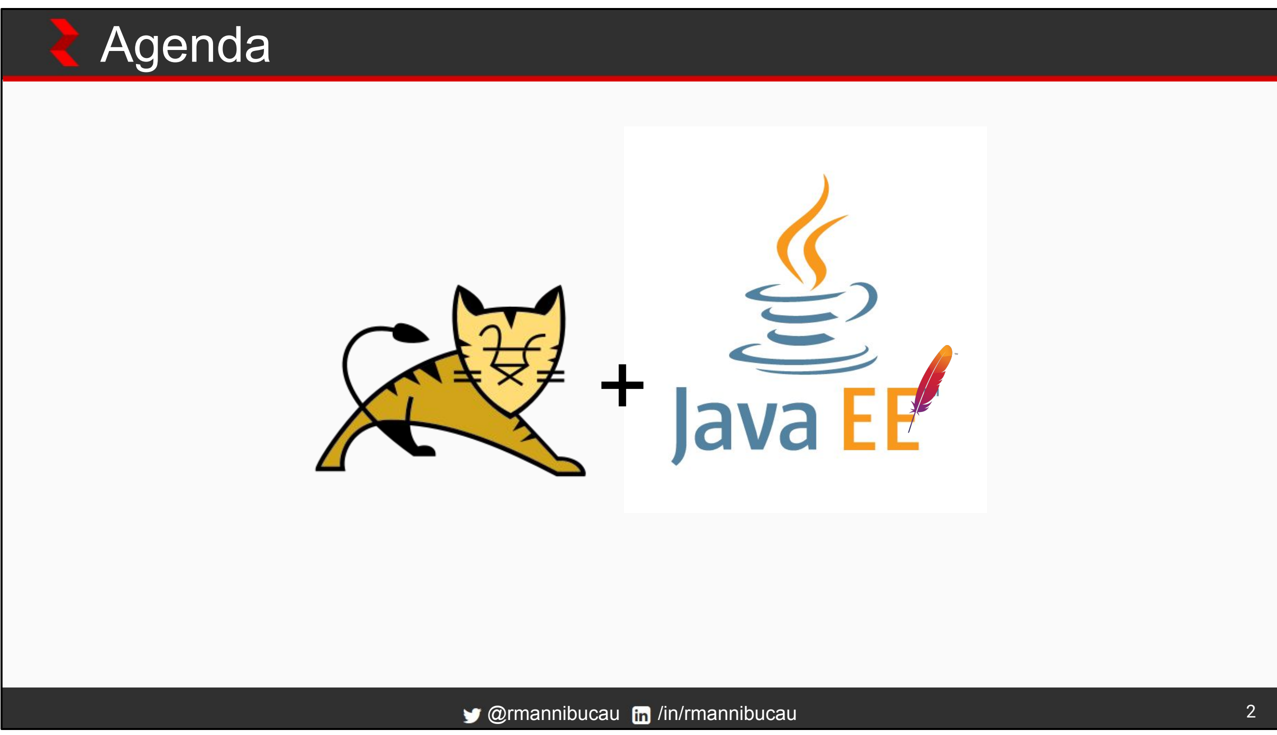
Romain Manni-Bucau

ApacheCon NA 2017

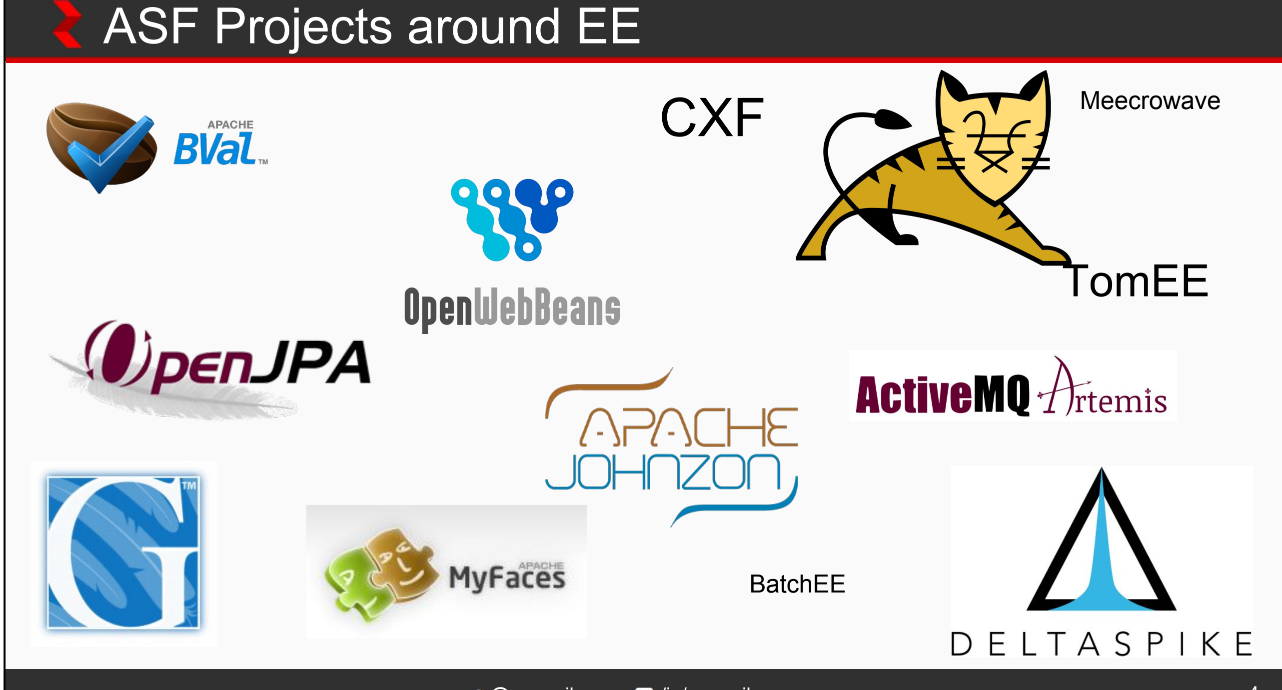


[joke] “well, it’s almost lunch time... but beers can wait! This presentation is a healthier option ”

Agenda:

- How ASF builds on top of Tomcat EE oriented servers/suites and applications
- We will go through 2 main parts (segments/target, ergonomy, tools)
    - TomEE which targets EE profiles
    - Meecrowave which targets usability, consistency and easiness

- I am an ASF committer since 2011
- You can find me on various social networks such as : twitter/github/linkedin
- I mainly worked in insurance/bank field, essentially in a transversal team working with development and operations teams and business unit.
- Today I’m doing product development with tomitribe.
- During my little spare time, I write JavaEE posts on my blog hosted on TomEE on Openshift (yeah) (see the address in the slide)
- It mainly means that I have a lot of development background and some production knowledge



Lot of activity around EE (limiting to EE as a stack, not softwares built on top of EE)

- Bean validation to validate objects and/or method inputs/outputs
- OpenJPA as JPA implementation (database -> java mapping)
- OpenWebBeans as CDI implementation: the EE IoC and basis since EE 6
- Geronimo for a set of utilities (we ignore the server which is dead now): connector for JCA, transaction manager for JTA, xbean as a generic resource factory, classpath scanning, telnet impl…
- Johnzon as JSON swiss knife (JSON-P for raw JSON creation, streaming etc), JSON-B for object mapping, mapper for more advanced usages and websocket, JAX-RS integrations
- ActiveMQ (Artemis or not) as JMS implementations
- MyFaces as JSF implementation
- Tomcat - should we present it - as Servlet/JSP/WebSocket/Jaspic implementation
- BatchEE as JBatch solution
- DeltaSpike as EE companion. It is a toolsuite/library built on top of CDI providing advanced features for CDI like configuration, JPA integrations, JSF advanced features etc…
- CXF as the most powerful webservice solution (JAX-WS, JAX-RS, Oauth2, …, fediz)
- TomEE/OpenEJB: as EJB and EE implementation

So we have a lot of EE power at ASF but even if communities work closely together we still “run” in parallel. That’s where TomEE and Meecrowave comes into the game: pick a consistent set of items of the stack and make them working together out of the box.

Reminder: today “EE” ecosystem

- EE 6: full/web profile
- Microprofile
- Spring boot

- EE 6: “EE” is too big and heavy -> split in 2 -> web profile
- Microprofile: Oracle doesnt really move on EE 8 so let’s make something without them + is web profile too big to start? Make CDI the *actual* center of the programming model!
- Spring boot: can spring be self contained for a full app (including the web server + autoconfiguration like in EE) ~= catch up its lateness on EE, make it > 2010s

Since its beginning OpenEJB/TomEE targets classpath deployment but these last years with microservices and productivity target, it is going mainstream, helped a lot by major vendor dynamism.

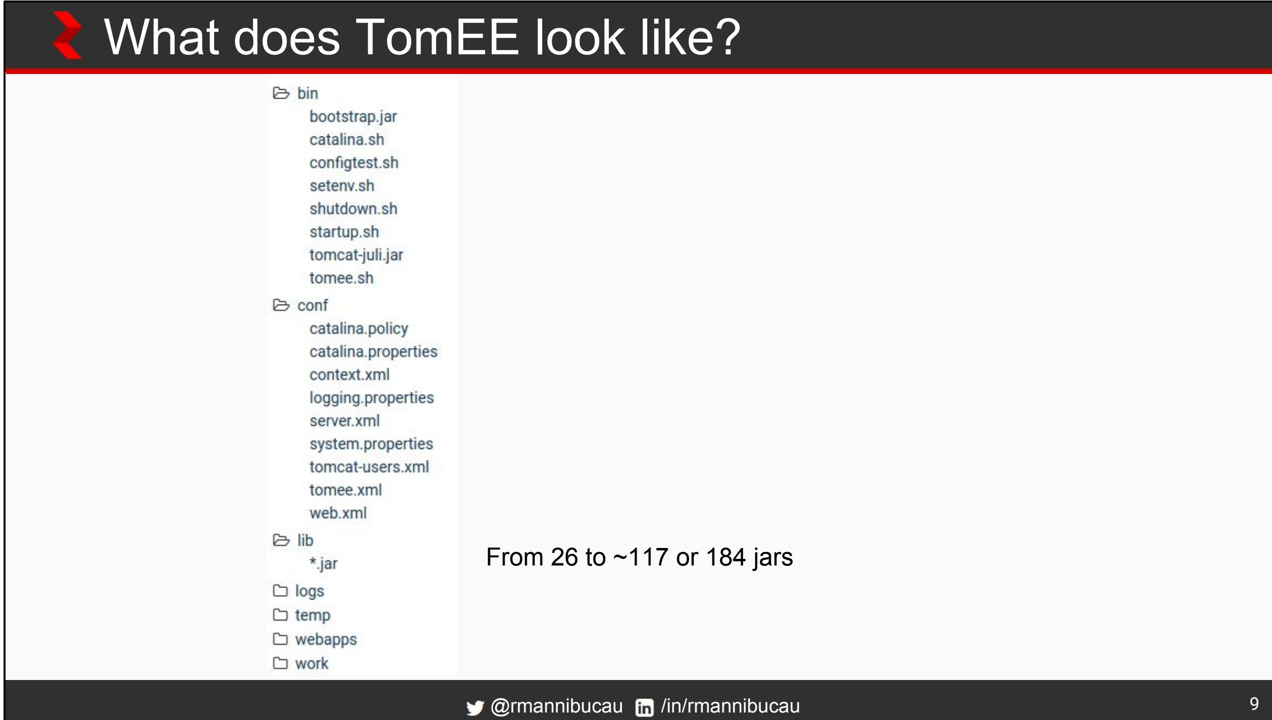
Part I > TomEE

Let’s dig into what TomEE is.

What TomEE is

- EE 6 certified
- EE 7
- Tomcat based
- Light!
- Usage oriented

Big lego game to assemble all EE components, ~30 specs for the full profile and ~14 for the web profile.
Lot of effort
Usage oriented: deploy classpath since its conception (1999) vs spring boot has few years ;) - "tomee spirit originally coming from "put the container in the container"" meaning putting openejb into Tomcat. And that's what is TomEE: we take Tomcat, add some libraries (the stack we saw previously more or less) and we just have a Tomcat Listener managing the lifecycle of TomEE….that's the high level view at least, in practise we need some deeper integration but spirit doesn't change.



Zip/tar.gz -> decompress (like Tomcat)

TomEE keeps Tomcat layout, “stay Tomcat”. Just adds a few files and jars.

- Jars corresponding to previous spec (from ~4 specs to 15/28)
- Files:
    - Tomee.sh : some tomee specific utilities like digesting password in resources for instance
    - tomee.xml : standard way to define tomee containers (EJB) and resources (resource adapter like JMS, datasources, customs etc…)

=> still is Tomcat for ops! Same tooling, monitoring etc...!

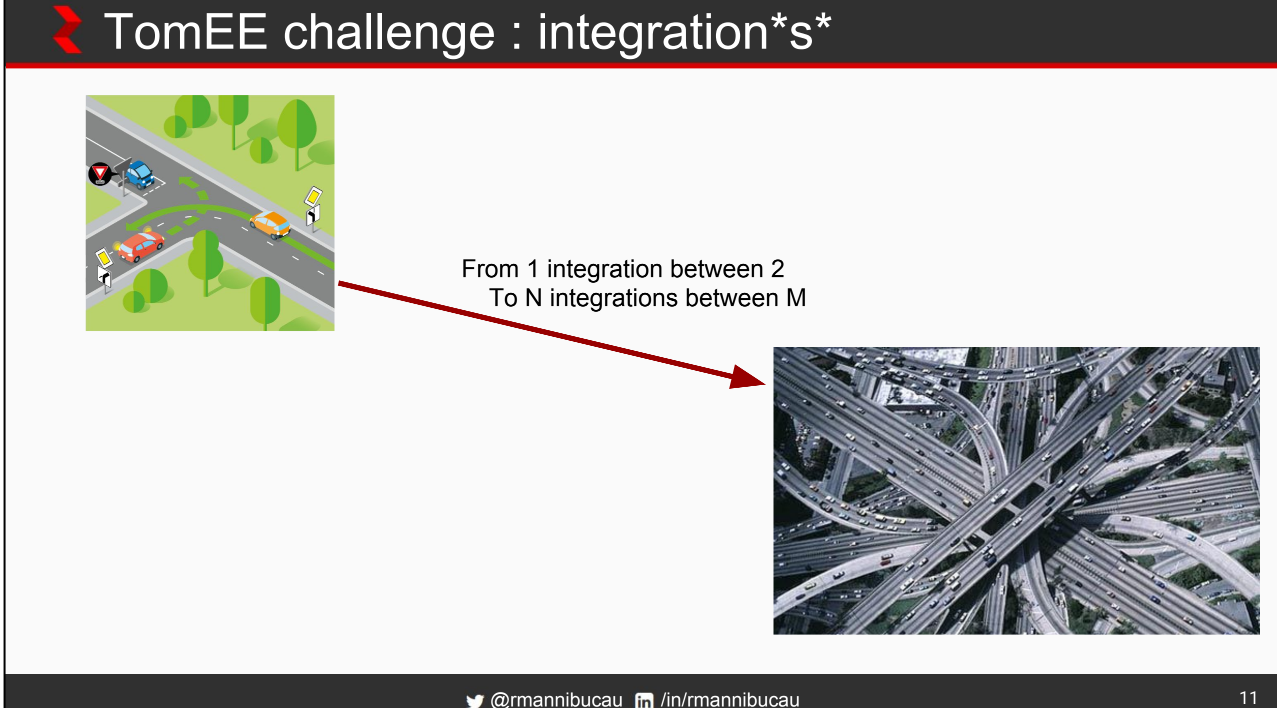
Some TomEE resource

```
<Resource id="MySQL" aliases="myAppDataSourceName" type="DataSource">
   JdbcDriver = com.mysql.jdbc.Driver
   JdbcUrl = jdbc:mysql://${OPENSHIFT_MYSQL_DB_HOST}:${OPENSHIFT_MYSQL_DB_PORT}/rmannibucau?tcpKeepAlive=true
   UserName = ${OPENSHIFT_MYSQL_DB_USERNAME}
   Password = cipher:Static3DES:ULkcoVik7DM=
   ValidationQuery = SELECT 1
   ValidationInterval = 30000
   NumTestsPerEvictionRun = 5
   TimeBetweenEvictionRuns = 30 seconds
   TestWhileIdle = true
   MaxActive = 200
 </Resource>
```

```
┌( rmannibucau @ alienware )─( 11:20 -:- 05/12 )
└( «he-tomee-webprofile-7.0.3 ) -> ./bin/tomee.sh cipher -c Static3DES test
ULkcoVik7DM=
```


- Mix of XML + Properties to have a good readiness
- Ciphering support with custom algorithm (SPI)
- Advanced datasource features like logging SQL, flushing a datasource (recreate it), reset on error (recreate on some kind of error), pluggable pool support (tomcat, dbcp, …)
- Placeholder support, aliases support (link in JPA but single resource definition), auto wiring and auto creation of resources etc…, duration syntax
- Tomee.xml equivalent embedded in the webapp at WEB-INF/resources.xml

Powered by xbean-reflect + some glue, yeah!



Big lego game to assemble all components
Requires a lot of effort.
Ex:

1. Your app requires CDI so you import OpenWebBeans, no real work but adding a dependency
2. Your app needs to expose a HTTP endpoint so you need to import Tomcat to get servlet -> you integrate OWB and Tomcat
3. Your app implements a Batch triggered by a HTTP endpoint -> you import BatchEE but need to ensure it works with OWB and Tomcat (classloader consistency, context, configuration, …), so 3 integrations
4. The HTTP endpoint uses JSON so you import johnzon -> you need to ensure provider works with Tomcat but is also integrated with CDI and with the HTTP layer (Servlet, JAX-RS) etc…
5. ….

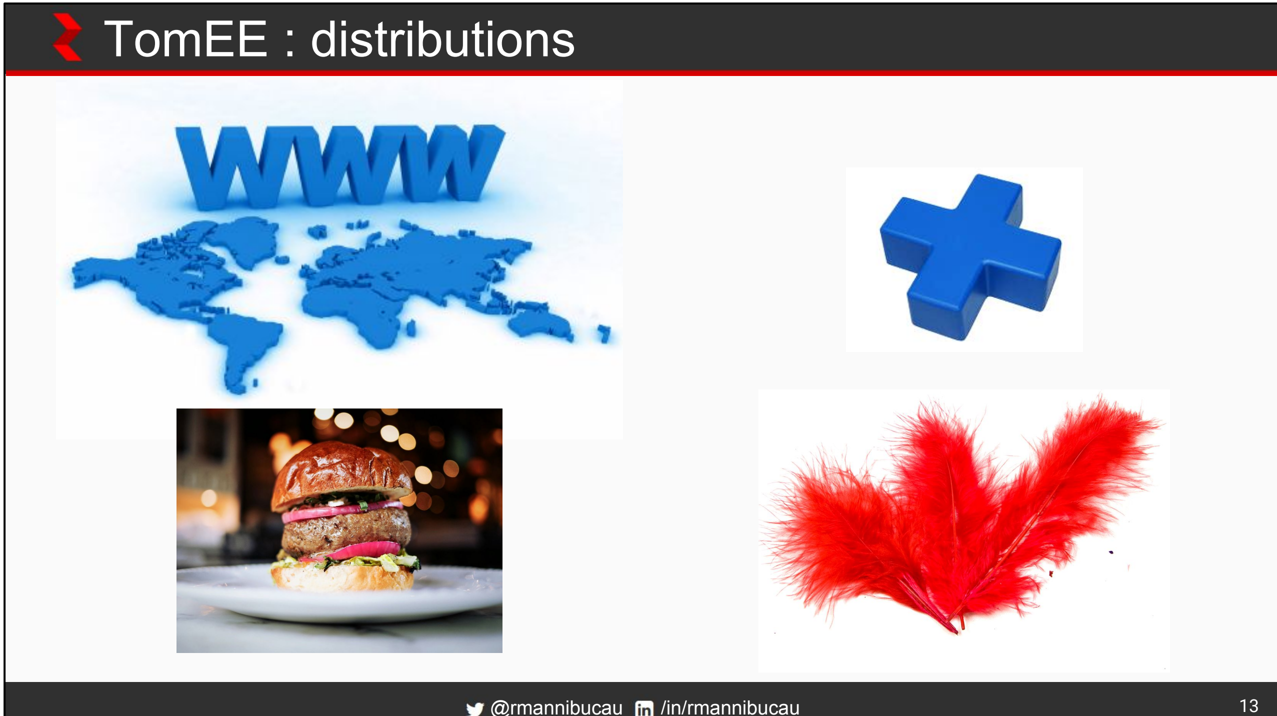
A lot of integrations to take care and to do right to avoid surprises

TomEE challenge : integration*s*

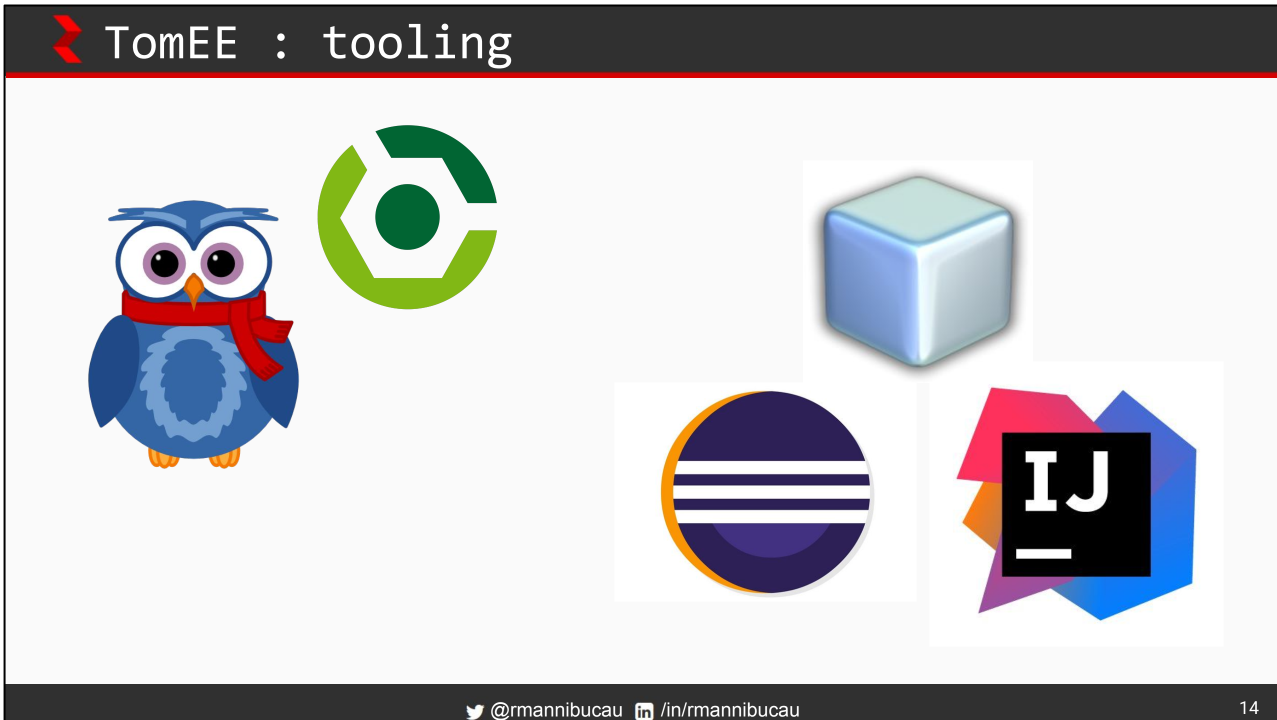
CDI
+
Servlet
+
JAX-RS
+
JSON-P/JSON-B
+
…
≠
It works
≠
EE

Conclusion is: if you build your own you need to keep in mind integration is made as smooth as possible by frameworks but you can still hit some pitfalls

- Multiple scanning
  - Servlet scans
  - EJB scans
  - JPA scans
  - CDI scans
  - JSF scans
  - …
  - => startup time > x5
  - => TomEE does it once!



Web profile: considered as the most used specs, see it as “modern” WebService (JAX-RS+ JSON + Servlet + Bean Validation),persistence (JPA), EJB (tx) -> SPA enabler!
Plus: jaxws, jms: enterprise flavor
Plume: ~webprofile with oracle stack (mojarra/eclipselink) -> request from users cause JSF/JPA are the worse specs in term of portability, mainly due to glassfish activity decrease
Embedded: based on webprofile but customizable as any app -> control at the price of the “knowledge” (integrations)



Maven:

- Releases on central
- Plugins (from local dev to run)
  - Standalone: fully customizable run to distribution build (zip/tar.gz)
  - Embedded: à la jetty, mvnDebug compatible (no fork), in place resource support (src/main/webapp)

Gradle:

- Recent embedded plugin: fully based on maven flavor with very few features in less, doesnt pollute gradle classpath! (custom configuration)

IDE:

- Adapter integration (based on tomcat and more)
  - Intellij, Eclipse (WTP - same pitfalls as with tomcat), Netbeans

TomEE : maven example

```
<plugin>
  <groupId>org.apache.tomee.maven</groupId>
  <artifactId>tomee-maven-plugin</artifactId>
  <version>${tomee7.version}</version>
  <configuration>
    <tomeeClassifier>plus</tomeeClassifier>
    <debug>false</debug>
    <debugPort>5005</debugPort>
    <args>-Dfoo=bar</args>
    <config>${project.basedir}/src/test/tomee/conf</config>
    <libs>
      <lib>mysql:mysql-connector-java:5.1.20</lib>
    </libs>
    <webapps>
       <webapp>org.superbiz:myapp:4.3?name=ROOT</webapp>
       <webapp>org.superbiz:api:1.1</webapp>
    </webapps>
    <apps>
        <app>org.superbiz:mybugapp:3.2:ear</app>
    </apps>
    <libs>
        <lib>mysql:mysql-connector-java:5.1.21</lib>
        <lib>unzip:org.superbiz:hibernate-bundle:4.1.0.Final:zip</lib>
        <lib>remove:openjpa-</lib>
    </libs>
  </configuration>
</plugin>
```

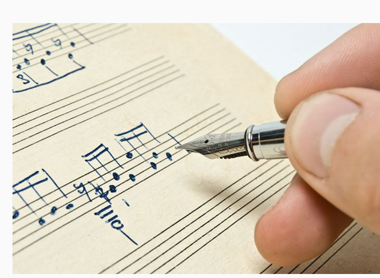
```
$ mvn tomee:run
$ mvn tomee:build
```


- Which artifact
- Debugging
- Add libs/apps/webapps/javaagent
- Config (files or inline -> systemVariables, tomeeXml etc...)

Embedded version: “F5”

- ApplicationComposer: pre-arquillian erea: control what you deploy for the test(s)
- Arquillian: JBoss movement, inspired from TCK (the suite validating you are EE/spec compliant)
  - Describe your application
  - Write JUnit/TestNG test as usual
  - Lifecycle is handled by the runtime (integration through an adapter)
- TomEE Embedded
  - Real embedded flavor (easy debugging), classpath deployment (awesome for application dev, bad for framework where each test has its own deployment)

TomEE : Testing / ApplicationComposer

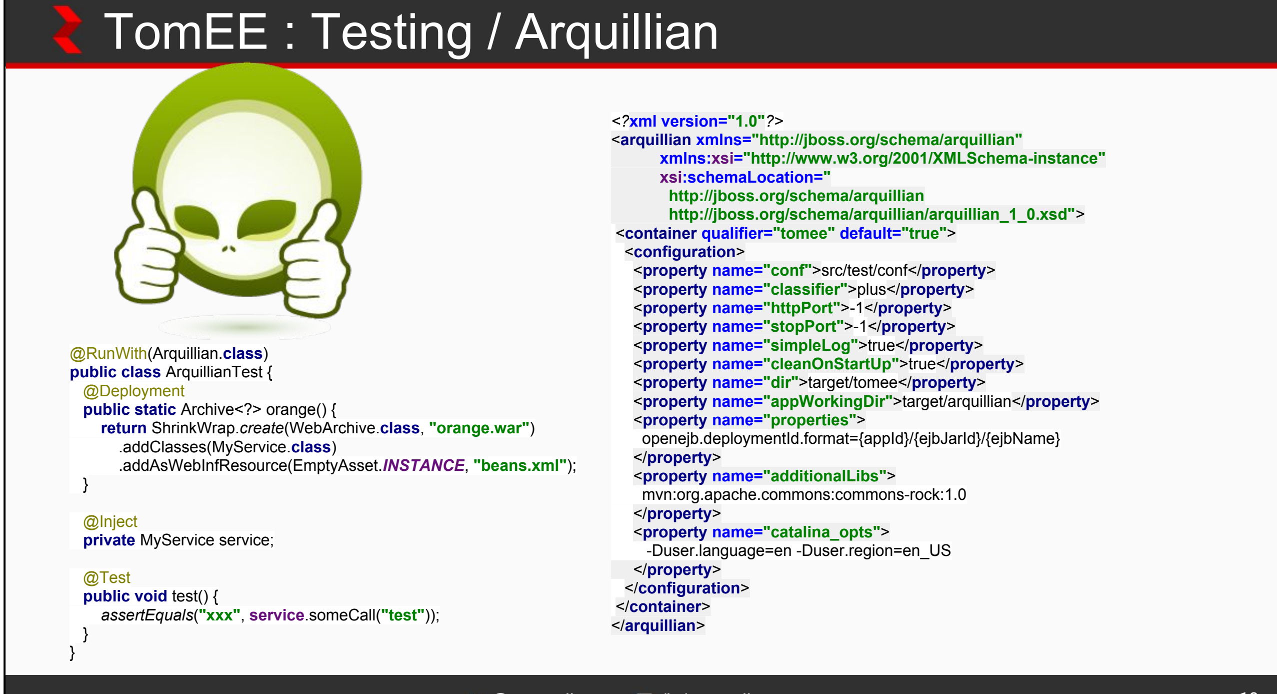


```java
@EnableServices("jaxrs")
@JaxrsProviders(JohnzonProvider.class)
@Classes(cdi = true, value = GenericClientService.class, ConfigurationProducer.class, OAuth2Mock.class)
public class GenericClientServiceTest {
  @Rule
  public final TestRule app = RuleChain.outerRule(new SftpServer())
                                       .around(new CassandraRule(this))
                                       .around(new ApplicationComposerRule(this));

  @RandomPort("http")
  private URL base;

  @Inject
  private SomeService service;

  @Test
  public void test() {
    assertEquals("xxx", service.someCall("test"));
    // or jaxrs client api using base
  }

  @Path("mock/oauth2")
  public static class OAuth2Mock {
    @POST @Path("token")
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public String token(final MultivaluedMap<String, String> form) {
      return "{\"access_token\":\"test-token\",\"token_type\":\"bearer\"}";
    }
  }
}
```


- As a runner or a rule - or testng!
  - Rule enables composition, allows to order test "components" like
    - Start a ftp server
    - Start cassandra
    - Start Elasticsearch
    - Start the app (which uses the 3 of them) with placholders for the config if you use random ports (${cassandra.port})
- Fully embedded
- Random *ports* (amq too) for tomee services (sets a system property and tomee knows how to override ports from system properties)
- Services: jaxrs/jaxws, … light http layer, not tomcat
- server/client testings in the same JMV, allows injections + client usage in the same test!
- Easy mocking: you control what you deploy so just put the mock impl, same for server, service=jaxrs + test endpoints etc...

- Close to arquillian without actual packaging phase

ApplicationComposer input is the deployment model and not a binary so it is very fast and efficient
Usage in standalone!



- JBoss powered solution, de factor standard for EE testing
- Relies on a container adapter (managing server lifecycle and grabbing the deployment and deploying it to the server -> vendor specific API)
- Lot of extensions (/!\ their stack and dependencies, can be huge and conflicting) - including for selenium, angular, spock (groovy BDD), spring...
- TomEE has:
    - Remote
        - Running instance pre-setup
        - Downloaded and managed by the adapter
    - Embedded
        - Same JVM as the test, still TomEE
    - OpenEJB (more than embedded, no zip dump)
        - Same JVM as the test, no deployment dump, no tomcat. Same kind of logic than application composer but based on arquillian API
- Configuration ~ maven plugin, really complete and based on maven coordinates for artifacts! Random ports friendly!

- Supports groups of container, ie you can test clustering!
- Some specific features like deploy once an application (and run a full client suite -> gain a lot of time, can do a x20 on a suite of 21 tests in term of time cause IO are reduced a lot)

TomEE : Testing / Others

```java
@RunWith(TomEEEmbeddedSingleRunner.class)
public class NoScannerSingleRunnerTest {
  @Application
  private ScanApp app;

  @TomEEEmbeddedApplicationRunner.Args
  private String[] args;

  @Test
  public void run() {
     app.check();
  }

  @Application
  @Classes(value = ScanMe.class)
  @ContainerProperties({
     @ContainerProperties.Property(
       name = "app.database.url", value = "jdbc:h2:mem:test")
  })
  public static class ScanApp {
     @Inject
     private ScanMe ok;

     @Inject
     private Instance<NotScanned> ko;

     public void check() {
        assertNotNull(ok);
        assertTrue(ko.isUnsatisfied());
     }
  }
}
```

JUnit

```java
@Properties({
  @Property(
    key = DeploymentFilterable.CLASSPATH_INCLUDE,
    value = ".*app.*")
})
@RunWith(EJBContainerRunner.class)
public class TestWithCdiBean {
  @Inject
  private CdiBean cdi;

  @Inject
  private EjbBean ejb;

  @Test
  public void test() {
     ...
  }
}
```


- The application instances can have injection (CDI, EE…)
- Main[] args can be injected
- ApplicationComposer light model support
    - Random port
    - Classes
    - …
- Single (or not) runner/rule (shared by all tests, makes it insanely fast for suites!)
- And much more JUnit integration coming from openejb (was test oriented a lot originally)
- TomEEEmbeddedSingleRunner / AppComposer usable to launch mains!

TomEE : Some more goodness

- System properties
  - Container/Resource definition
  - Overriding (MDB, resource placehodlers)
- Container events
- Resource templates/properties provider
- Reloadable PersistenceUnit
- @MBean
- Lazy tomcat components (valve, realm) to implement them from the webapp classloader
- ...

- The application instances can have injection (CDI, EE…)
- Main[] args can be injected
- ApplicationComposer light model support
  - Random port
  - Classes
  - …
- Single (or not) runner/rule (shared by all tests, makes it insanely fast for suites!)
- And much more JUnit integration coming from openejb (was test oriented a lot originally)
- TomEEEmbeddedSingleRunner / AppComposer usable to launch mains!

Part II > Meecrowave

Now let see what meecrowave is and how it differs from TomEE.

No, Meecrowave is not a microwave with an embedded EE server to manage the clock...could be fun but no.
It is a subproject of OpenWebBeans ASF Top Level Project created last year.

More and more applications requirements turn around:

- Single Page Application
- Microservices

=> JAX-RS + JSON are the first citizen

OpenWebBeans is a strong CDI Impl, CXF a strong JAXRS impl, Johnzon a string JSON-P/B implementation, Tomcat is a strong backbone for both! So let’s glue them!

Kinf of microprofile but not explicit yet. Very ASF opiniated (no portable goal but high integration quality)

Let’s make it running OOTB!

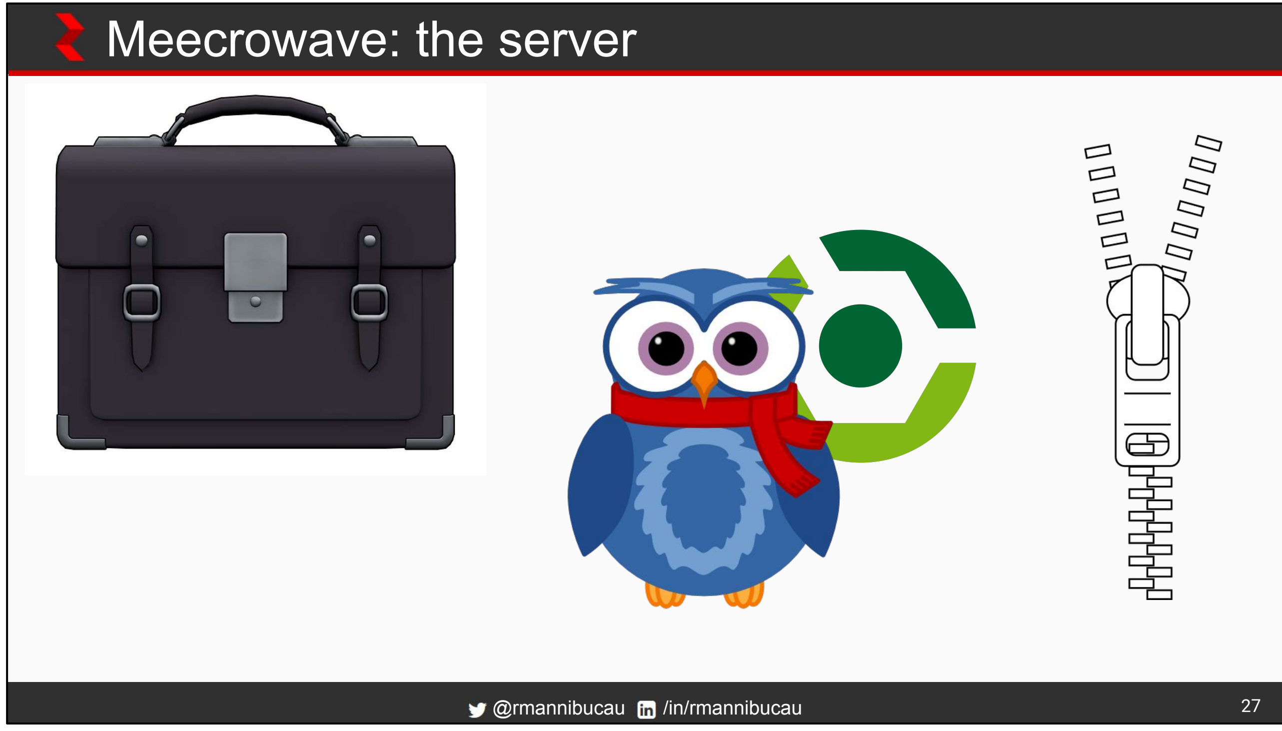
Mainly created from the tomee-embedded experience (“extract the best”) and last year developments. One small but impacting change is the usage of log4j2 by default which makes the logging way more powerful than default JUL.

Meecrowave: scope

- Server (tomcat based)
- Test solutions
- Tooling/dev

All dev cycle:

- Server: a runtime for production, based on tomcat features even if layout is different (we'll see it later and why)
- Testing: if a server is great in prod but bad in dev the productivity will be reduced a lot, in particular in microservice erea so it would be a poor server, meecrowave therefore targets testing as well
- Tooling: for the same reason the server needs to be easy to use in dev and allow to share configuration easily, that's why there are some additional plugins in meecrowave



- Play classpath run (java -cp *.jar MyMain)
- Jar “bundles”
    - App shade
    - Ready to run uber jar (Cli)
        - Reshaded
        - java -jar meecrowave-runner.jar -webapp app.war
- Through dependencies (maven/gradle) + custom packaging [+ deployment] // assemblies
- Maven plugin to create a ready to run zip/tar.gz (based on tomcat idea)

Meecrowave can be seen as a launcher or a library (vs a full container you run in), closer to main(String[] args), more flexible

Meecrowave: programmatic flavor

```
new Meecrowave().bake().await()
```

```
final Meecrowave.Builder builder = new Meecrowave.Builder();
builder.setHttpPort(8080);
builder.setScanningPackageIncludes("com.company.app");
builder.setScanningIncludes("app-");
builder.setConf("app/conf");
builder.setJaxrsMapping("/api/*");
builder.setJsonpPrettify(true);
builder.setRealm(new RealmBase() {
  @Override
  protected String getPassword(final String s) {
    return "test";
  }

  @Override
  protected Principal getPrincipal(final String s) {
    return new GenericPrincipal(s, "test", singletonList("admin"));
  }
});
builder.instanceCustomizer(t -> t.getHost().getPipeline().addValve(new ValveBase() {
  @Override
  public void invoke(final Request request, final Response response) throws IOException, ServletException {
    response.getWriter().write("custom");
  }
}));
// ...

try (final Meecrowave meecrowave = new Meecrowave(builder).bake().deployWebApp(new File("app.war"))) {
  meecrowave.await(); // or any command handling/CLI fitting your soft
}
```


- Meecrowave is a wrapper to Tomcat (tomcat embedded launcher), it is fully customizable and integrates smoothly with any application need (CLI or not)
- Configuration comes from simple ones as port numbers to advanced ones like tomcat pipelines/valves without forgetting all the defaults (providers, scanning, …)
- All programmatic but also loadable from meecrowave.properties automatically or the CLI (configuration is wired from this class to all the integrations like arquillian, maven, gradle, …)

Meecrowave: CLI flavor

```
$ java -jar meecrowave-bundle.jar \
>> --http 8080 \
>> --tomcat-access-log-pattern ‘%h %l %u %t "%r" %s %b’ \
>> --web-resource-cached true \
>> --meecrowave-properties /home/meecrowave/defaults.properties
```


- Based on the Builder (1-1)
- Can deploy classpath or .war(s)
- Ops can still rely on a global meecrowave.properties if they need to avoid to repeat the configuration on each instance, same for logging, yeah!
    - Very smooth with microservices when machines are mutualized

Meecrowave: build tools/dev

```
apply plugin: 'org.apache.meecrowave.meecrowave'

meecrowave { // ~builder
  httpPort = 9090
}

$ gradle meecrowave
```

```
<plugin>
  <groupId>org.apache.meecrowave</groupId>
  <artifactId>meecrowave-maven-plugin</artifactId>
  <version>${meecrowave.version}</version>
</plugin>

$ mvn meecrowave:run
$ mvn meecrowave:bundle

.
├── bin
│   └── meecrowave.sh
├── conf
│   ├── log4j2.xml
│   └── meecrowave.properties
├── lib
│   └── *.jar
├── logs
│   └── meecrowave.log
└── temp
```


- Config 1-1 with Cli/Builder but aligned on the related build tool
- 2 goals for maven (vs 1 for gradle: run)
  - Run (same for gradle) to start the configured instance
  - Bundle to create a small ready to run zip
    - Close to tomcat layout
    - Compatible with home/base if you extract meecrowave-core from lib and move it elsewhere

Meecrowave: testing

```
public class MyResourceTest {
  private final MonoMeecrowave.Rule container = new MonoMeecrowave.Rule();

  @Rule
  public final TestRule rule = RuleChain.outerRule(MySQLRule.instance()).around(container);

  @Test
  public void test() {
    final Client client = ClientBuilder.newClient();
    try {
      final WebTarget base = client.target("http://localhost:" + container.getConfiguration().getHttpPort() + "/api");
      //...
    } finally {
      client.close();
    }
  }
```


- Runner/Rules (same reason as for tomee)
  - 2 flavors:
    - Normal (per test or method depending how it is used)
    - Mono (single instance for the JVM)
  - Access to the configuration to read the config/ports etc even with random ports like on the slide
  - Customization hooks (config and Tomcat instance) / SPI
-

- jolokia to setup automatically /jolokia/* + /hawtio/
- CXF oauth2 server ready to run (bundle or classpath), jpa/jcache impls
- JTA (if needed for JMS/XA tx)
    - @Transactional, TransactionManager
- @Unit: jpa embedded API
    - Mainly to propose a new programming model (see later)

Extensions are mainly CDI integrations with potentially some Meecrowave configuration integration (--hawtio-active true). Role generally are:

- Consistent configuration with CLI/builder
- Manage some new components automatically

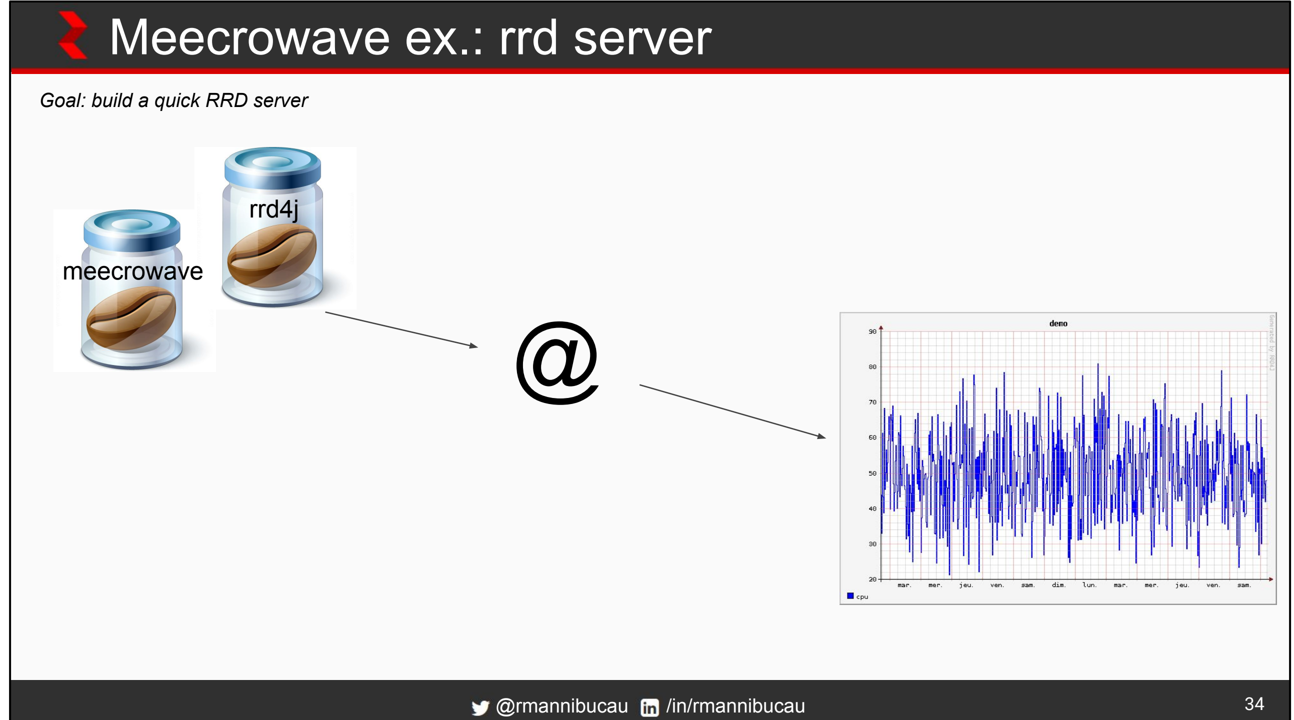
Meecrowave: new CDI paradigm (JPA ex)

```
@ApplicationScoped
public static class DataSourceConfig {
  @Produces
  @ApplicationScoped
  public DataSource dataSource() {
     final BasicDataSource source = new BasicDataSource();
     source.setDriver(new Driver());
     source.setUrl("jdbc:h2:mem:apachecon");
     return source;
  }
}

@ApplicationScoped
public class JpaConfig {
  @Produces
  public PersistenceUnitInfoBuilder unit(final DataSource ds) {
     return new PersistenceUnitInfoBuilder()
           .setUnitName("apachecon")
           .setDataSource(ds)
           .setExcludeUnlistedClasses(true)
           .addManagedClazz(User.class)
           .addProperty("openjpa.jdbc.SynchronizeMappings", "buildSchema");
  }
}
```

```
@ApplicationScoped
public class JPADao {
  @Inject
  @Unit(name = "apachecon")
  private EntityManager em;

  // tx by default
  public User save(final User user) {
     em.persist(user);
     return user;
  }

  @Jpa(transactional = false) // no tx
  public User find(final long id) {
     return em.find(User.class, id);
  }
}
```


- Configuration is owned by the application which will wire it
  - Configuration as a contract (vs container config)
    - -new thing to do
    - +uniform contract (for app + resources)
    - +no limitation due to the container, fully customizable and under the application control
- Ex for JPA:
  1. You configure the datasource somewhere
  2. You configure with meecrowave-jpa API the persistence unit getting the datasource injected (loose coupling)
  3. The meecrowave-jpa extension captures the unit builder and create entity manager factories for each of them and enable to then use JPA directly in services/resources

This is a new paradigm in EE inspired from Spring one which fits more and more applications and is not constrained by EE rules and lifecycle.



- Small example to show you how we develop with meecrowave
  - Import dependencies (meecrowave and rrd4j there)
  - Wrap rrd4j behind http (“business impl”)
  - (optional) add some meecrowave options if you need cli/builder control - or use another mecanism
  - Rrd4j generates graphs for you (service is done)

Meecrowave ex.: rrd server / code

Defaults / Build a rrd4j graph spec / Fill graph data / Respond with the picture

```java
@GET
@Path("{db}/graph")
public Response graphs(@PathParam("db") String name,
                       @QueryParam("fun") @DefaultValue("AVERAGE") String fun,
                       @QueryParam("from") long from,
                       @QueryParam("to") long to) {
    // Defaults
    final RrdDb rrdDb = getDb(name, true);
    if (to == 0) to = TimeUnit.MILLISECONDS.toSeconds(System.currentTimeMillis());
    if (from == 0) from = to - TimeUnit.DAYS.toSeconds(7);

    // Build a rrd4j graph spec
    final RrdGraphDef gDef = new RrdGraphDef();
    gDef.setWidth(600);
    gDef.setHeight(400);
    gDef.setStartTime(from);
    gDef.setEndTime(to);
    gDef.setTitle(name);
    gDef.setImageFormat("png");

    // Fill graph data
    final long[] interval = new long[]{from, to};
    IntStream.range(0, rrdDb.getDsCount()).forEach(it -> {
        final Datasource ds = rrdDb.getDatasource(it);
        try {
            gDef.datasource(ds.getName(), rrdDb.createFetchRequest(ConsolFun.AVERAGE, interval[0], interval[1]).fetchData());
            gDef.line(ds.getName(), Color.BLUE, ds.getName());
        } catch (final IOException e) {
            throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
        }
    });

    // Respond with the picture
    try {
        final byte[] payload = Optional.ofNullable(new RrdGraph(gDef).getRrdGraphInfo())
                .map(RrdGraphInfo::getBytes)
                .orElseGet(() -> "No data".getBytes(StandardCharsets.UTF_8));
        return Response.ok(payload)
                .header("Content-Type", "image/png")
                .build();
    } catch (final IOException e) {
        throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
    }
}
```


- 175 lines (with imports) for the endpoint, other is just DTO mapping rrd4j!
  - Create RrdDB and exploit them (add data and get graphs)
- No technical code, all is ready! (provided by the backbone)
  - No serialization
  - No routing
  - No parameter parsing
  - And we get the graph we saw previously

=> huge win for the productivity!

Meecrowave for batchs/daemons?

```java
@Named
@Dependent
public class Task extends AbstractBatchlet {
  @Override
  public String process() throws Exception {
     return "demo :)";
  }
}
```

```xml
<job xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       version="1.0" id="demo">
 <step id="start">
   <batchlet ref="task" />
 </step>
</job>
```

```xml
<dependencies>
 <dependency>
   <groupId>org.apache.geronimo.specs</groupId>
   <artifactId>geronimo-jbatch_1.0_spec</artifactId>
   <version>1.0</version>
 </dependency>
 <dependency>
   <groupId>org.apache.batchee</groupId>
   <artifactId>batchee-jbatch</artifactId>
   <version>0.4-incubating</version>
 </dependency>
 <dependency>
   <groupId>org.apache.meecrowave</groupId>
   <artifactId>meecrowave-core</artifactId>
   <version>0.3.1</version>
 </dependency>
</dependencies>
```


- Ok so meecrowave is great for web but what about daemons and batches?
  - Tomcat now supports a no connector mode meecroawve respects so it can be used!
    - Gain: same programming model, same lifecycle, same tooling (until it uses http), same habits
  - So a jbatch short life program would follow the same process as our rrd4j example:
    - Add dependencies
    - Write the business code (the batchlet + batch.xml)
      - Side note: batchlet should impl stop() if possible, not that useful here ;)
    - Some glue to run and wait the batch end (next slide)

=> No new main(), no new stack, no new monitoring (even if JBatch adds its own metrics/monitoring data related to the batch not the process, etc..)

Meecrowave short life softs, jbatch ex

```java
@ApplicationScoped
public class BatchExecutor {
  private long taskId;
  private CountDownLatch latch = new CountDownLatch(1);
  private JobOperator jobOperator;

  public void launchAndWait(@Observes @Initialized(ApplicationScoped.class) final ServletContext context) {
    jobOperator = BatchRuntime.getJobOperator();
    taskId = jobOperator.start("demo", new Properties());
    latch.countDown();
  }

  public void waitBatchResult() throws Exception {
    latch.await(5, TimeUnit.MINUTES);

    // portable wait
    while (!ofNullable(jobOperator.getJobExecution(taskId))
          .map(JobExecution::getBatchStatus)
          .map(Enum::name)
          .filter(s -> s.endsWith("ED")).isPresent()) {
      sleep(1000);
    }

    /* batchee wait
    JobExecutionCallbackService callbackService = ServicesManager.find().service(JobExecutionCallbackService.class);
    callbackService.waitFor(taskId);
    */

    dumpExecution(jobOperator.getJobExecution(taskId));
    jobOperator.getStepExecutions(taskId).forEach(this::dump);
  }
}
```

```java
public static void main(String[] args) throws Exception {
    new Meecrowave(new Meecrowave.Builder() {{ setSkipHttp(true); }}).bake();
    lookup(CDI.current().getBeanManager()).waitBatchResult();
}
```


- When JBatch starts a job it doesn't wait for it, designed for long life programs (like a EE server)
  - Concretely it ~starts a thread
- The component responsible to wait for the end will be a normal CDI bean
  - When the application starts it will start our job -> standard @initialized from CDI 1.1, ie built in startup support/hook
  - Then it will poll the execution status of the job and log the result if needed
- To make it working we'll write a small main starting meecrowave, then instead of waiting for meecrowave end we'll wait for the job end with our custom strategy
  - Don't forget to disable connectors if not needed (setSkipHttp(true))
  - CDI provides through CDI.current() a way to look up this custom strategy -> thanks classpath deployment (same loader)

And here we are :). Way harder to do it with a standalone server.

DeltaSpike the Meecrowave friend

```java
@ApplicationScoped
public class SomeService {
  @Inject
  private EntityManager entityManager;

  public DataSourcePayload findById(final String id) {
    return entityManager.find(SomeEntity.class, id);
  }

  @Transactional
  public SomeEntity create(final SomeEntity entity) {
    entityManager.persist(entity);
    entityManager.flush();
    return entity;
  }
```

```java
@Repository // @MappingConfig
public interface MyRepository extends
    EntityRepository<MyEntity, String> {
}

@ApplicationScoped
public class MyService {
  @Inject
  private MyRepository repository;

  public MyEntity find(String id) {
    return repository.findBy(id);
  }

  public List<MyEntity> findAll(int start, String id) {
    return repository.findAll(start, 10);
  }
}
```


Like Spring stack, EE stack passes by DeltaSpike, will bring

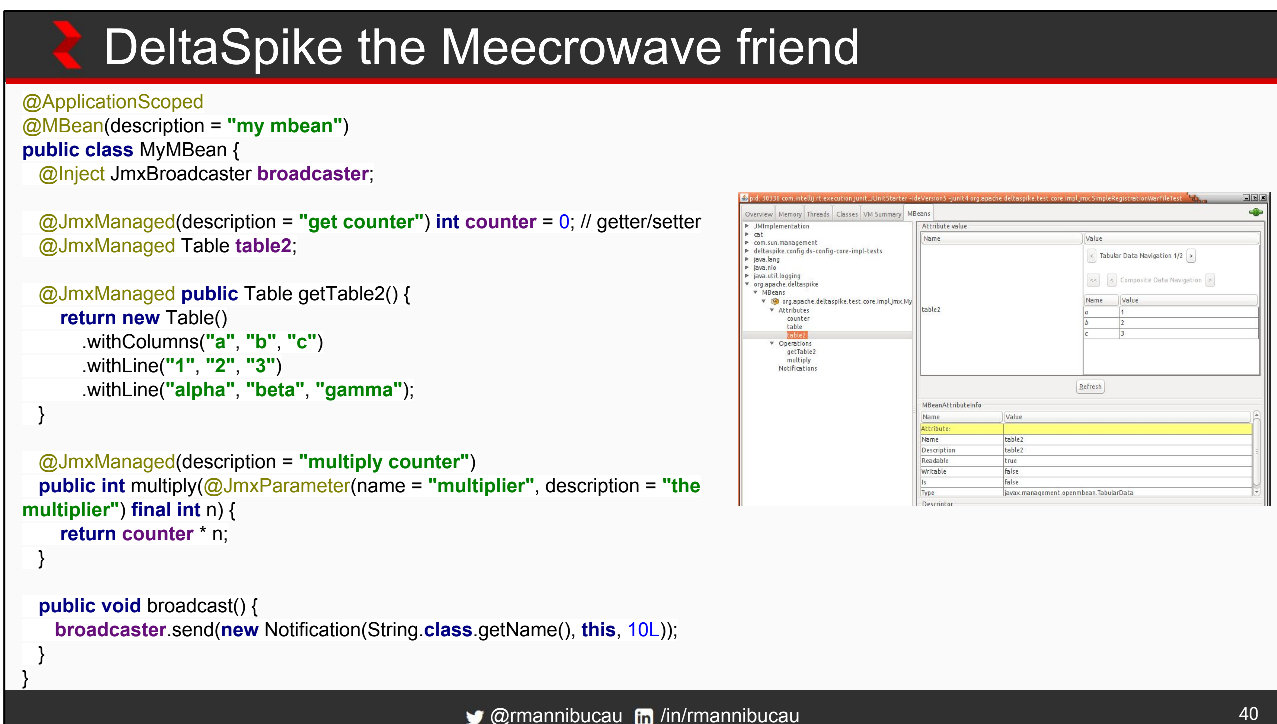
- MBean (ex later)
- Config (ex later)
- JPA
- Data (supports DTO directly)
- …

Often said Meecrowave was mainly missing JPA/Tx integration but DS fills this gap

DeltaSpike the Meecrowave friend

```java
@Configuration(prefix = "app.database.")
public interface AppConfiguration {
  @ConfigProperty(name = "url", defaultValue = "jdbc:mysql://localhost:3306/app")
  String databaseUrl();

  @ConfigProperty(name = "driver", defaultValue = "com.mysql.cj.jdbc.Driver")
  String databaseDriver();

  @ConfigProperty(name = "validation.query")
  String databaseValidationQuery();

  @ConfigProperty(name = "validation.interval", defaultValue = "-1")
  long databaseValidationInterval();

  @ConfigProperty(name = "username", defaultValue = "user")
  String databaseUser();

  @ConfigProperty(name = "password", defaultValue = "pass")
  String databasePassword();

  @ConfigProperty(name = "idle.min", defaultValue = "5")
  int databaseMinIdle();

  @ConfigProperty(name = "idle.max", defaultValue = "256")
  int databaseMaxTotal();
}
```


Either property injection or Proxy based API
Cache support for perf
Pluggable through ConfigSources + ordinal to sort them
Default support of system props/env



Bring any CDI bean to JMX

- Attributes
- Methods
- Notifications

Injectable in any other beans (easier state sharing)

TomEE or Meecrowave?

Which server?

Criteria	TomEE	Meecrowave
Stack	*Standard* (JPA, JTA, JAX-*S, …)	*Web, DS* or *Custom* (NoSQL, ...)
Front	JSF, any	Angular/React, Vaadin, ...
Library support	full EE	Servlet, JAX-RS, JSON*, CDI
Transactions (XA)	Yes	~No*
Resource definition	External (dev vs ops)	Internal (dev = ops)
Customization	8/10	9/10
Team responsibilities	9/10	7/10

- Resource definition: datasources, queues etc…
    - TomEE allows a lot of overriding
    - Meecrowave
- Stack
    - TomEE comes with more spec so if you need > web and not a custom setup (not requiring EE) then TomEE is smoother
        - Useful for dependency security scans and upgrades in companies
    - If the stack is huge meecrowave will have less conflict and you can control the classpath if there is some issues, harder with tomee
- If the front is JSF TomEE is well integrated (bval, jpa) otherwise both are good candidates
- Library support: not very different since CDI is now the backbone of EE and Meecrowave got the most used part (servlet, JAX-RS, CDI)
- Distibuted transactions are supported by TomEE but not Meecrowave OOTB (needs custom work)
- If intended to build a custom distribution, TomEE fits better than meecrowave but if intended to build a custom embedded server meecrowave is easier and more powerful to customize
    - Logging (log4j2)
    - Customizers

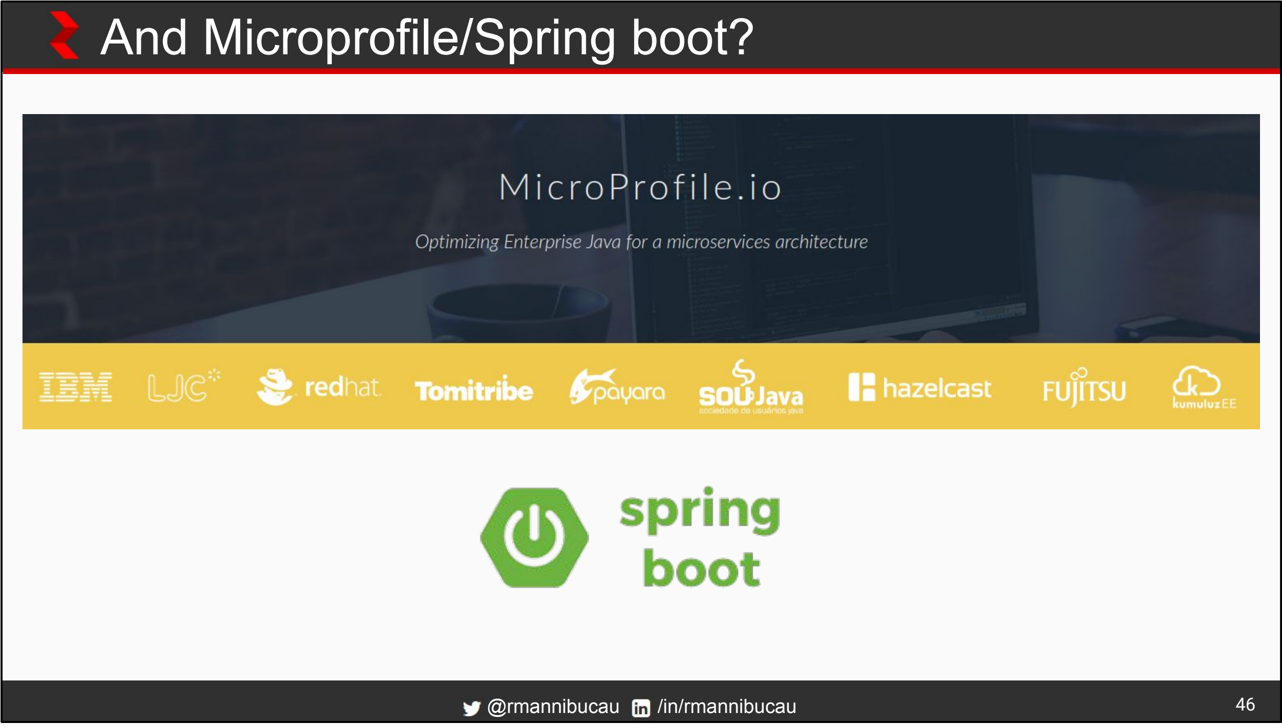
    - No assumptions on names
    - No JNDI constraint
    - Easier to tune (custom dbcp pool etc)
    - ….
    - However for “standard” distribution meecrowave doesn’t support yet win (exploded mode)
    - Meecrowave embedded mode makes Dockerization easy! Very close of a lib vs a container for tomee (not embedded)
- Team responsability: goal is to split the container from the app, this exclude the embedded mode
    - Tomee and meecrowave being tomcat based support it…
    - ...but meecrowave bundling doesnt encourage it more than that, you will need to grab the jar, remove it from the bundle etc...vs tomee which is ready OOTB as tomcat distrib

So which one?

The beast or the beauty?

- Meecrowave being inspired from tomee embedded it takes some goodness from it but enabling “container as lib” opens door TomEE can’t by its stack constraint
- It is also faster and without “EE” constraints or flag requirement
- On another side TomEE is generally a better fit for companies cause more common to scan (libs, CVE) and closer to Tomcat in term of deployment
- Main difference on an everyday basis is
  - Meecrowave stack is consistent (deployment/testing) vs TomEE has 5-6 modes
  - Meecrowave is a bit faster
  - But TomEE has JPA and BVal OOTB!

Microprofile?



- After Full -> Web Profile some company joined to create an even smaller distribution
    - Not mandatory but embedded oriented
    - TomEE Embedded, Payara micro, wildfly swarm, …
    - Meecrowave matches these criteria but doesn't target it yet (there is no real spec :D)
- Spring boot:
    - Reversed paradigm: you define your container in the code (vs) code is launched with the container
    - … close in practise (flat classpath usage, custom stacks) but more web oriented. CDI Extension ~ Actuator. PIck the ecosystem you prefer :)

Concretely choice can list a lot of concrete criterias but will always be done on feelings (jboss experience is different from “ASF” one, same with spring, in term of feature nothing is not possible or really hard to fill) so pick one!

@rmannibucau | https://blog-rmannibucau.rhcloud.com/