redhat.

Wildfly-Swarm

Does my fat-jar look fat in this?

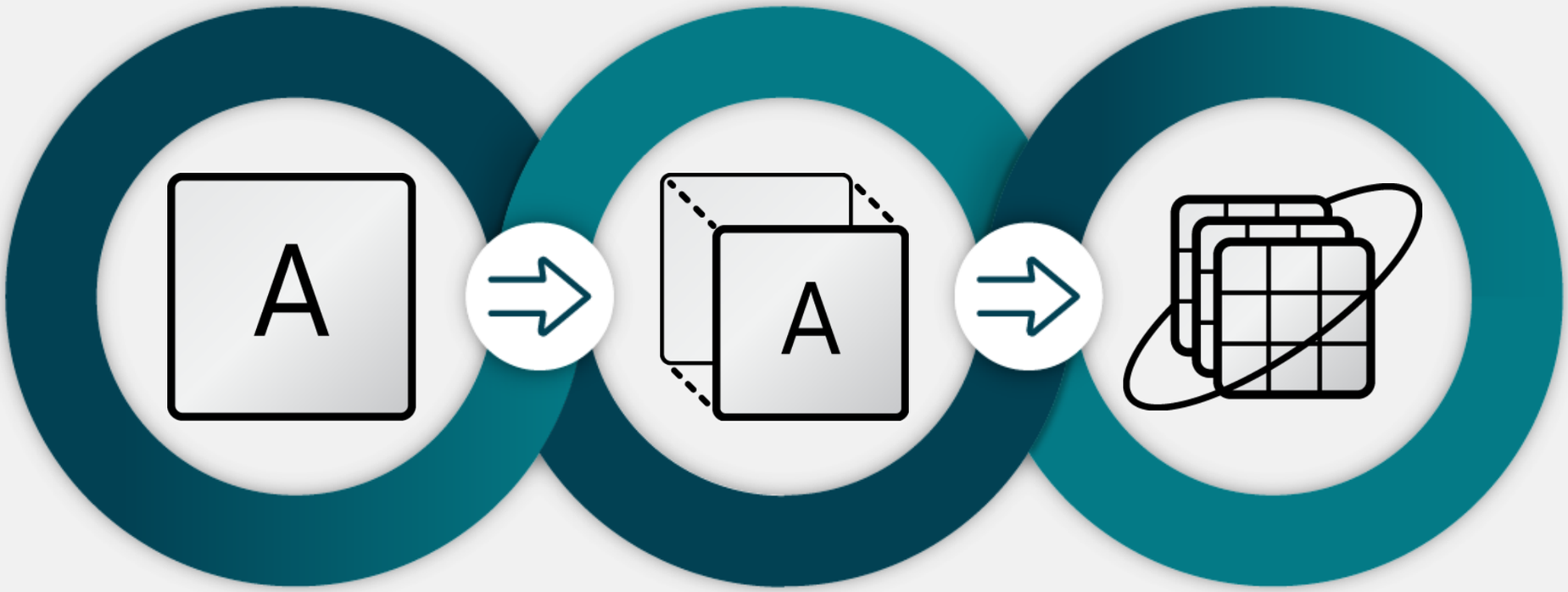
Bruno P. Georges. Director, Engineering

George Gastaldi. Principal Software Engineer

Sebastien Blanc. Senior Software Engineer

Microservices

An evolution



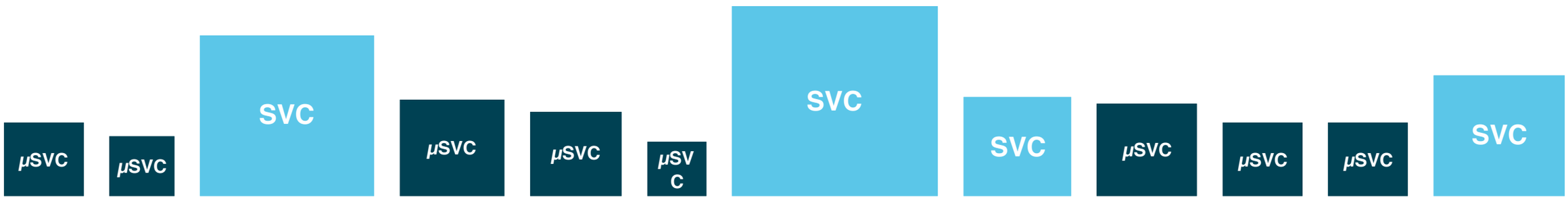
Traditional SOA Microservices

Microservices

Characteristics

- Decoupled
- Independent release cycles
- Micro functionality, not lines of code
- Ideally self-contained
- Scales independently
- All aspects owned by a 2 pizza teams

One Size != ALL



Microservices + containers

Docker ...

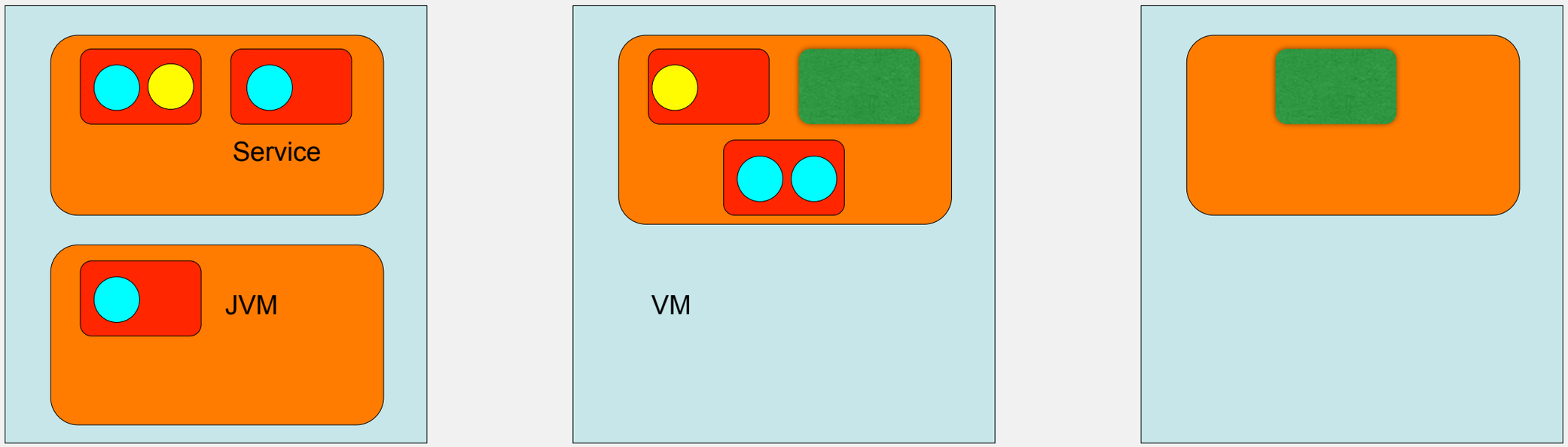
Microservices and Java EE

Building Modern architecture while preserving your investments

- Not everyone wants to use Docker
- Not everyone wants to use Node.js
- Many developers are happy with Java EE
  - Robust and mature components
  - Scalable, standards compliant, integrates well
- Not everyone wants to use all of Java EE
  - Stripping down EAP/WildFly is common
  - Higher cloud density and multi-tenancy
- JSR 111 (Java Services Framework)

Services, Linux containers and JVMs

Java EE services split across machines, containers and JVMs



Enterprise Mobile

JBoss by Red Hat

JavaEE

JBoss by Red Hat

Jopr

API (Java, Ruby, Python, C++, etc...)

HTML

Infinispan

errai

jBPM

JBossMSC

Services

Social Aspect

redhat.

Wildfly-Swarm

Just Enough App Server

- Allows Java EE components to become independently deployable (micro) services
  - Applications deploy with only the components needed
    - Just enough Application Server (JeAS)
- Re-uses existing WildFly and EAP
  - Self-contained services without wrapping it all in Docker
- Build applications as Uber jars (Java circa 1996)
  - Avian?
- The 2009 JBossAS 7 re-architecture makes it possible

Wildfly-Swarm

Just Enough App Server

- Automatic configuration
- Convention over Configuration
- Inheritance through maven dependencies
  - i.e the `jax-rs` fraction implies the `undertow` fraction, keeping the developer's `pom.xml` relatively clean
- Increase developer productivity.
  - Now available as Forge add-on !
- Beyond Java-EE
  - Netflix OSS - Ribbon, Hystrix
  - Logstash

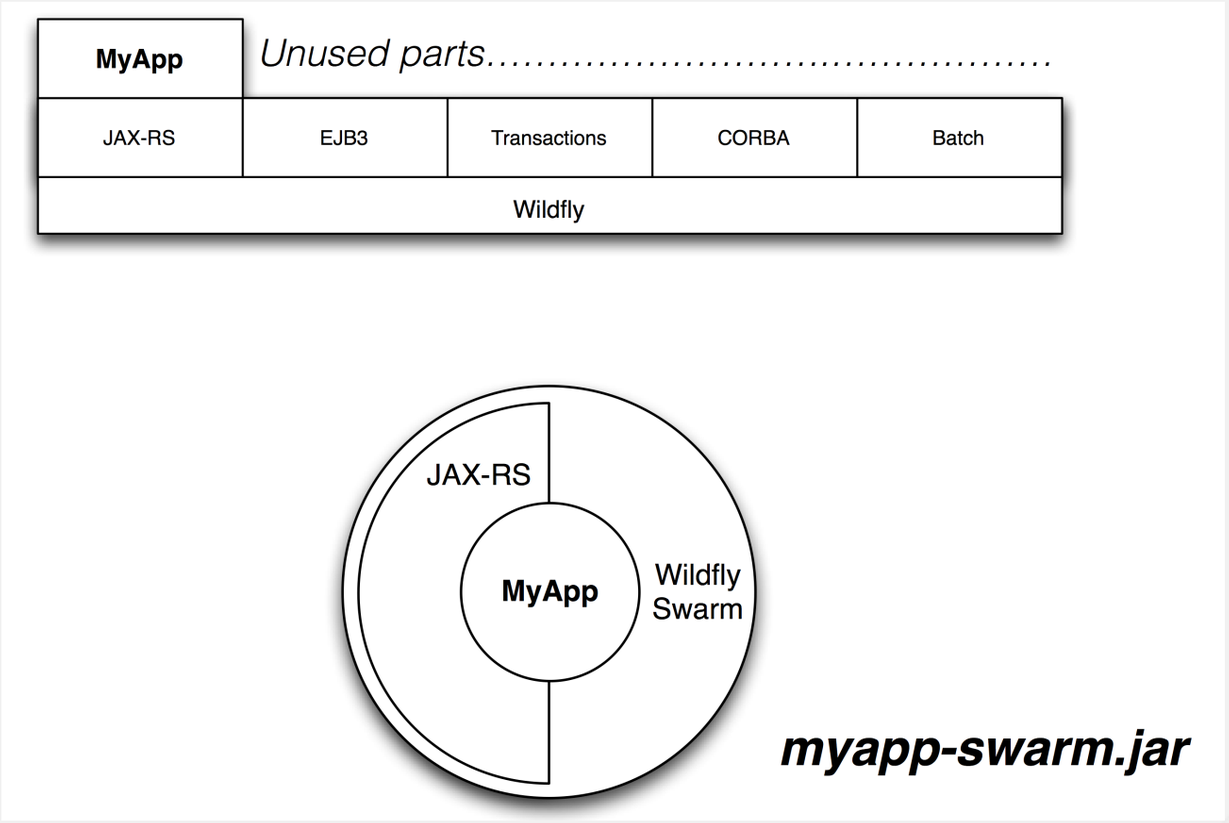
Where might it be useful

- Building EE applications with limited capabilities
  - Comfortable with the Java EE model
- Need multiple components/services for business logic
  - WildFly-core handles class loading and lifecycle issues
- More streamlined “virtual” application server
  - Shared services
  - Multi-tenancy/higher densities
- Microservices (aka SOA)

Just Enough App Server

or, Wildfly broken apart

- Maven Addressable components
- Fat-jarrable
- you can provide your own `main(..)`
- Pragmatic configuration
  - instead of providing your own `standalone.xml`



How does it work?

JAX-RS resource

```
@Path("persons")
public class PersonResource {

  @Inject
  PersonDatabase database;

  @GET
  @Produces("application/xml")
  public Person[] get() {
    return database.currentList();
  }

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Person get(@PathParam("id") int id) {
    return database.getPerson(id);
  }
}
```

JAX-RS resource with WildFly Swarm

```
@Path("persons")
public class PersonResource {

  @Inject
  PersonDatabase database;

  @GET
  @Produces("application/xml")
  public Person[] get() {
    return database.currentList();
  }

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Person get(@PathParam("id") int id) {
    return database.getPerson(id);
  }
}
```

Spot the difference

JAX-RS Resource

```
@Path("persons")
public class PersonResource {

  @Inject
  PersonDatabase database;

  @GET
  @Produces("application/xml")
  public Person[] get() {
    return database.currentList();
  }

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Person get(
      @PathParam("id") int id) {
    return database.getPerson(id);
  }
}
```

JAX-RS Resource with Wildfly-Swarm

```
@Path("persons")
public class PersonResource {

  @Inject
  PersonDatabase database;

  @GET
  @Produces("application/xml")
  public Person[] get() {
    return database.currentList();
  }

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Person get(
      @PathParam("id") int id) {
    return database.getPerson(id);
  }
}
```

What did change ?

Maven Plugin

```
<plugin>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>wildfly-swarm-plugin</artifactId>
  <version>${version.wildfly-swarm}</version>
  <executions>
    <execution>
      <goals>
        <goal>package</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <contextPath>demo</contextPath>
  </configuration>
</plugin>
```

What did change ?

Fractions

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jpa</artifactId>
  <version>${version.wildfly-swarm}</version>
</dependency>
```

Wildly-Swarm

Fractions

- Wildfly Subsystems via Fractions
- Fraction:
  - Defines a module.xml for a jar(s), with any required module dependencies
- Typical usages:
  - Non WildFly subsystems, ie. RxJava, RxNetty
  - Activating WildFly modules that are excluded by default

WildFly Swarm

Fractions

Wildfly Subsystem

JAX-RS
Datasources
EJB
JMX
JSF
Mail
JPA
CDI/Weld
Bean Validation
Clustering
Keycloak
Infinispan
Removing
Hawkular
Undertow

Non-Wildfly

LogStash
Ribbon
Ribbon Secured
Hystrix
RxJava
RxNetty

Wildly-Swarm

main or not main

- main() not a big player in server-side Java EE
- Much is defaulted to ease developers burden
- If you have no main() then a default Container is created and every fraction is defaulted
- If you provide a main() you can configure any fractions
  - Any not explicitly configured will have defaults
- Could also do a lot more in main()
  - Locate other services?
  - Dynamically adapt components?

Wildly-Swarm

main or not main

```
public static void main (String [] args) throws Exception {

  Container container = new Container();
  JAXRSArchive deployment = ShrinkWrap.create( JAXRSArchive.class, “myapp.war");
  deployment.addClass(Hola.class);
  deployment.as(Secured.class).protect(“/rest/hola").withMethod( "GET" ).withRole("admin");
  deployment.addAllDependencies();
  container.start().deploy(deployment);

}
```

Stop talking... show some code

Use Forge's wildfly-swarm add-on

The most efficient way to start your swarm based microservice

Pre-requisite:

- maven 3.3.8 or later
- JDK1.8 [needed by Swarm]
- Forge 3.0.0.Beta4 or later

1. Enable swarm

```
$ forge
$ addon-install-from-git --url https://github.com/forge/wildfly-
swarm-addon.git
```

Use Forge's wildfly-swarm add-on

The most efficient way to start your swarm based microservice

2. Forge your project

```
$ project-new –named demo –stack JAVA_EE_7
$ wildfly-swarm-setup
# create your JPA Entities
$ jpa-new-entity –named Author
$ jpa-new-field –named named
$ ….
```

Getting started

What do you need to do to build an executable jar

Build your project

```
mvn package
```

Run your project

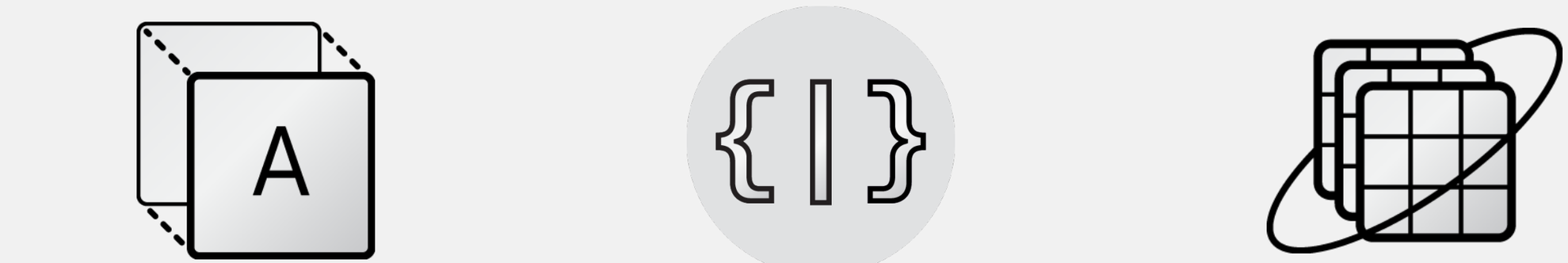
```
mvn wildfly-swarm:run
# or
java -jar target/myapp-swarm.jar
```

or... in your IDE

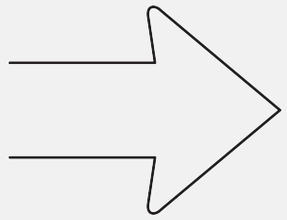
```
org.wildfly.swarm.Swarm
com.mycompany.myapp.MyMain
```

Swarm My App

Decomposing Java EE monolith app into Microservices



Monolith



Micro Services

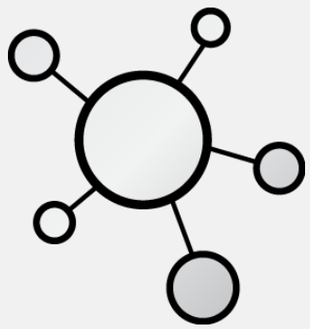
Deeper Dive

full booker demo

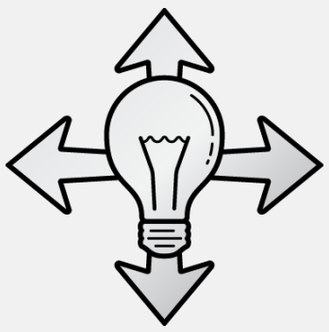
https://github.com/wildfly-swarm/booker/blob/master/BOOKER_WITH_DOCKER.md

Conclusions

“And miles to go before I sleep”, Robert Frost
Fast iteration, space to grow



Wildfly Swarm is a community
@wildflyswarm, #wildfly-swarm, github.com/wildfly-swarm, wildfly.org/swarm



Share your ideas
We are interested in feedback and input on directions

Thank you!

redhat.