# Easy Java Integration Testing with Testcontainers

## Simplifying Integration Tests for Enterprise Java

🎙 Fabio Turizo

# Speaking Today

**Fabio Turizo**

Service Manager

**Bluesky:** @fturizo.bsky.social

**LinkedIn:** www.linkedin.com/in/fturizo/

# Give me Feedback!

Scan the QR to share comments on the talk

# Payara Services Helps Shape the Future of the Industry

- Strategic Members of the Eclipse Foundation
- Project Management Committee member of Jakarta EE

JAKARTA EE COMPATIBLE

eclipse SOLUTIONS MEMBERS

MICROPROFILE™ OPTIMIZING ENTERPRISE JAVA

# Payara Platform Enterprise

**Payara Server Enterprise**
Robust. Reliable. Supported.

The best application platform for production Jakarta EE apps.

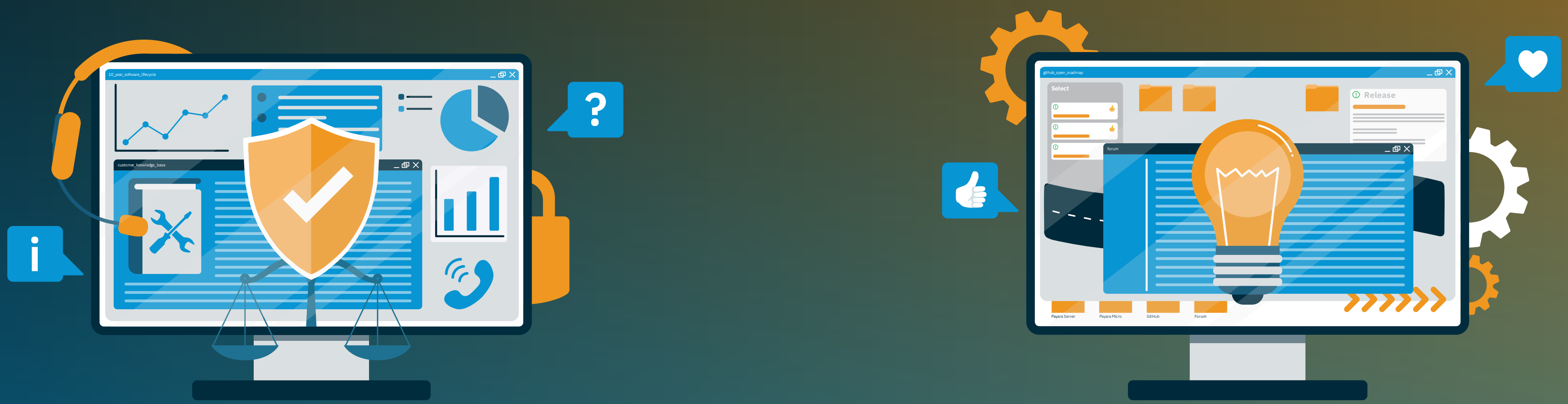
**Payara Micro Enterprise**
Small. Simple. Serious.

The platform of choice for containerized Jakarta EE microservices deployments.

JAKARTA EE COMPATIBLE

MICROPROFILE™
OPTIMIZING ENTERPRISE JAVA

# Payara Enterprise vs Payara Community



## Enterprise

- Built for the needs of production environments
- Automated with focus on scalability & availability
- Focus on stability with 10-year software lifecycle
- Security alerts and patches for ‘regulatory compliance’ & quality assurance
- Migration & Project Support, 24x7, or 10x5 support options
- Backwards compatibility

## Community

- Built for the needs of development environments
- Focus on performance over scalability & availability
- Focus on leading edge innovation
- Security issues dealt with at next release
- Community driven
- Manual focus rather than automated
- No guarantee of backwards compatibility or software lifecycle

# Integration Testing Basics

- A **level above** basic unit testing
- Multiple units come together to be tested
- Quickly test for regressions on newer versions
- Expose flaws in interface design
  - Guarantee platform updates
  - Test system dependencies

# Integration Testing - Java

- Simple UT frameworks
  - Junit, TestNG, Spock
- Highly dependable of the platform
- What about **Mocking** ?
  - Good to simulate interactions
  - Bad for real-time scenarios

# Integration Testing - Java

- *“Works in my machine”* persists 🤦
- Lots of challenges for Enterprise Java
  - Jakarta EE lack proper tools (Arquillian helps)
  - Spring Test is useful but not comprehensive
  - … But complex environments have lots of dependencies 🤔
- What about cloud-native applications?

# Arquillian Framework

- Focus on testing Jakarta EE components
  - Vendor-agnostic !
- Portable(*) Shrink-wrapped tests
- Container-specific adapters need to be written
- Added complexity in writing tests

# Arquillian Sample (JUnit 4.x)

```
@RunWith(Arquillian.class)
public class PersonDaoTest {

    @EJB
    private PersonDao personDao;

    @Inject TestData testData;

    @Deployment
    public static WebArchive createDeployment() {
        return ShrinkWrap.create(WebArchive.class, "arquillian-example.war")
                .addClass(Person.class)
                .addClass(PersonDao.class)
                .addAsResource("test-persistence.xml", "META-INF/persistence.xml");
    }
```

# Arquillian Sample (JUnit 4.x)

```
@Before
public void prepareTestData() {
    testData.prepareForShouldReturnAllPerson();
}

@Test
public void shouldReturnAllPerson() throws Exception {
    List<Person> personList = personDao.getAll();

    assertNotNull(personList);
    assertThat(personList.size(), is(1));
    assertThat(personList.get(0).getName(), is("John"));
    assertThat(personList.get(0).getLastName(), is("Malkovich"));
}
```

# Arquillian Sample (JUnit 4.x)

```
    @Dependent
    public static class TestData {

        @PersistenceContext
        private EntityManager entityManager;

        @Transactional
        public void prepareForShouldReturnAllPerson() {
            entityManager.persist(new Person("John", "Malkovich"));
        }
    }
}
```

# Spring Integration Testing

- Focus on JDBC testing and Container IoC capabilities
- Quickly test IoC container caching and DI features
- `TestContext` setup is highly customizable, but has a high learning curve.
- Some dependencies cannot be black-boxed

# Spring Test Sample

```
@SpringBootTest
public class PersonDaoTest {

    @Autowired
    private PersonDao personDao;

    @Autowired JdbcTemplate jdbcTemplate;

    @BeforeAll
    public static void setupData() {
        jdbcTemplate.execute("CREATE TABLE person (id INT PRIMARY KEY, name VARCHAR(255), last_name
VARCHAR(255)");
          jdbcTemplate.execute("INSERT INTO person (id, name, last_name) VALUES (1, 'John’, ‘Doe')");
          jdbcTemplate.execute("INSERT INTO person (id, name, last_name) VALUES (2, 'Jane’, ‘Smith')")
    }
```

# Spring Test Sample

```
@Test
public class testPersonCount {
        int count = personDao.getPersonCount();
        assertThat(count).isEqualTo(2);
}


@Test
public class testPersonCount {
        String name = personDao.getPersonName(1);
        assertThat(name).isEqualTo("John");
}
```

# Some Gaps to be Filled 😖

- Test issues:
  - Assemble a _*part of the application*_ to test it
  - This includes code and resources
  - The persistence layer is tested, but not the store
  - No way to easily test different configurations
- Sadly, not real-world tests.

# Enter Testcontainers!

- Dependencies as code
- Based on Docker containers
  - Ease up networking setup
- Data access layer tests support
- Fully* portable integration tests!

# Testcontainers Benefits

- Effective black-box testing
- Tests are user-focused
- Run UA testing with little overhead
- Resource management is automated
- API bindings for Java, Ruby, Rust, Go, Python, etc.

payara®

# Testcontainers Requirements

- Docker (only Linux containers)
- Test Frameworks
  - Junit 4
  - Junit 5
  - Spock
- Maven/Gradle Dependencies

# Getting Started – JUnit5

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.11.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.11.3</version>
    <scope>test</scope>
</dependency>
```

# Getting Started – TC + JUnit5

```
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>testcontainers</artifactId>
        <version>1.20.4</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>1.20.4</version>
        <scope>test</scope>
</dependency>
```

# Testcontainer Setup

```
@TestContainers
public class BasicApplicationTest{
        @Container
        GenericContainer myContainer = new
        GenericContainer(DockerImageName.parse(“fturizo/myapp"))
                        .withExposedPorts(8080, 9009, 28080)
                        .withCommand(“./deploy-application.sh”);

        @Test
        public void test_running(){
                assert(isTrue(myappcontainer.isRunning()));
        }
}
```

# Container Access - Boundary

```
@Test
public void test_application(){
      String url = String.format(“http://%s:%s/%s”,
                   myContainer.getHost(),
                   myContainer.getMappedPort(8080), “/myapp”);
      int status = http.get(url).response().status();
      //Careful!
      assertEquals(status, 200);
}
```

# Waiting for Readiness - HTTP

```
@TestContainers
public class BasicApplicationTest{
      @Container
      GenericContainer myContainer = new
      GenericContainer(DockerImageName.parse(“fturizo/myapp"))
                 .withExposedPorts(8080, 9009, 28080)
                 .waitingFor(Wait.forHttp(“/myapp/ready”)
                                  .forStatusCode(200));

}
```

# Waiting for Readiness – Log Message

```
@TestContainers
public class BasicApplicationTest{
      @Container
      GenericContainer myappContainer = new
      GenericContainer(DockerImageName.parse(“fturizo/myapp"))
                 .withExposedPorts(8080, 9009, 28080)
                 .waitingFor(Wait.forLogMessage(“.*Application
                                                 is ready.*”);
}
```

# Dependency Configuration (1)

```
Network network = Network.newNetwork();

@Container
GenericContainer dbContainer = new
      GenericContainer(DockerImageName.parse(“mysql:8.0"))
                  .withEnv(“MYSQL_ROOT_PASSWORD”, “rootPass”)
                  .withEnv(“MYSQL_USER”, “test”)
                  .withEnv(“MYSQL_PASSWORD”, “test”)
                  .withEnv(“MYSQL_DATABASE”, “testDB”)
                  .withNetwork(network)
                  .withNetworkAlias(“mysql_db”);
```

# Dependency Configuration (2)

```
@Container
GenericContainer appContainer = new
     GenericContainer(DockerImageName.parse(“fturizo/myapp"))
               .withEnv(“DB_SERVER”, “mysql_db”)
               .withEnv(“DB_USER”, “test”)
               .withEnv(“DB_PASSWORD”, “test”)
               .withEnv(“DB_NAME”, “testDB”)
               .withNetwork(network)
               .dependsOn(dbContainer)
               …
```

# Database Support

- Special objects for wrapped containers
- Popular market choices for:
  - Relational: MySQL, MariaDB, OracleXE, DB2, Postgres
  - NoSQL: Couchbase, MongoDB, Neo4J, Cassandra, OrientDB
- Easy instantiation and integration

# MySQL Database Configuration

```xml
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>mysql</artifactId>
        <version>1.20.4</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>8.3.0</version>
        <scope>test</scope>
</dependency>
```

# MySQL Testcontainer (1)

```
@Container
MySQLContainer mysqlContainer = new
     MySQLContainer<>(DockerImageName.parse(“mysql:8.0”))
               .withNetwork(network)
               .withNetworkAliases(“mysql_db”);
```

# MySQL Testcontainer (2)

```
@Container
GenericContainer appContainer = new
      GenericContainer(DockerImageName.parse(“fturizo/myapp"))
                  .withEnv(“DB_SERVER”, “mysql_db”)
                  .withEnv(“DB_USER”, mysqlContainer.getUser())
                  .withEnv(“DB_PASSWORD”,
                              mysqlContainer.getPassword())
                  .withEnv(“DB_NAME”,
                              mysqlContainer.getDatabaseName())
                  .withNetwork(network)
                  .dependsOn(mysqlContainer)
                  …
```

# MySQL Testcontainer (3)

```
String query = "select * from …”;
try(Connection connection =
       DriverManager.getConnection(mysqlContainer.getJdbcUrl(),
                                   mysqlContainer.getUsername(),
                                   mysqlContainer.getPassword());
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery(query))){
       while(resultSet.next()) {
          assertThat(resultSet.get(0), isEqual(“XYZ”));
       }
} catch (SQLException e) {
       assert false;
}
```

# More Features!

- Docker Compose is supported, too:

```
DockerComposeContainer environment =
    new DockerComposeContainer(new
              File(“src/test/compose.yaml”))
          .withExposedService(“mysql_1”, 3306)
          .withExposedService(“myapp_1”, 8080);
```

# More Features !

- Official modules for popular solutions:
  - ElasticSearch (Distributed Search)
  - Apache Kafka (Distributed Messaging)
  - RabbitMQ (JMS)
  - Solr (Text Search)
  - Nginx (Loadbalancing)

# Testcontainers Caveats

- Docker adds an extra layer of processing
  - More resources needed for full coverage
  - Test time will increase overall!
- Middleware must be prepared for Docker*
  - And so are its dependencies!
- Black-box testing is not suited for all software tests

# Demo Time

**https://github.com/fturizo/ConferenceDemo**

# Give me Feedback (again)!

Scan the QR to share comments on the talk

# Thank You

Please visit us at:

payara.fish/join-us