI

```
@ApplicationScoped
class MyChecks {

  @Produces
  @Liveness
  HealthCheck check1() {
    return () -> HealthCheckResponse.named("heap-memory")
      .status(getMemUsage() < 0.9).build();
  }

  @Produces
  @Readiness
  HealthCheck check2() {
    return () -> HealthCheckResponse.named("cpu-usage")
      .status(getCpuUsage() < 0.9).build();
  }
}
```

HealtCheck is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

JSON Processing (JSON-P)

- **Parse, generate, transform and query** JSON messages.
- Produces and consumes JSON text in a streaming fashion (similar to StAX API for XML) and allows to build a Java object model for JSON text using API classes (similar to DOM API for XML).
- Example:

```
JsonObject json = Json.createObjectBuilder()
   .add("name", "Falco")
   .add("age", BigDecimal.valueOf(3))
   .add("biteable", Boolean.FALSE).build();
String result = json.toString();
```

JSON Binding (JSON-B)

- Standard binding layer for **converting Java objects to/from JSON messages**. It defines a default mapping algorithm for converting existing Java classes to JSON, while enabling developers to customize the mapping process through the use of Java annotations.
- Example:

```
Dog dog = new Dog();
dog.name = "Falco";
dog.age = 4;
dog.bitable = false;

// Serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dog);

// Deserialize
dog = jsonb.fromJson("…", Dog.class);
```

Testing

- Bussines logic (usual functional testing)
- Service Component
  - testing a service in isolation is easier, faster, more reliable and cheap
  - use test doubles for any services that it invokes.
- Consumer-side
- Service Integration
  - written by the developers of another service that consumes it
- *Frameworks*
  - Junit, Microshed, Arquillian, TestNG, JBehave, Mockito, Selenium, Serenity, etc.

Testing with Microshed

https://microshed.org/microshed-testing/

- Writing and running true-to-production integration tests for Java microservice applications.
- Exercises your containerized application from outside the container so you are testing the exact same image that runs in production.
- If the repository containing the tests does not have a Dockerfile in it:
  - it is possible to use vendor-specific adapters (for example, the microshed-testing-liberty adapter) that will provide the default logic for building an application container.
  - a docker image label can be supplied instead.

Testing Example

```
@MicroShedTest
public class HelloWorldIT {

   // This will search for a Dockerfile in the repository and start up the application
   // in a Docker container, and wait for it to be ready before starting the tests.
   @Container
   public static MicroProfileApplication app
           = new MicroProfileApplication()
                   .withAppContextRoot("/service")
                   .withReadinessPath("/health/ready");

   // This injects a REST _Client_ proxy of the HelloController
   // This allows us to easily invoke HTTP requests on the running application container
   @RestClient //or @Inject
   public static HelloController resource;

   @Test
   public void sayHello() throws IOException {

       // This invokes an HTTP GET request to the running container, which triggers
       // the HelloController#sayHello endpoint and returns the response
       String response = resource.sayHello();
       Assert.assertTrue(response.startsWith("Hello World"));
   }

}
```

Documenting – OpenAPI

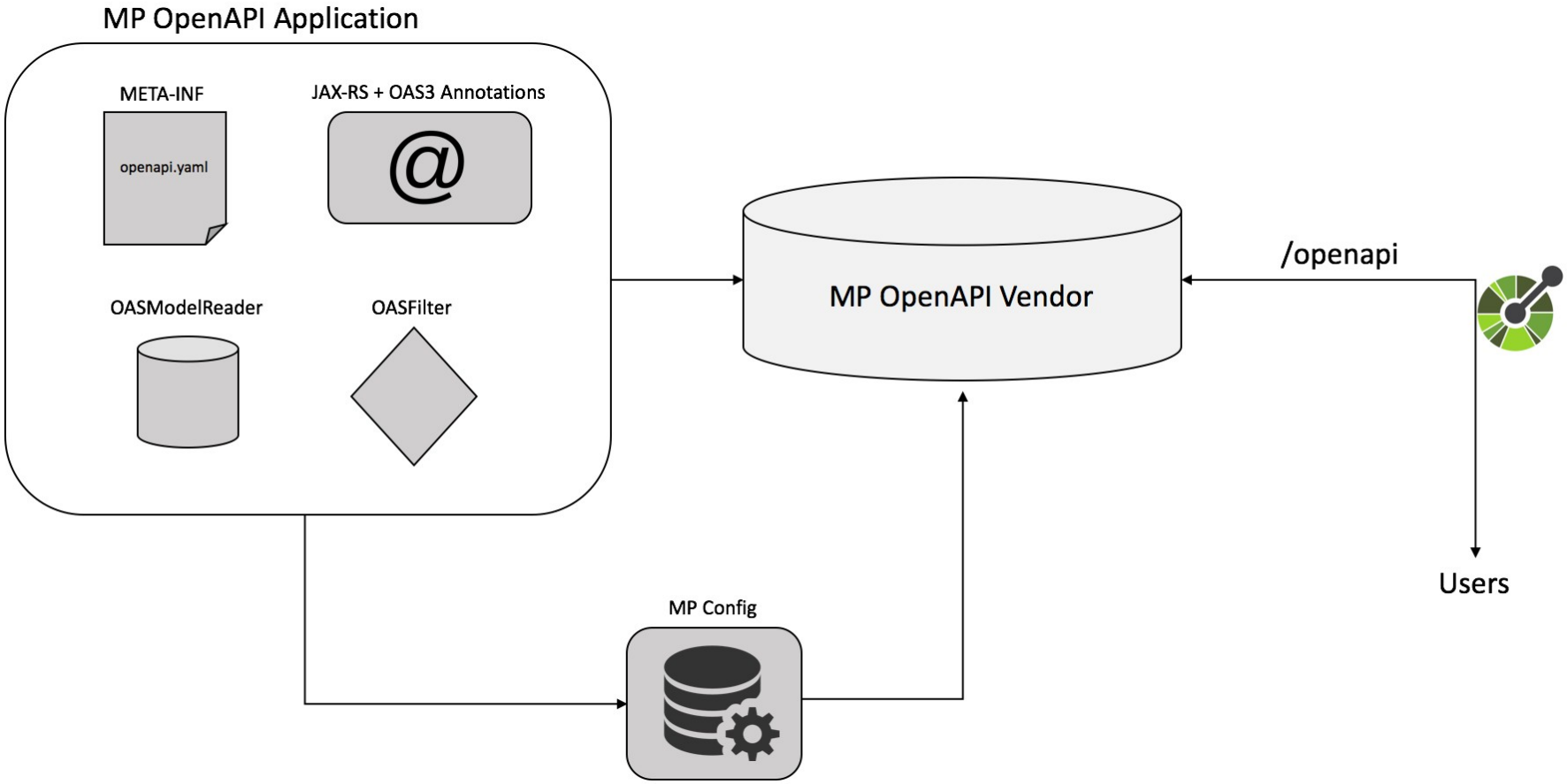
https://swagger.io/

- **OpenAPI**: description format for REST APIs.
  - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
  - Operation parameters input and output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information.
- **Swagger is** a set of open-source tools built around the OpenAPI Specification
  - Browser-based **editor** where you can write OpenAPI specs.
  - **UI** – renders OpenAPI specs as interactive API doc.
  - **Codegen** – generates server stubs and client libraries

MicroProfile OpenAPI

https://download.eclipse.org/microprofile/microprofile-open-api-1.1.2/microprofile-openapi-spec.html

Aims to provide a set of interfaces and programming models which allow developers to natively produce OpenAPI documents from their JAX-RS applications



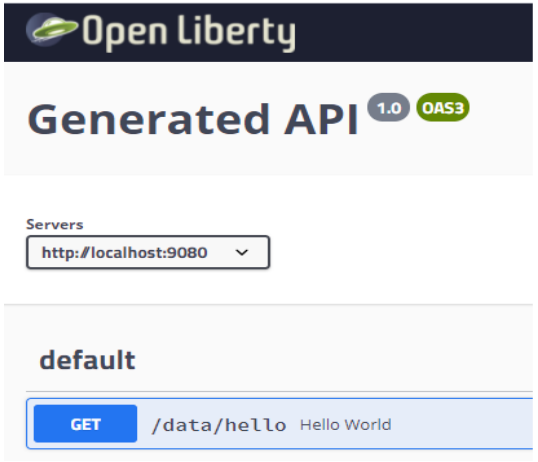
Example OpenAPI

https://openliberty.io/guides/microprofile-openapi.html

```
@Path("/hello")
@ApplicationScoped
public class HelloController {
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  @Operation(
      summary = "Hello World",
      description = "Returns the <b>Hello World</b> message ")
  public String sayHello() {
      return "Hello World!";
  }
}
```

http://localhost:9080/openapi/ →

http://localhost:9080/openapi/ui




```
openapi: 3.0.3
info:
  title: Generated API
  version: "1.0"
servers:
- url: http://localhost:9080
- url: https://localhost:9443
paths:
  /data/hello:
    get:
      summary: Hello World
      description: 'Returns the ...'
      responses:
        "200":
          description: OK
          content:
            text/plain:
              schema:
                type: string
```

Deployment

- Services are written using a variety of languages, frameworks, and framework versions.
- Each service consists of multiple service instances for throughput and availability.
- Instances must be independently deployable and scalable.
- Instances need to be isolated from one another.
- You need to be able to quickly build and deploy a service.
- You need to be able to constrain the resources (CPU and memory) consumed by a service.
- You need to monitor the behavior of each service instance.
- You want deployment to be reliable.
- You must deploy the application as cost-effectively as possible.

Deployment

- **Single** service instance **per host**
  - Host = Physical machine
- **Multiple** service instances **per host**
- Service **instance per VM**
- Service **instance per container**
- Service **deployment platform**
- **Serverless** deployment

Single Instance per Host

- Deploy each single service instance on its own host
- The benefits of this approach include:
  - Services instances are isolated from one another
  - There is no possibility of conflicting resource requirements or dependency versions
  - A service instance can only consume at most the resources of a single host
  - Its straightforward to monitor, manage, and redeploy each service instance
- The drawbacks of this approach include:
  - Potentially less efficient resource utilization

Multiple Instances per Host

- Run multiple instances of different services on a host.
  - Deploy each service instance as a JVM process. For example, a Tomcat or Jetty instances per instance.
  - Deploy multiple service instances in the same JVM. For example, as web applications or OSGI bundles.
- The drawbacks of this approach include:
  - Risk of conflicting resource requirements
  - Risk of conflicting dependency versions
  - Difficult to limit the resources consumed by an instance
  - If multiple services instances are deployed in the same process, it is difficult to monitor the resource consumption of each instance. Its also impossible to isolate each instance.

Service Instance per VM

- Package the service as a virtual machine image and deploy each service instance as a separate VM. For example, Netflix packages each service as an EC2 AMI and deploys each service instance as a EC2 instance.
- The benefits of this approach include:
  - Its straightforward to scale the service by increasing the number of instances. Amazon Autoscaling Groups can even do this automatically based on load.
  - The VM encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
  - Each service instance is isolated
  - A VM imposes limits on the CPU and memory consumed
  - IaaS solutions such as AWS provide a mature and feature rich infrastructure for deploying and managing virtual machines. For example, Elastic Load Balancer, Autoscaling Groups, etc.
- The drawback: building a VM image is slow and time consuming

Service Instance per Container

- Package the service as a (Docker) container image and deploy each service instance as a container.
- The benefits of this approach include:
  - It is straightforward to scale up and down a service by changing the number of container instances.
  - The container encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
  - Each service instance is isolated
  - A container imposes limits on the CPU and memory consumed
  - Containers are extremely fast to build and start. For example, it's 100x faster to package an application as a Docker container than it is to package it as an AMI. Docker containers also start much faster than a VM since only the application process starts rather than an entire OS.
- The drawback: the infrastructure for deploying containers is not as rich as the infrastructure for deploying virtual machines.

Service Deployment Platform

- Use a deployment platform, which is automated infrastructure for application deployment. It provides a service abstraction, which is a named, set of highly available (e.g. load balanced) service instances.
- Examples:
  - Docker orchestration frameworks including
    - Kubernetes
    - Docker swarm
  - Serverless platforms such as AWS Lambda
  - PaaS including Cloud Foundry and AWS Elastic Beanstalk

Serverless Deployment

AWS Lambda, Google Cloud Functions, Azure Functions

- Use a deployment infrastructure that hides any concept of servers (i.e. reserved or preallocated resources)- physical or virtual hosts, or containers. The infrastructure takes your service's code and runs it. You are charged for each request based on the resources consumed.
- To deploy your service using this approach, you package the code (e.g. as a ZIP file), upload it to the deployment infrastructure and describe the desired performance characteristics.
- The deployment infrastructure is a utility operated by a public cloud provider. It typically uses either containers or virtual machines to isolate the services. However, these details are hidden from you. Neither you nor anyone else in your organization is responsible for managing any low-level infrastructure such as operating systems, virtual machines, etc.

Serverless Deployment Benefits

- It eliminates the need to spend time on the undifferentiated heavy lifting of managing low-level infrastructure. Instead, you can focus on your code.
- The serverless deployment infrastructure is extremely elastic. It automatically scales your services to handle the load.
- You pay for each request rather than provisioning what might be under utilized virtual machines or containers.

Serverless Deployment Drawbacks

- Significant limitation and constraints - A serverless deployment environment typically has far more constraints that a VM-based or Container-based infrastructure. For example, AWS Lambda only supports a few languages. It is only suitable for deploying stateless applications that run in response to a request. You cannot deploy a long running stateful application such as a database or message broker.
- Limited “input sources” - lambdas can only respond to requests from a limited set of input sources. AWS Lambda is not intended to run services that, for example, subscribe to a message broker such as RabbitMQ.
- Applications must startup quickly - serverless deployment is not a good fit your service takes a long time to start
- Risk of high latency - the time it takes for the infrastructure to provision an instance of your function and for the function to initialize might result in significant latency. Moreover, a serverless deployment infrastructure can only react to increases in load. You cannot proactively pre-provision capacity. As a result, your application might initially exhibit high latency when there are sudden, massive spikes in load.

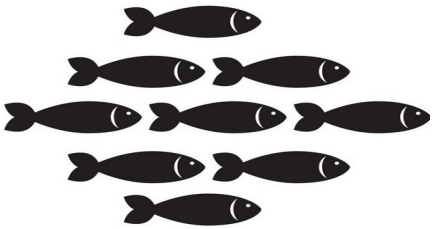
Auto(Dynamic) Scaling

- How to setup application infrastructure?
  - constant, predictable load → on-premise
  - varying loads, with high peaks → cloud
- Dynamic Scaling
  - **Scale Out** (create more instances as the load increases)
  - **Scale In** (reduces instances as the load goes down)
- **Naming servers** and **location transparency**.
  - Every microservice **registers** with a naming service
  - Microservice **discovery** uses the name services.

Instance Configuration

Whenever you start a new microservice.

```
...
{
    "Instance Configuration": {
        "Host": "Cristi-PC",
        "Http Port(s)": "8080",
        "Https Port(s)": "",
        "Instance Name": "myService1",
        "Instance Group": "MicroShoal",
        "Hazelcast Member UUID": "5147e01b-0c5f-4ea0-bf52-1d6cdf558d0a",
        "Deployed": [
            {
                "Name": "HelloWorld",
                "Type": "war",
                "Context Root": "/demo"
            }
        ]
    }
}
```



Clustering

- One service → multiple instances
- You can add as many instances to a cluster:

```
java -jar payara-micro-5.jar --autoBindHttp
     --clusterName myCluster --name myService1

java -jar payara-micro-5.jar --autoBindHttp
     --clusterName myCluster --name myService2
```

- We need an infrastructure that manages all these instances.

In-Memory Data Grids

- An **in-memory data grid (IMDG)** is a set of networked / clustered computers that pool together their RAMs to let applications share data with other applications running in the cluster.
- Once a new instance joins the cluster, every instance will report the current status of the Data Grid.

```
Data Grid Status
Payara Data Grid State: DG Version: 35 DG Name: myCluster DG Size: 2
Instances: {
 DataGrid: myCluster Instance Group: MicroShoal
           Name: myService1 Address: /192.168.1.11:6900
 DataGrid: myCluster Instance Group: MicroShoal
           Name: myService2 Address: /192.168.1.11:6901
}
```

- **Hazelcast IMDG** is an open source distributed In-Memory Data Grid solution for Java applications

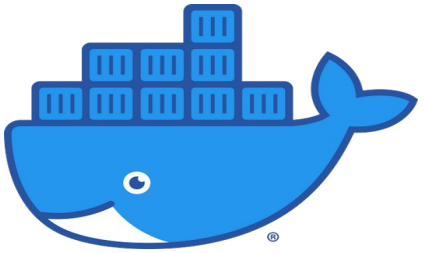
Containers and Process Isolation

- **Containerization** = bundling an application together with all of its related configuration files, libraries and dependencies required for it to run across different computing environments. Alternative or companion to **virtualization**.
- **Process isolation** = the segregation of different software processes to prevent them from accessing memory space they do not own
  - each to their own virtual address space
- *“Container deployment era”: Containers, microservices and cloud computing*

Container Benefits

- **Portability** : “write once, run anywhere”
- **Agility** (industry standards, OCI)
- **Speed** (fast start-times)
- **Fault isolation** (operates independently)
- **Efficiency** (compared to VMs)
- **Ease of management** (via orchestration)
- **Security** (safe of malicious code)
- *Implementations (Container Runtimes)*:
  - **Docker, Containerd**
  - CoreOS rkt (Red Hat), Mesos Containerizer (Apache), LXC Linux Containers, etc

Docker



- An open platform for developing, shipping, and running applications. Separates your applications from your infrastructure so you can deliver software quickly.
- Provides the ability to package and run an application in a loosely isolated environment called a **container**
- Written in the Go programming language, takes advantage of several features of the Linux kernel to deliver its functionality.
- Install:
    - *Nothing on Linux (runs natively)*
    - Docker Desktop (Win 10, Mac OS)
    - Docker Toolbox (deprecated) for Win 8

Docker Terminology

- **Image** - An image is a snapshot of how a container should look before it starts up. It will define which programs are installed, what the startup program will be, and which ports will be exposed.
- **Containe**r - A container is an image at runtime. It will be running the process and the configuration defined by the image, although the configuration may differ from the image if any new commands have been run inside the container since startup.
- **Docker Daemon** is the service that runs on your host operating system. Containers are run from the Docker daemon. The Docker CLI (command line interface) just interacts with this process.
- **DockerHub** is a central repository where images can be uploaded, comparable to Maven Central. When pulling an image remotely, it will be pulled from DockerHub if no other repository is specified.
- **Repository** - a collection of Docker images. Different images in the repository are labelled using tags.
- **Tag** - A tag distinguishes multiple images within a repository from one another. Often these tags correspond to specific versions. Because of this, when no tag is specified when running an image, the ‘latest’ tag is assumed.
- **Entrypoint** - The Entrypoint command is run when the container starts. Containers will exit as soon as the entrypoint process terminates.

Docker CLI

docker --help

- **version**
- **info**
- **build** Build an image from a Dockerfile
- **run** Download the Docker Image if needed and create and start a new container from the Image.
- **ps** Get a list of all the containers maintained by Docker Engine.
- **start** Start a container.
- **stop** Stop a container
- **rm** Remove a stopped container.
- **image ls** List of all the local images
- ...

docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

Create a simple container

- In your HelloWorld project, create the file ***Dockerfile***

```
FROM openjdk:8
COPY ./build/classes /myapp
WORKDIR /myapp
ENTRYPOINT ["java","HelloWorld"]
```

- In the Docker CLI

```
docker build -t demo:latest /d/java/MyApp
```

Create a tagged image *demo:latest*

- Run

```
docker run demo:latest
```

- List your images

```
docker image ls
```

- Push to DockerHub, save or load

```
docker push | save | load
```

Docker and MicroProfile

- *Dockerfile*

```
FROM payara/micro

COPY myapplication.war $DEPLOY_DIR
```

- From within the directory containing the Dockerfile and the myapplication.war files:

```
docker build -t myapplication .
```

- Run

```
docker run -d
           -p 8080:8080
           --name myapplication
           myapplication
```

http://192.168.99.100:8080/myapplication/

Container Orchestration

- You can containerize each microservice using Docker and create an image.
- How do you handle a large number of containers, each representing services?
- How do you:
  - discover services and perform load balancing,
  - auto scale up and down, based on the load,
  - monitor the containers health,
  - analyze various metrics?

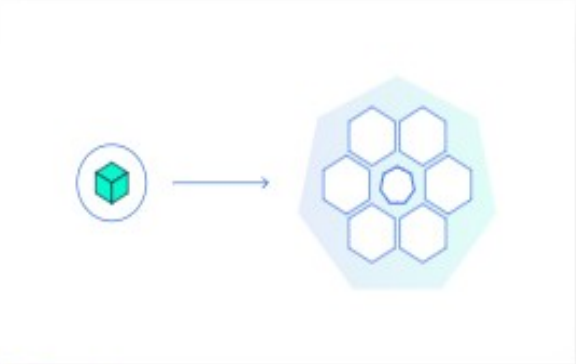
Kubernetes Benefits

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and RAM each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

Kubernetes Basic Modules

https://kubernetes.io/docs/tutorials

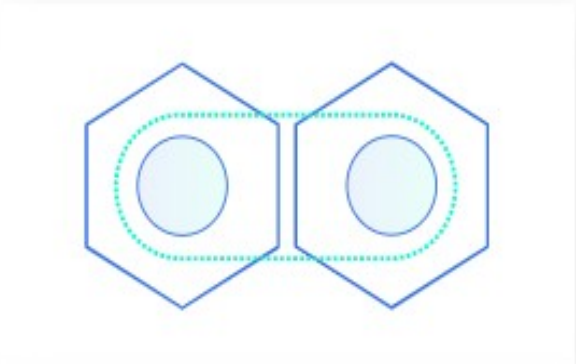
1. Create a Kubernetes cluster



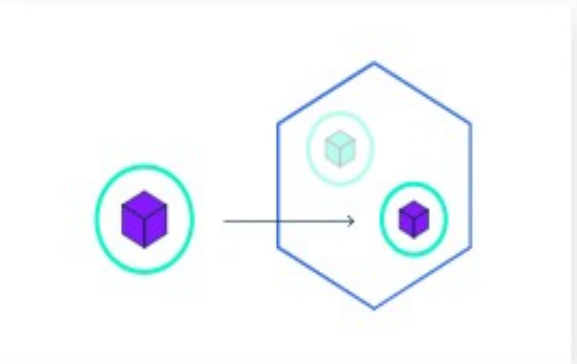
2. Deploy an app

3. Explore your app

4. Expose your app publicly



5. Scale up your app



6. Update your app

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.

Kubernetes Implementations

- **Minikube**: An open-source tool that you can install in your local machine to use Kubernetes locally. This tool uses a virtualization solution (like VirtualBox or similar) to set up a local Kubernetes cluster.
- **Google Kubernetes Engine (GKE)**: Google's solution that manages production-ready Kubernetes clusters for you.
- **Amazon Elastic Kubernetes Service (EKS)**: Amazon's solution that manages production-ready Kubernetes clusters for you.
- **Azure Kubernetes Service (AKS)**: Microsoft's
- **OpenShift Kubernetes**: Red Hat's
- ...

Hello Minikube

https://kubernetes.io/docs/tutorials/hello-minikube/

- Install Minikube
- **`minikube start`**
  - * Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
- **`minikube dashboard`**
  - General-purpose web UI for Kubernetes clusters
- *Create a Deployment (pod=group of containers)*
- *Create a Service (expose the pod to the public)*
- *Clean up, stop the Minikube VM*
- **`minikube stop`**

The kubectl command line tool lets you control Kubernetes clusters.