Users of TomEE 9.x onwards should be aware that, as a result of the move from Java EE to Jakarta EE as part of the transfer of Java EE to the Eclipse Foundation, the primary package for all implemented APIs has changed from javax.* to jakarta.*.

This will almost certainly require code changes to enable applications to migrate to TomEE 9.x and later.

TomEE 9.x onwards has an integrated functionality for automatically migrating J2EE / Java EE WARs to Jakarta EE.

How to Migrate from Java EE to Jakarta EE

With the transition from Java EE to Jakarta EE, many developers are looking to migrate their applications to the new platform. Jakarta EE, managed by the Eclipse Foundation, offers improved features and performance, making it a worthy upgrade for your projects.

Understand the Key Differences between Java EE and Jakarta EE

The transition from Java EE to Jakarta EE involves several key differences that are essential to grasp before initiating the migration process. The most significant change is the shift in the namespace from `javax.*` to `jakarta.*`. This alteration impacts all the Java EE APIs which are now consolidated under the Jakarta EE umbrella. Understanding this change is crucial as it affects every aspect of application development, from annotations and imports in your code to configuration files and dependencies.

Beyond the namespace changes, Jakarta EE also introduces improvements and modern approaches to enterprise Java, including improvements in cloud-native application development capabilities, updates to microservices architectures, and better integrations with modern CI/CD tools. It supports a more open and collaborative development environment compared to Java EE, which was primarily controlled by Oracle.

Prepare Your Environment

First, ensure that you have the necessary tools and software installed:

- JDK 8 or higher (Jakarta EE supports JDK 8 and 11)
- Jakarta EE compatible application server (tomEE)
- IDE with Jakarta EE support (e.g., Eclipse, IntelliJ IDEA)

Update Your Dependencies

Updating dependencies is a critical step in migrating to Jakarta EE as it ensures that your application utilizes the latest libraries, which are compliant with the new `jakarta.*` namespace. This process involves replacing all the Java EE dependencies with their respective Jakarta EE counterparts in your project's build configuration files, such as Maven or Gradle.

For Maven projects, modify the `pom.xml` to include the Jakarta EE libraries. Here's how you can update the dependencies in your Maven configuration:

```
<dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-api</artifactId>
    <version>9.1.0</version>
    <scope>provided</scope>
</dependency>
```

For Gradle projects, you will need to update your `build.gradle` file similarly. Ensure that you use the `providedCompile` scope to avoid including these APIs in the final artifact, as they are typically provided by the application server:

```
dependencies {
    providedCompile 'jakarta.platform:jakarta.jakartaee-api:9.1.0'
}
```

It's important to review all project dependencies for any indirect usage of old Java EE APIs and update them accordingly. This might require checking third-party libraries or frameworks that are being used in your project to ensure they have been updated to support Jakarta EE.

Refactor Your Code

Refactoring your code to use the new `jakarta.*` namespace is perhaps the most labor-intensive part of the migration. This step involves updating import statements and fully-qualified class names throughout your codebase.

1. **Manual Refactoring:** This can be done manually but is only recommended for smaller projects due to its error-prone nature. You would need to search and replace all occurrences of `javax.*` with `jakarta.*` in your source files.
2. **Automated Migration Tools:** For larger codebases, consider using an automated migration tool to streamline the process. The Eclipse Transformer, for example, is a comprehensive tool designed for this purpose. It can be executed as follows:

```
java -jar org.eclipse.transformer.cli-0.4.0-SNAPSHOT-all.jar \
  -i input_directory \
  -o output_directory \
  -x jakarta-renames.properties \
  -t jakarta-direct.properties
```

This tool scans all files in the input directory, transforms them according to the specified rename and direct properties, and outputs the transformed files into the output directory.

Test Your Application

Testing is a critical phase in the migration process to ensure that the application functions as expected after the transition to Jakarta EE. Here's how you can thoroughly test your application:

1. **Unit Testing:** Begin with unit tests to verify that individual components work correctly with the new `jakarta.*` namespaces. This involves testing all the classes and methods independently to ensure they perform as expected.

2. **Integration Testing:** After unit testing, perform integration tests. These tests are crucial as they help confirm that different parts of your application work together as intended. This includes testing interactions with databases, other services, and integration with middleware.

3. **System Testing:** Conduct system-wide tests to evaluate the application's performance and behavior in an environment that mimics production.

4. **Performance Testing:** It's also essential to test how the application performs under load. Tools like Apache JMeter can be used to simulate multiple requests to the application and measure response times and throughput.

5. **Regression Testing:** Ensure that features that were working before the migration are still operational. This helps in identifying any new issues that might have been introduced during the namespace change.

6. **User Acceptance Testing (UAT):** Finally, involve end-users to perform UAT. This step is vital to confirm that the application meets the business requirements and is ready for deployment.

Deploy Your Application to the Jakarta EE Compatible Server

Deploying your application on a Jakarta EE-compatible server is the next step after testing. Here are the steps involved:

1. **Choose a Compatible Server:** First, ensure that your chosen server supports Jakarta EE.
2. **Server Configuration:** Configure the server according to the needs of your application. This might include setting up data sources, JMS providers, and integrating any required external systems.
3. **Deployment:** Deploy your application to the server. This can typically be done through the server's administration console, command line tools, or through an integrated development environment (IDE) that supports deployment.
4. **Verify Deployment:** After deployment, verify that the application starts up correctly and that there are no startup errors in the server logs.
5. **Smoke Testing:** Perform a quick round of tests to ensure that all major functionalities are working as expected.

Monitor and Optimize Your Application

Once your application is deployed, ongoing monitoring and optimization are crucial to maintain and improve its performance:

1. **Monitoring Tools:** Utilize monitoring tools to continuously observe the application's performance. Tools like Prometheus, Grafana, or the monitoring capabilities of your application server can provide insights into throughput, memory usage, response times, and more.

2. **Log Analysis:** Keep an eye on server logs for any errors or unusual activity. Log management tools can help in aggregating and analyzing logs to detect potential issues early.

3. **Performance Tuning:** Based on the monitoring data, adjust configurations to optimize performance. This might involve tuning JVM settings, optimizing database queries, and scaling the application as needed.

4. **Feedback Loop:** Establish a feedback loop where performance insights and user feedback lead to further refinements and adjustments in the application.

5. **Security Updates:** Regularly update your application and server to patch any security vulnerabilities.

Migrating from IBM WebSphere Application Server to Apache TomEE

Migrating from IBM WebSphere Application Server to Apache TomEE involves several steps. Here's a structured approach to facilitate the migration:

1. Assessment and Planning

- **Inventory Applications**: List all applications running on WebSphere, including their dependencies.
- **Identify Features**: Determine which WebSphere features are used (e.g., JMS, EJB, JPA) and find their equivalents in TomEE.
- **Compatibility Check**: Review Java EE specifications and ensure that your applications comply with those supported by TomEE.

2. Environment Setup

- **Install TomEE**: Download and install the appropriate version of TomEE from the Apache TomEE website.
- **Configure Server**: Set up the server configuration, including data sources, JNDI resources, and other necessary settings.

3. Code Modifications

- **JNDI Changes**: Update JNDI resource lookups to match TomEE configurations.
- **Dependency Updates**: Replace any WebSphere-specific libraries with their TomEE counterparts. Ensure that you use compatible versions of Java EE APIs.
- **Configuration Files**: Convert `web.xml`, `ejb-jar.xml`, and other configuration files to match TomEE's structure.

4. Testing

- **Unit Testing**: Run unit tests to ensure that individual components function correctly after migration.
- **Integration Testing**: Conduct integration tests to verify that the application works as expected within the TomEE environment.
- **Performance Testing**: Benchmark performance and resource usage to identify any potential issues.

5. Deployment

- **Package Applications**: Create WAR or EAR files as needed for deployment in TomEE.
- **Deploy to TomEE**: Use the TomEE management console or directly place the packaged files in the `webapps` directory.
- **Monitor Logs**: Check logs for any errors or warnings during startup and operation.

6. Post-Migration

- **Monitoring**: Set up monitoring to track application performance and stability in the new environment.
- **Documentation**: Update documentation to reflect changes made during migration.
- **Training**: Provide training for your team on TomEE features and management.

7. Rollback Plan

- **Prepare for Rollback**: Ensure you have a rollback strategy in case issues arise post-migration. Keep backups of the WebSphere environment until you confirm the stability of the TomEE setup.

  Here’s a simplified example of migrating a Java EE application from IBM WebSphere Application Server (WAS) to Apache TomEE. This example focuses on a sample web application that uses EJB and JPA.

Example Application: Simple E-Commerce Application

1. Assessment

- **Current Setup**: The application uses EJB for business logic and JPA for database access, deployed on WAS.
- **Dependencies**: WebSphere libraries for EJB and JPA.

2. Environment Setup

- **Install TomEE**: Download and install TomEE (e.g., TomEE Plus).
- **Configure Data Source**: Configure a data source in `tomee.xml`:

```
<Resources>
    <Resource name="jdbc/myDataSource"
              auth="Container"
              type="javax.sql.DataSource"
              driverClassName="org.h2.Driver"
              url="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"
              username="sa"
              password=""/>
</Resources>
```

3. Code Modifications

- **Update JNDI Lookups**: Change JNDI lookups in your code. For example, in WAS, you might have:

```
@EJB
private MyService myService;
```

In TomEE, you may need to ensure the correct annotations are used, but the code remains largely the same.

- **Modify `persistence.xml`**: Update the `persistence.xml` to match TomEE's configuration:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
    <persistence-unit name="myPU">
        <jta-data-source>jdbc/myDataSource</jta-data-source>
        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

4. Testing

- **Unit Testing**: Run unit tests to ensure business logic works correctly.

- **Integration Testing**: Test the application in a local TomEE instance to validate that all components interact as expected.

5. Deployment

- **Package Application**: Create a WAR file of your application.
- **Deploy to TomEE**: Place the WAR file in the `webapps` directory of TomEE.

6. Post-Migration

- **Monitor Logs**: Check `catalina.out` and application-specific logs for any errors during startup or operation.
- **Performance Testing**: Conduct performance tests to ensure the application meets requirements.

Sample Code Snippet

Here's a simple EJB and a JPA entity for the application:

EJB Example:

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class ProductService {
    @PersistenceContext(unitName = "myPU")
    private EntityManager em;

    public void addProduct(Product product) {
        em.persist(product);
    }
}
```

JPA Entity Example:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Product {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private double price;

    // Getters and Setters
```

```
}
```