

Migrate Spring Boot applications to Azure Container Apps

Article • 09/30/2024

This guide describes what you should be aware of when you want to migrate an existing Spring Boot application to run on Azure Container Apps.

Pre-migration

To ensure a successful migration, before you start, complete the assessment and inventory steps described in the following sections.

If you can't meet any of these pre-migration requirements, see the following companion migration guides:

- Migrate executable JAR applications to containers on Azure Kubernetes Service (guidance planned)
- Migrate executable JAR Applications to Azure Virtual Machines (guidance planned)


Inspect application components

Identify local state

On PaaS environments, no application is guaranteed to be running exactly once at any given time. Even if you configure an application to run in a single instance, a duplicate instance can be created in the following cases:

- The application must be relocated to a physical host due to failure or system update.
- The application is being updated.

In any of these cases, the original instance remains running until the new instance has finished starting up. This pattern can have the following potentially significant implications for your application:

- No [singleton](#)  can be guaranteed to be truly single.
- Any data not persisted to outside storage will likely be lost sooner than it would be on a single physical server or VM.

Before migrating to Azure Container Apps, ensure that your code doesn't contain local state that must not be lost or duplicated. If local state exists, change the code to store that state outside the application. Cloud-ready applications typically store application state in locations such as the following options:

- [Azure Cache for Redis](#)
- [Azure Cosmos DB](#)
- Another external database, such as [Azure SQL](#), [Azure Database for MySQL](#), or [Azure Database for PostgreSQL](#).
- [Azure Storage](#), used for storing unstructured data or even serialized objects.

Determine whether and how the file system is used

Find any instances where your services write to and/or read from the local file system. Identify where short-term/temporary files are written and read and where long-lived files are written and read.

Azure Container Apps offers several types of storage. Ephemeral storage can read and write temporary data and be available to a running container or replica. Azure File provides permanent storage and can be shared across multiple containers. For more information, see [Use storage mounts in Azure Container Apps](#).

Read-only static content

If your application currently serves static content, you need an alternate location for it. You might wish to consider moving static content to Azure Blob Storage and adding Azure CDN for lightning-fast downloads globally. For more information, see [Static website hosting in Azure Storage](#) and [Quickstart: Integrate an Azure storage account with Azure CDN](#).

Dynamically published static content

If your application supports static content, whether uploaded or generated by the application itself, that remains unchanged after its creation, you can integrate Azure Blob Storage and Azure CDN. You can also use an Azure Function to manage uploads and trigger CDN refreshes when necessary. We've provided a sample implementation for your use at [Uploading and CDN-preloading static content with Azure Functions](#)^[1].

Determine whether any of the services contain OS-specific code

If your application contains any code with dependencies on the host OS, then you need to refactor it to remove those dependencies. For example, you may need to replace any use of `/` or `\` in file system paths with [File.Separator](#) or [Paths.get](#) if your application is running on Windows.

Switch to a supported platform

If you create your Dockerfile manually and deploy containerized application to Azure Container Apps, you take full control over your deployment including JRE/JDK versions.

For deployment from artifacts, Azure Container Apps also offers specific versions of Java (8, 11, 17, and 21) and specific versions of Spring Boot and Spring Cloud components. To ensure compatibility, first migrate your application to one of the supported versions of Java in its current environment, then proceed with the remaining migration steps. Be sure to fully test the resulting configuration. Use the latest stable release of your Linux distribution in such tests.

ⓘ Note

This validation is especially important if your current server is running on an unsupported JDK (such as Oracle JDK or IBM OpenJ9).

To obtain your current Java version, sign in to your production server and run the following command:

```
Bash
```

```
java -version
```

For supported versions of Java, Spring Boot, and Spring Cloud, as well instructions for updating, see [Java on Azure Container Apps overview](#).

Determine whether your application relies on scheduled jobs

Ephemeral application such as Unix cron jobs or short-live applications based on Spring Batch framework should run as a job on Azure Container Apps. For more information, see [Jobs in Azure Container Apps](#). If your application is a long-running application and executes tasks regularly using a scheduling framework such as Quartz or Spring Batch, Azure Container Apps can host that application. However, the application must handle scaling appropriately to avoid race conditions where the same application instances are executed more than once per scheduled period during scale-out or rolling upgrade.

Inventory any scheduled tasks running on the production servers, inside or outside your application code.

Identify Spring Boot versions

Examine the dependencies of each application being migrated to determine its Spring Boot version.

Maven

In Maven projects, the Spring Boot version is typically found in the `<parent>` element of the POM file:

XML

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.3.3</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Gradle

In Gradle projects, the Spring Boot version will typically be found in the `plugins` section, as the version of the `org.springframework.boot` plugin:

Gradle

```
plugins {
  id 'org.springframework.boot' version '3.3.3'
  id 'io.spring.dependency-management' version '1.1.6'
  id 'java'
}
```

For any applications using Spring Boot versions prior to 3.x, follow the [Spring Boot 2.0 migration guide](#) or [Spring Boot 3.0 Migration Guide](#) to update them to a supported Spring Boot version. For supported versions, see the [Spring Boot and Spring Cloud versions](#).

Identify log aggregation solutions

Identify any log aggregation solutions in use by the applications you're migrating. You need to configure diagnostic settings in migration to make logged events available for consumption. For more information, see [Ensure console logging and configure diagnostic settings](#) section.

Identify application performance management (APM) agents

Identify any application performance management agents used by your applications. Azure Containers Apps doesn't offer built-in support for APM integration. You need to prepare your container image or integrate APM tool directly into your code. If you want to measure your application's performance but haven't integrated any APM yet, consider using Azure Application Insights. For more information, see the [Migration](#) section.

Inventory external resources

Identify external resources, such as data sources, JMS message brokers, and URLs of other services. In Spring Boot applications, you can typically find the configuration for such resources in the *src/main/resources* folder, in a file typically called *application.properties* or *application.yml*.

Databases

For a Spring Boot application, connection strings typically appear in configuration files when it depends on an external database. Here's an example from an *application.properties* file:

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/mysql_db
spring.datasource.username=dbuser
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Here's an example from an *application.yml* file:

YAML

```
spring:
  data:
    mongodb:
      uri: mongodb://mongouser:deepsecret@mongoserver.contoso.com:27017
```

See Spring Data documentation for more possible configuration scenarios:

- [JPA Repositories](#) ↗
- [JDBC Repositories](#) ↗
- [Cassandra Repositories](#) ↗
- [MongoDB Repositories](#) ↗

JMS message brokers

Identify the broker or brokers in use by looking in the build manifest (typically, a *pom.xml* or *build.gradle* file) for the relevant dependencies.

For example, a Spring Boot application using ActiveMQ would typically contain this dependency in its *pom.xml* file:

XML

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

Spring Boot applications using commercial brokers typically contain dependencies directly on the brokers' JMS driver libraries. Here's an example from a *build.gradle* file:

JSON

```
dependencies {
  ...
  compile("com.ibm.mq:com.ibm.mq.allclient:9.4.0.5")
  ...
}
```

After you've identified the broker or brokers in use, find the corresponding settings. In Spring Boot applications, you can typically find them in the *application.properties* and *application.yml* files in the application directory.

Here's an ActiveMQ example from an *application.properties* file:

properties

```
spring.activemq.brokerurl=broker:
(tcp://localhost:61616,network:static:tcp://remotehost:61616)?
persistent=false&useJmx=true
```

```
spring.activemq.user=admin
spring.activemq.password=tryandguess
```

For more information on ActiveMQ configuration, see the [Spring Boot messaging documentation](#).

Here's an IBM MQ example from an *application.yaml* file:

YAML

```
ibm:
  mq:
    queueManager: qm1
    channel: dev.ORDERS
    connName: localhost(14)
    user: admin
    password: big$ecr3t
```

For more information on IBM MQ configuration, see the [IBM MQ Spring components documentation](#).

Identify external caches

Identify any external caches in use. Frequently, Redis is used via Spring Data Redis. For configuration information, see the [Spring Data Redis](#) documentation.

Determine whether session data is being cached via [Spring Session](#) by searching for the respective configuration (in [Java](#) or [XML](#)).

Identity providers

Identify any identity provider(s) used by your application. For information on how identity providers may be configured, consult the following:

- For OAuth2 configuration, see the [Spring Security reference](#).
- For Auth0 Spring Security configuration, see the [Auth0 Spring Security documentation](#).
- For PingFederate Spring Security configuration, see the [Auth0 PingFederate instructions](#).

Identify any clients relying on a non-standard port

Azure Container Apps allows you to expose port according to your Azure Container Apps resource configuration. For instance, a Spring Boot application listens to port of

8080 by default, but it can be set with `server.port` or environment variable `SERVER_PORT` as you need.

All other external resources

It isn't feasible for this guide to document every possible external dependency. After the migration, it's your responsibility to verify that you can satisfy every external dependency of your application.

Inventory configuration sources and secrets

Inventory passwords and secure strings

Check all properties and configuration files and all environment variables on the production deployment(s) for any secret strings and passwords. In a Spring Boot application, you can typically find such strings in the *application.properties* or *application.yml* file.

Inventory certificates

Document all the certificates used for public SSL endpoints or communication with backend databases and other systems. You can view all certificates on the production server(s) by running the following command:

```
Bash
```

```
keytool -list -v -keystore <path to keystore>
```

Inspect the deployment architecture

Document hardware requirements for each service

Document the following information for your Spring Boot application:

- The number of instances running.
- The number of CPUs allocated to each instance.
- The amount of RAM allocated to each instance.

Document geo-replication/distribution

Determine whether your Spring Boot application instances are currently distributed among several regions or data centers. Document the uptime requirements/SLA for the applications you're migrating.

Migration

Create an Azure Container Apps environment and deploy apps

Provision an Azure Container Apps instance in your Azure subscription. Its secure hosting environment is created along with it. For more information, see [Quickstart: Deploy your first container app using the Azure portal](#).

Ensure console logging and configure diagnostic settings

Configure your logging to ensure that all output is routed to the console rather than to files.

After an application is deployed to Azure Container Apps, you can configure the logging options within your Container Apps environment to define one or more destinations of the logs. These destinations can include Azure Monitor Log Analytics, Azure Event hub, or even other third-party monitoring solutions. You also have the option to disable log data and view logs only at runtime. For detailed configuration instruction, see [Log storage and monitoring options in Azure Container Apps](#).

Configure persistent storage

If any part of your application reads or writes to the local file system, you need to configure persistent storage to replace the local file system. You can specify the path to mount in the container through the app settings and align it with the path your app is using. For more information, see [Use storage mounts in Azure Container Apps](#).

Migrate all certificates to KeyVault

Azure Containers Apps supports secure communication between apps. Your application doesn't need to manage the process of establishing secure communication. You can upload the private certificate to Azure Container Apps or use a free managed certificate provided by Azure Container Apps. Using Azure Key Vault to manage certificates is a recommended approach. For more information, see [Certificates in Azure Container Apps](#).

Configure application performance management (APM) integrations

Whether your app is deployed from a container image or from code, Azure Container Apps doesn't interfere with your image or code. Therefore, integrating your application with an APM tool depends on your own preferences and implementation.

If your application isn't using a supported APM, Azure Application Insights is one option. For more information, see [Using Azure Monitor Application Insights with Spring Boot](#).

Deploy the application

Deploy each of the migrated microservices (not including Spring Cloud Config Server and Spring Cloud Service Registry), as described in [Deploy Azure Container Apps with the `az containerapp up` command](#).

Configure per-service secrets and externalized settings

You can inject configuration settings into each application as environment variables. You can set these variables as manually entries or as references to secrets. For more information about configuration, see [Manage environment variables on Azure Container Apps](#).

Migrate and enable the identity provider

If any of the Spring Cloud applications require authentication or authorization, ensure they're configured to access the identity provider:

- If the identity provider is Microsoft Entra ID, no changes should be necessary.
- If the identity provider is an on-premises Active Directory forest, consider implementing a hybrid identity solution with Microsoft Entra ID. For more information, see the [Hybrid identity documentation](#).
- If the identity provider is another on-premises solution, such as PingFederate, consult the [Custom installation of Microsoft Entra Connect](#) topic to configure federation with Microsoft Entra ID. Alternatively, consider using Spring Security to use your identity provider through [OAuth2/OpenID Connect](#) or [SAML](#).

Expose the application

By default, an application deployed to Azure Container Apps is accessible via an application URL. If your app is deployed in the context of a managed environment with

its own virtual network, you need to determine the app's accessibility level to allow public ingress or ingress from your virtual network only. For more information, see [Networking in Azure Container Apps environment](#).

Post-migration

Now that you've completed your migration, verify that your application works as you expect. You can then make your application more cloud-native by using the following recommendations.

- Consider enabling your application to work with Spring Cloud Registry. This component enables your application to be dynamically discovered by other deployed Spring applications and clients. For more information, see [Configure settings for the Eureka Server for Spring component in Azure Container Apps](#). Then, modify any application clients to use the Spring Client Load Balancer. The Spring Client Load Balancer enables the client to obtain addresses of all the running instances of the application and find an instance that works if another instance becomes corrupted or unresponsive. For more information, see [Spring Tips: Spring Cloud Load Balancer](#) [↗](#) in the Spring Blog.
- Instead of making your application public, consider adding a [Spring Cloud Gateway](#) [↗](#) instance. Spring Cloud Gateway provides a single endpoint for all applications deployed in your Azure Container Apps environment. If a Spring Cloud Gateway is already deployed, ensure that a routing rule is configured to route traffic to your newly deployed application.
- Consider adding a Spring Cloud Config Server to centrally manage and version-control configuration for all your Spring Cloud applications. First, create a Git repository to house the configuration and configure app instance to use it. For more information, see [Configure settings for the Config Server for Spring component in Azure Container Apps](#). Then, migrate your configuration using the following steps:

1. Inside the application's *src/main/resources* directory, create a *bootstrap.yml* file with the following contents:

```
yml

spring:
  application:
    name: <your-application-name>
```

2. In the configuration Git repository, create a `<your-application-name>.yaml` file, where `your-application-name` is the same as in the preceding step. Move the settings from `application.yml` file in `src/main/resources` to the new file you created. If the settings were previously in a `.properties` file, converted them to YAML first. You can find online tools or IntelliJ plugins to perform this conversion.
 3. Create an `application.yml` file in the directory above. You can use this file to define settings and resources that are shared among all applications in the Azure Container Apps environment. Such settings typically include data sources, logging settings, Spring Boot Actuator configuration, and others.
 4. Commit and push these changes to the Git repository.
 5. Remove the `application.properties` or `application.yml` file from the application.
- Consider adding the Admin for Spring managed component to enable an administrative interface for Spring Boot web applications that expose actuator endpoints. For more information, see [Configure the Spring Boot Admin component in Azure Container Apps](#).
 - Consider adding a deployment pipeline for automatic, consistent deployments. Instructions are available for [Azure Pipelines](#) and for [GitHub Actions](#).
 - Consider using container apps revisions, revision labels, and ingress traffic weights to enable blue-green deployment, which enables you to test code changes in production before they're made available to some or all of your end users. For more information, see [Blue-Green Deployment in Azure Container Apps](#).
 - Consider adding service bindings to connect your application to supported Azure databases. These service bindings would eliminate the need for you to provide connection information, including credentials, to your Spring Cloud applications.
 - Consider enabling the Java development stack to collect JVM core metrics for your applications. For more information, see [Java metrics for Java apps in Azure Container Apps](#).
 - Consider adding Azure Monitor alert rules and action groups to quickly detect and address aberrant conditions. For more information, see [Set up alerts in Azure Container Apps](#).
 - Consider replicating your app across the zones in the region by enabling Azure Container Apps zone redundancy. Traffic is load balanced and automatically routed

to replicas if a zone outage occurs. For more information on redundant settings, see [Reliability in Azure Container Apps](#).

- Consider protecting Azure Container Apps from common exploits and vulnerabilities by using Web Application Firewall on Application Gateway. For more information, see [Protect Azure Container Apps with Web Application Firewall on Application Gateway](#).

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)