AWS App2Container

User Guide

AWS App2Container: User Guide

Table of Contents

What is AWS App2Container?

AWS App2Container (A2C) is a command line tool to help you lift and shift applications that run in your on-premises data centers or on virtual machines, so that they run in containers that are managed by Amazon ECS, Amazon EKS, or AWS App Runner.

Moving legacy applications to containers is often the starting point toward application modernization. There are many benefits to containerization:

- Reduces operational overhead and infrastructure costs
- Increases development and deployment agility
- Standardizes build and deployment processes across an organization

Contents

How App2Container works

You can use App2Container to generate container images for one or more applications running on Windows or Linux servers that are compatible with the Open Containers Initiative (OCI). This includes commercial off-the-shelf applications (COTs). App2Container does not need source code for the application to containerize it.

You can use App2Container directly on the application servers that are running your applications, or perform the containerization and deployment steps on a worker machine.

App2Container performs the following tasks:

- Creates an inventory list for the application server that identifies all running ASP.NET (Windows) and Java applications (Linux) that are candidates to containerize.
- Analyzes the runtime dependencies of supported applications that are running, including cooperating processes and network port dependencies.
- Extracts application artifacts for containerization and generates a Dockerfile.
- Initiates builds for the application container.
- Generates AWS artifacts and optionally deploys the containers on Amazon ECS, Amazon EKS, or AWS App Runner. For example:
  - An AWS CloudFormation template to configure required compute, network, and security infrastructure to deploy containers using Amazon ECS, Amazon EKS, or AWS App Runner.
  - An Amazon ECR container image, Amazon ECS task definitions, or AWS CloudFormation templates for Amazon EKS or AWS App Runner that incorporate best practices for security and scalability of the application by integrating with various AWS services.
  - When deploying directly, App2Container can upload AWS CloudFormation resources to an Amazon S3 bucket, and create an AWS CloudFormation stack.
- Optionally creates a CI/CD pipeline with AWS CodePipeline and associated services, to automate building and deploying your application containers.

Accessing AWS through App2Container

When you initialize App2Container, you provide it with your AWS credentials. This allows App2Container to do the following:

- Store artifacts in Amazon S3, if you configured it to do so.
- Create and deploy application containers using AWS services such as Amazon ECS, Amazon EKS, and AWS App Runner.
- Create CI/CD pipelines using AWS CodePipeline.

Pricing

App2Container is offered at no additional charge. You are charged only when you use other AWS services to run your containerized application, such as Amazon ECR, Amazon ECS, Amazon EKS, and AWS App Runner. For more information, see AWS Pricing.

Applications you can containerize using AWS App2Container

App2Container supports the following application types:

- Java applications (Linux)
- ASP.NET applications (Windows)

For more details on application and framework support, expand the section that matches your application operating system.

Important
App2Container does not containerize database layer components. If your application requires access to a database, you must configure your application container to have access to the database server.

Supported applications for Linux

App2Container for Linux supports identification and containerization of Java applications. The CLI identifies Java processes, and can generate container images that replicate the running state of each process. App2Container determines which files to include in the application container image, based on the Java application framework.

For supported application frameworks, App2Container targets only the application files and dependencies that are needed for containerization, thereby minimizing the size of the resulting container image. If App2Container does not find a supported framework running on your application server, or if you have other dependent processes running on your server, App2Container takes a conservative approach to identifying dependencies. For process mode, all non-system files on the application server are included in the container image.

Application mode is supported for the following Java application frameworks:

- Tomcat
- TomEE
- JBoss (standalone mode)

Supported Linux distributions:

- Ubuntu
- CentOS
- RHEL
- Amazon Linux

Unsupported for Java application frameworks:

- Cluster/HA mode

Supported applications for Windows

App2Container supports containerization of ASP.NET applications deployed on IIS, including IIS-hosted WCF applications, running on Windows Server 2016, Windows Server 2019, or Windows Server Core 2004. It uses Windows Server Core as a base image for its container artifacts, matching the Windows Server Core version to the operating system (OS) version of the server where you run containerization commands.

If you use a worker machine to containerize your application, the version matches your worker machine OS. If you are running containerization directly on application servers, the version matches your application server OS.

If your applications are running on Windows Server 2008 or 2012 R2, you might still be able to use App2Container by setting up a worker machine for containerization and deployment steps. App2Container does not support applications running on Windows client operating systems, such as Windows 7 or Windows 10.

Application framework and system requirements

- Containerization commands must run on Windows OS versions that support containers: Windows Server 2016 or 2019, or Windows Server Core 2004. This can be the worker machine, if you configure one, or the application server.
- If you use a worker machine to run containerization commands, App2Container supports Windows Server 2008 and up for the application server.
- IIS 7.5 or later.
- .NET framework version 3.5 or later.
- Docker version 17.07 or later (to install).

Supported applications

- Simple ASP.NET applications running in a single container
- A Windows service running in a single container
- Complex ASP.NET applications that depend on WCF, running in a single container or multiple containers
- Complex ASP.NET applications that depend on Windows services or processes outside of IIS, running in a single container or multiple containers
- Complex, multi-node IIS or Windows service applications, running in a single container or multiple containers

Unsupported applications

- ASP.NET applications that use files and registries outside of IIS web application directories
- ASP.NET applications that depend on features of a Windows operating system version prior to Windows Server Core 2016

Containerizing complex Windows .NET applications with App2Container

Containerization for complex multi-tier Windows .NET applications requires careful planning. When functionality is shared between the root application and one or more lower-level or system applications, you need to make decisions about packaging, deployment, and orchestration for all of the components.

To summarize how AWS App2Container works to containerize a complex Windows .NET application, we'll visit each step in the App2Container workflow, and call out the highlights and things to consider.

Step 1: Setup and initialization

Setup and initialization are the same for complex Windows .NET applications as for other types of applications. Setup tasks include installing software, configuring your AWS profile and IAM permissions, and deciding which servers the App2Container commands should run on. To learn more about setting up your environment before running App2Container for the first time, see Setting up AWS App2Container (p. 14).

After you have completed the setup tasks, but before you use App2Container for the first time, you must initialize the servers where you plan to run App2Container commands. To learn more about initialization and worker machine configuration, see the Initialize (p. 73) section in the App2Container command reference (p. 73).

Step 2: Analysis phase

After you have completed setup and initialization tasks on your servers, App2Container helps you to take an inventory of your running applications, and perform analysis to determine what should be included in your application containers.

Inventory

The first step in the analysis phase is to take an inventory of your applications. When you run the **app2container inventory** command (or the **app2container remote inventory** command, if you have configured a worker machine), App2Container detects the applications that are running in IIS. It also detects the Windows services that could be configured as dependent application components.

App2Container identifies each IIS application or Windows service as a separate application, with its own application ID in the `inventory.json` file. App2Container makes an effort to exclude basic operating system services that you would not want to add to your containers. However, even when these services are excluded, the inventory list can still be quite long.

To narrow the results of the **app2container inventory** or **app2container remote inventory** commands, you can specify what type of application you are looking for with the `--type` option:

- To run an inventory of your IIS applications, you can set the `--type` option to "iis".
- To run an inventory of your Windows services, you can set the `--type` option to "service".

If you don't want App2Container to filter inventory results at all, you can use the `--nofilter` option. This option prevents App2Container from filtering out default system services when building the inventory list. For more information and command syntax, see the **inventory (p. 94)** or **remote inventory (p. 102)** command in the command reference section.

Analysis

When you run the **app2container analyze** or **app2container remote analyze** commands, App2Container analyzes the application component that you specify with the `--application-id` parameter.

App2Container creates the folder structure for the application component, inside of the App2Container directory on your application server or worker machine. It produces the `analysis.json` file, and saves it to the new folder structure, along with other artifacts that are required for containerization. The `analysis.json` file is where you begin to define your container structure.

Tip
Run the **app2container analyze** or **app2container remote analyze** command for every component in your multi-tier application before you configure your container structure.

You can implement the following container structures for a multi-tier Windows .NET application:

- **Multiple application components running in separate containers (recommended)**

  In this scenario, each application component in your multi-tier Windows .NET application runs in a separate container. Relationships between the root application and up to two dependent applications are configured in the `deployment.json` file for the root application. This file is produced during the containerization phase.

  When your application components are running in separate containers, leave the `additionalApps` array in the `analysis.json` file empty for all components.
- **Multiple application components running in a single container**

  In this scenario, the application components in a multi-tier Windows .NET application run together in one container. We recommend that packaging multiple application components in a single container is only done when there are cross-dependencies between the components.

  To specify multiple application components running in a single container, you can include up to five dependent component application IDs in the `additionalApps` array in the `analysis.json` file for the root application.

  Note
  This configuration has the following limitations:
  - Only the port that is defined for the root application is exposed to outside traffic through your load balancer. Ports that are defined for other application components are exposed only from the container, and are not accessible through the load balancer.
  - If you are using remote commands on a worker machine, all of the application components in a multi-tier application must be running on the same application server if you want them to run in a single container.

To learn more about configuring containers, see Configuring application containers (p. 29). To compare configuration examples for a simple .NET application, and for complex multi-tier .NET applications, see the **Examples** section for the .NET application analysis file (p. 32).

For more information and command syntax, see the **analyze (p. 76)** or **remote analyze (p. 96)** command in the command reference section.

Step 3: Containerization

This phase creates containers for your application, based on the output of the analysis phase and on your configuration in the `analysis.json` file.

Extract

If you are using a worker machine to run App2Container commands, or if you want to store an application archive for reference, this phase starts with an **app2container extract** or **app2container remote extract** command. Because this has no effect on the configuration for multi-tier application containers, we will not cover that here.

Containerize

The **app2container containerize** command performs the following tasks for the application that's specified in the `--application id` parameter:

- Extracts application artifacts from the server it runs on, or reads from an extract archive. For complex multi-tier applications, the extract includes all artifacts that are needed for all of the components running in the container.

- Generates a Dockerfile and a container image, based on the application artifacts and the application settings in the `analysis.json` file.
- Creates the `deployment.json` file that defines initial settings for container deployment during the deployment phase.

You must run the **app2container containerize** command for the root application container, and for each additional application component that runs in a separate container. Do not run the command for any components that are included in the root application container. The command displays real-time task completion messages, followed by instructions for next steps. This includes the AWS commands that you run if you are deploying manually.

To configure the `deployment.json` file for a complex multi-tier application, refer to the following scenario that describes your implementation:

- **Multiple application components running in separate containers**

  In this scenario, each application component is running in a separate container, and each has its own deployment file. Before running the **generate app-deployment** command, configure the `deployment.json` file for the root application to include all dependent applications or services in the `dependentApps` array, including the application ID, private root domain, and DNS record name for each one.
- **Multiple application components running in a single container**

  If you are running multiple application components in a single container, the process for configuring the `deployment.json` file is the same as for any other containerized application. Leave the `dependentApps` array empty.

Note
If you are deploying to a specific VPC, make sure that all components point to that VPC in the `vpcId` parameter in the `reuseResources` array in the `deployment.json` file.

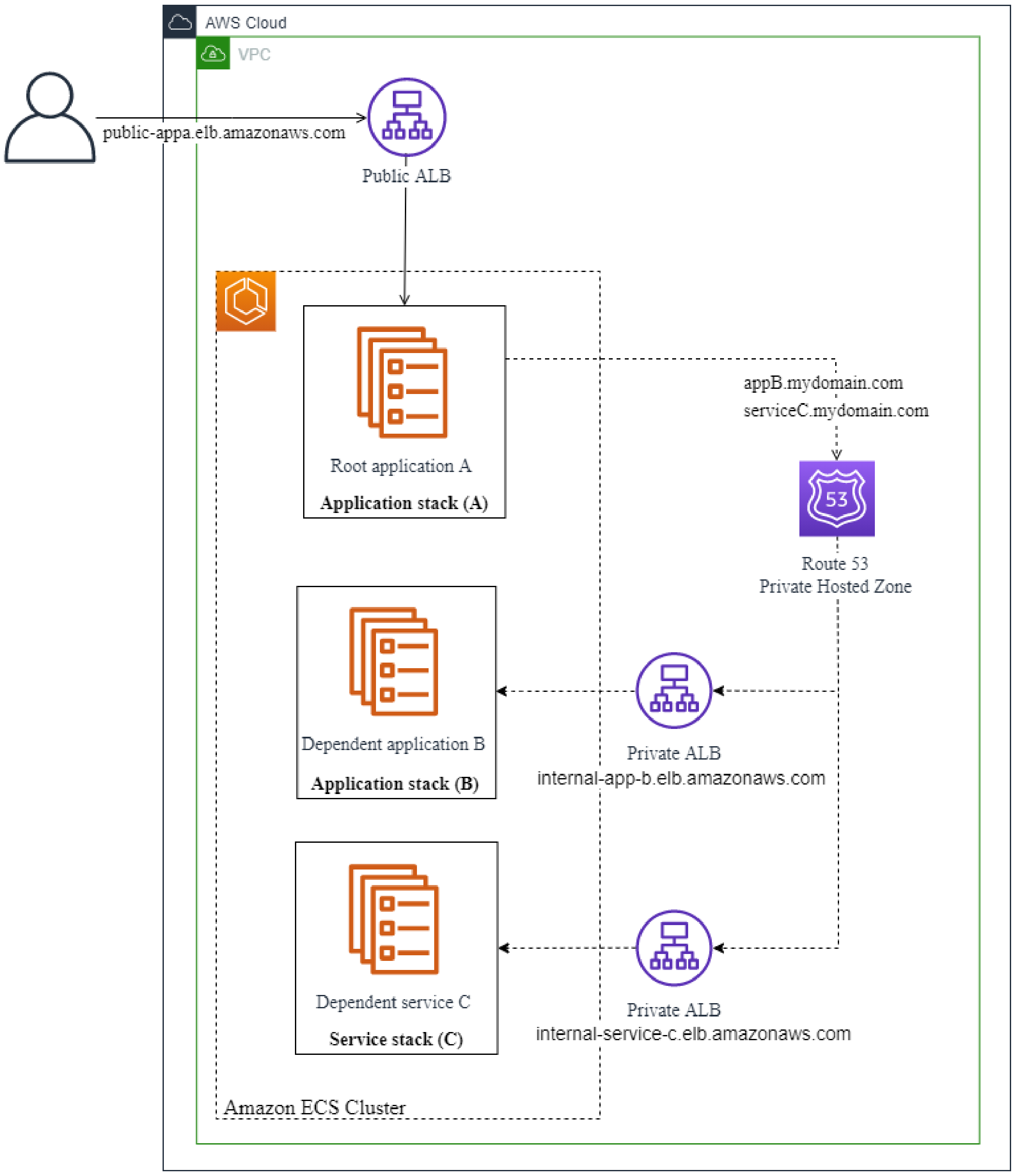
To learn more about configuring your `deployment.json` file, see Configuring container deployment (p. 39). For more information and command syntax for creating your application container, see the **containerize (p. 77)** command in the command reference section.

Step 4: Deployment

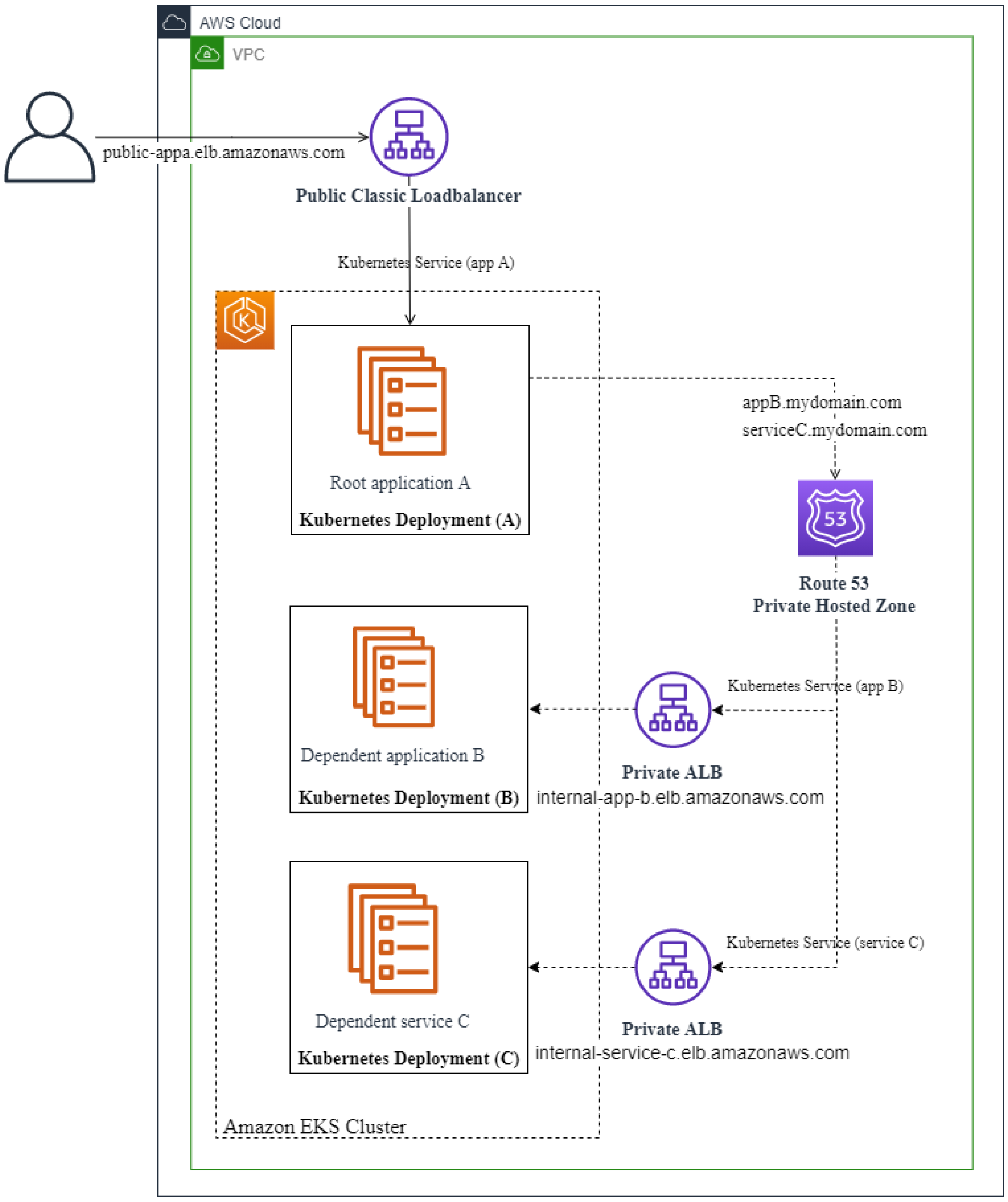
Deployment steps for complex Windows .NET applications with multiple application components running in a single container are handled the same as any other application deployment. For more information and command syntax for deploying your application container, see the **generate app-deployment (p. 82)** command in the command reference section.

The remainder of the content in this section applies to complex Windows .NET applications that have multiple application components running in separate containers, similar to the application example shown in the following diagrams:

Amazon ECS deployment



Amazon EKS deployment


AWS Cloud
VPC
public-appa.elb.amazonaws.com
Public Classic Loadbalancer
Kubernetes Service (app A)
Root application A
Kubernetes Deployment (A)
appB.mydomain.com
serviceC.mydomain.com
Route 53
Private Hosted Zone
Kubernetes Service (app B)
Dependent application B
Kubernetes Deployment (B)
Private ALB
internal-app-b.elb.amazonaws.com
Kubernetes Service (service C)
Dependent service C
Kubernetes Deployment (C)
Private ALB
internal-service-c.elb.amazonaws.com
Amazon EKS Cluster

Normally, you run the **generate app-deployment** command for each application container that you create. However, with complex Windows .NET applications that have dependent applications running in separate containers, App2Container takes care of some of that for you. When you run the **generate app-**

deployment command for the root application, App2Container completes the following tasks for the root application *and each of its dependent application components*:

- Checks for AWS and Docker prerequisites.
- Creates an Amazon ECR repository.
- Pushes the container image to the Amazon ECR repository.
- Generates the following artifacts, depending on your target container management service:

  Amazon ECS

  - An Amazon ECS task definition.
  - The `ecs-master.yml` file that you can use for Amazon ECS deployment.

  Amazon EKS

  - The Kubernetes `eks-master.yml` file that you can use for Amazon EKS deployment.
  - The `eks_deployment.yaml` and `eks_service.yaml` files that you can use with the **kubectl** command.
- Generates a `pipeline.json` file.

Additionally, if you use the `--deploy` option, App2Container takes care of all of those deployments in the order in which they need to run, and configures shared infrastructure settings. When App2Container handles the deployment for you, it follows these conventions:

- The root application and all dependent application components are deployed to the same cluster.
- All dependent application components are configured with an internal load balancer only.
- Each application component has its own Amazon ECS or Amazon EKS service running in a shared cluster.

If you want to customize the deployment artifacts, you can deploy manually, using the AWS Management Console or AWS CLI when you are ready.

For deployment steps, choose the tab that matches your deployment scenario.

Automated (A2C)

Follow these steps if you are using the App2Container automated deployment.

1. Verify that the values are set correctly in the `deployment.json` files for all of your application components, before running the **generate app-deployment** command for your root application, as follows:
   - None of the application components in the multi-tier application should specify `reuseCfnStack`.
   - Dependent application components should not specify any of the following parameters: `vpcId`, `gMSAParameters`.
   - The following parameters can be specified in the root application, and App2Container applies the same values for all dependent application components: `vpcId`, `resourceTags`, and `gMSAParameters`.
2. The following example shows the **generate app-deployment** command for the root application in our sample multi-tier application, using the `--deploy` option, with the `--application-id` parameter set to the application ID for the root application. This example handles the full deployment for all application components.

```
PS> app2container generate app-deployment --deploy --application-id iis-
colormvciis-b69c09ab --profile admin-profile
# AWS prerequisite check succeeded
# Docker prerequisite check succeeded
... [more notifications as deployment steps are completed for each dependent
 application component, followed by the root application and shared configurations]
Deployment successful for application iis-colormvciis-b69c09ab

The URL to your Load Balancer Endpoint is:
a2c-i-Publi-1A2BCD3EFGRW-4567890123.us-west-2.elb.amazonaws.com

Successfully created Amazon ECS stack a2c-iis-colormvciis-b69c09ab-ECS. Check the
 AWS CloudFormation Console for additional details.
3. Set up a pipeline for your application stack using app2container:

        app2container generate pipeline --application-id iis-colormvciis-b69c09ab
```

The first deployment for a dependent application component creates shared AWS resources, such as the VPC and Amazon ECS or Amazon EKS cluster. After the first dependent application component is successfully deployed, App2Container updates deployment artifacts for all of the other application components to reference the shared AWS resources prior to completing the remaining deployments.

Manual (AWS CLI)

Follow these steps to customize your deployment files and use the AWS CLI to deploy manually. We do not include AWS Management Console instructions here. However, you can follow the same general order of operations in the console.

1. Verify that the values are set correctly in the `deployment.json` files for all of your application components, before running the **generate app-deployment** command for your root application, as follows:

   - None of the application components in the multi-tier application should specify `reuseCfnStack`.
   - Dependent application components should not specify any of the following parameters: `vpcId`, `gMSAParameters`.
   - The following parameters can be specified in the root application, and App2Container applies the same values for all dependent application components: `vpcId`, `resourceTags`, and `gMSAParameters`.

2. The following example shows the **generate app-deployment** command for the root application in our sample multi-tier application, with the `--application-id` parameter set to the application ID for the root application. The `--deploy` option is not used in this case, as we plan to customize deployment files and then deploy using AWS CLI commands to control deployment for each application component.

   Note
   App2Container creates deployment artifacts for all application components in the complex Windows .NET application when you run the **generate app-deployment** command for the root application.

   Use the **generate app-deployment** command, specifying the application ID for your root application, as follows:

```
PS> app2container generate app-deployment --application-id iis-colormvciis-b69c09ab
 --profile admin-profile
# AWS prerequisite check succeeded
# Docker prerequisite check succeeded
... [more notifications as deployment steps are completed for each dependent
 component, followed by the root application and shared configurations]
CloudFormation templates and additional deployment artifacts generated successfully
 for application iis-colormvciis-b69c09ab

You're all set to use AWS CloudFormation to manage your application stack.

Next Steps:
1. Create application stacks for first dependent application using the AWS CLI or
 the AWS Console. AWS CLI commands:

        aws cloudformation deploy --template-file C:\Users\Administrator\AppData
\Local\app2container\iis-dependentappb-12345bcd\EcsDeployment\ecs-master.yml --
capabilities CAPABILITY_NAMED_IAM CAPABILITY_AUTO_EXPAND --stack-name a2c-iis-
dependentappb-12345bcd-ECS

2. Required! Reuse the VpcId, ClusterId and PublicSubnets from above
 CloudFormation console outputs and assign them in master templates of service-
colorwindowsservice-69f90194, iis-colormvciis-b69c09ab
If your other dependent application(s) that share the same root domain, also assign
 HostedZoneId to their master template(s).
Create application stacks for remaining applications using the AWS CLI or the AWS
 Console. AWS CLI commands:

        aws cloudformation deploy --template-file C:\Users\Administrator\AppData
\Local\app2container\service-colorwindowsservice-69f90194\EcsDeployment\ecs-
master.yml --capabilities CAPABILITY_NAMED_IAM CAPABILITY_AUTO_EXPAND --stack-name
 a2c-service-colorwindowsservice-69f90194-ECS

        aws cloudformation deploy --template-file C:\Users\Administrator\AppData
\Local\app2container\iis-colormvciis-b69c09ab\EcsDeployment\ecs-master.yml --
capabilities CAPABILITY_NAMED_IAM CAPABILITY_AUTO_EXPAND --stack-name a2c-iis-
colormvciis-b69c09ab-ECS

3. Set up a pipeline for your application stack using app2container:

        app2container generate pipeline --application-id iis-colormvciis-b69c09ab
```

3. Review the deployment artifacts that were generated in the prior step, and customize the YAML deployment templates and other deployment artifacts as needed.

   Manual deployment follows this step, beginning with one of the dependent applications. The first deployment creates any shared infrastructure that is required.

   Note
   If you are using an existing VPC, the `vpcId` that you specified in the `deployment.json` file for the root application should be reflected in the YAML deployment templates for all of the dependent applications.

4. To deploy your first dependent application and create shared infrastructure, run the following command in the AWS CLI, using your dependent application's details.

```
PS> aws cloudformation deploy --template-file C:\Users\Administrator\AppData
\Local\app2container\iis-dependentappb-12345bcd\EcsDeployment\ecs-master.yml --
capabilities CAPABILITY_NAMED_IAM CAPABILITY_AUTO_EXPAND --stack-name a2c-iis-
dependentappb-12345bcd-ECS
```

5. After your first stack is ready (stack status is `CREATE_COMPLETE`), update the YAML deployment templates for all remaining application components in your application to reference the following shared infrastructure in the parameters for existing resources:

   - VpcId
   - PublicSubnets

- ClusterId

Additionally, for any remaining dependent applications, update the following references:

- DomainName
- RecordName
- ExistingHostedZoneId – update this if dependent applications share the root domain, or if they are using an existing domain.
- RecordExist – set this to "true" (string) if the record already exists in the hosted zone. If you are creating a new domain, set this to "false". The default value is "true".

6. Deploy any remaining dependent applications, using your application component information and the updated YAML deployment templates, with the **cloudformation deploy** command. The following command example deploys the service component in our sample multi-tier application.

```
PS> aws cloudformation deploy --template-file C:\Users\Administrator\AppData\Local
\app2container\service-colorwindowsservice-69f90194\EcsDeployment\ecs-master.yml --
capabilities CAPABILITY_NAMED_IAM CAPABILITY_AUTO_EXPAND --stack-name a2c-service-
colorwindowsservice-69f90194-ECS
```

7. After you've created all of your dependent component stacks, deploy your root application with the **cloudformation deploy** command. The following command example deploys the root application in our sample multi-tier application.

```
PS> aws cloudformation deploy --template-file C:\Users\Administrator\AppData\Local
\app2container\iis-colormvciis-b69c09ab\EcsDeployment\ecs-master.yml --capabilities
 CAPABILITY_NAMED_IAM CAPABILITY_AUTO_EXPAND --stack-name a2c-iis-colormvciis-
b69c09ab-ECS
```

Tip

It can take a few minutes to spin up an AWS CloudFormation stack, along with the other infrastructure that is created for your deployment. You can use one of the following methods to check the stack status for your deployment:

- Sign in to the AWS Management Console and open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.

  In the console, you can see stacks that are being created, as well as existing stacks. For more information, see Viewing AWS CloudFormation stack data and resources on the AWS Management Console in the *AWS CloudFormation User Guide*.
- Use one of these AWS CloudFormation commands in the AWS CLI: **list-stacks** or **describe-stacks**. For more information, see **Available Commands** in the AWS CLI Command Reference.
- Use one of these AWS CloudFormation API commands: **ListStacks** or **DescribeStacks**. For more information, see Actions in the *AWS CloudFormation API Reference*.

Setting up AWS App2Container

Complete these tasks before you use App2Container for the first time.

Tasks

Sign up for AWS

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all services in AWS. You are charged only for the services that you use.

If you do not have an AWS account already, use the following procedure to create one.

To create an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Decide where containerization will run

To use App2Container on the server where the applications are running, you must set up an AWS profile, install App2Container, and install the Docker engine. If your server does not meet the requirements to containerize your application and deploy it to AWS, or if you do not want to install the Docker engine on the application server, you can set up and use a worker machine. On the worker machine, you can run the steps to containerize your application and deploy it to AWS, or you can set up connectivity between the worker machine and the application servers to run remote commands from the worker machine, targeting the application servers.

The following are example situations where you might decide to set up a worker machine:

- Your application servers are running in an on-premises data center and they do not have internet access.
- Your application server is running on a Windows operating system that does not support containers. For more information, see Supported applications (p. 3).
- You prefer to use a dedicated server to run the containerization and deployment steps.

- You want to consolidate your work by using a worker machine to run commands for all of your application servers.

When you set up a worker machine to handle the steps to containerize and deploy your applications, it must have the same base operating system as your application server (Linux or Windows), and the operating system must support containers. We recommend that you launch an Amazon EC2 instance as the worker machine, using an Amazon Machine Image (AMI) that is optimized for Amazon ECS.

Grant permissions to run AWS App2Container commands

App2Container needs access to AWS services in order to run most of its commands. There are two very different sets of permissions needed to run **app2container** commands.

- The general purpose IAM user, group, or role can run all of the commands *except* commands that are run with the `--deploy` option.
- For deployment, App2Container must be able to create or update AWS objects for container management services (Amazon ECR with Amazon ECS, Amazon EKS, or AWS App Runner), and to create CI/CD pipelines with AWS CodePipeline. This requires elevated permissions that should only be used for deployment.

We recommend that you create general purpose IAM resources, and if you plan to use App2Container to deploy your containers or create pipelines, that you create separate IAM resources for deployment.

For instructions on how to set up your IAM resources for App2Container, and policy examples that include resources and actions that App2Container needs access to, see Identity and access management in App2Container (p. 63).

Note
You can use an instance profile to pass an IAM role to an Amazon EC2 instance. App2Container detects if there is an instance profile associated with the application server or worker machine when you run the **init** command. If it detects an instance profile, the **init** command prompts if you want to use it.
To find out more about using instance profiles, see Using instance profiles in the *IAM User Guide*.

Enable remote access for a worker machine (optional)

To enable your worker machine to run remote commands for your application servers, you must ensure that the worker machine can connect.

For the required setup to enable remote access, choose the operating system tab that matches your application server.

Linux

For Linux application servers, you can use SSH key-based or SSH Certificate-based connections. You must ensure that there is network connectivity between the worker machine and the application server, and verify that your worker machine can connect.

Windows

To connect to a Windows application server from a Windows Server 2016 or 2019 worker machine, use the WinRM protocol. Your application server must meet the requirements that are listed for Windows in the Supported applications (p. 3) section of this user guide.

Note
App2Container does not support applications running on Windows client operating systems, such as Windows 7 or Windows 10.

1. **Worker machine**

   To ensure that you can run PowerShell scripts on the worker machine, set the PowerShell Execution Policy to one of the following values:

   RemoteSigned

   Example:

   ```
   PS> Set-ExecutionPolicy RemoteSigned
   ```

   Unrestricted

   Example:

   ```
   PS> Set-ExecutionPolicy Unrestricted
   ```

2. **Application servers**

   Complete the following steps on each application server to enable remote access from the worker machine.

   1. Ensure network connectivity to the application server over WinRM port 5986.
   2. Download the WinRMSetup.ps1 PowerShell script to your application server from the following location: WinRMSetup.ps1.

      Note
      Checksum files for this script can be downloaded using the following links:
      - WinRMSetup.ps1.sha256
      - WinRMSetup.ps1.md5
   3. Download the New-SelfsignedCertificateEx.ps1 PowerShell script from the Microsoft Technet gallery. The WinRMSetup.ps1 PowerShell script from step 2 uses it to generate a self-signed certificate.

      Note
      This script must run from the same directory where the WinRMSetup.ps1 PowerShell script from step 2 is located.
   4. Run the WinRMSetup.ps1 PowerShell script on the application server. The script ensures that WinRM is enabled, and generates self-signed certificates that are used to secure the connection from the worker machine.

Configure your AWS profile

AWS App2Container requires command line access to AWS resources for containerization and deployment commands. It uses information from your AWS profile to configure access to AWS resources for your account. To run App2Container commands, you must install and configure a command line tool on the application servers and worker machines where you run the commands.

Note

- AWS Tools for Windows PowerShell is required for running App2Container commands in PowerShell on a Windows server.
- Tools for Windows PowerShell comes pre-installed on Windows-based Amazon Machine Images (AMIs). If your application server or worker machine is an Amazon EC2 instance that was launched from one of these AMIs, you can skip to configuring your AWS profile. See Shared credentials in the *AWS Tools for Windows PowerShell User Guide* for more details.

To install the AWS Command Line Interface (AWS CLI) or AWS Tools for Windows PowerShell command line tools, and to configure your AWS profile, follow the instructions on the tab that matches your command line tool.

AWS CLI

To install the AWS CLI and set up your AWS profile, follow these steps:

1. Install the AWS CLI according to the instructions in the *AWS Command Line Interface User Guide*. For more information, see Installing the AWS CLI.
2. To configure your AWS default profile, use the **aws configure** command. For more information, see Configuration basics in the *AWS Command Line Interface User Guide*.

Tools for Windows PowerShell

To install Tools for Windows PowerShell and set up your AWS profile, follow these steps:

1. Install the Tools for Windows PowerShell according to the instructions in the *AWS Tools for Windows PowerShell User Guide*. For more information see Installing the AWS Tools for Windows PowerShell.
2. To set up your AWS default profile, use the Initialize-AWSDefaultConfiguration cmdlet. For more information about shared credentials in Tools for Windows PowerShell, see Shared credentials in the *AWS Tools for Windows PowerShell User Guide*.

After you containerize your applications, you can also use the AWS CLI or Tools for Windows PowerShell to deploy them on AWS, though we recommend using the `--deploy` option with the **generate app-deployment** and **generate pipeline** commands to do your deployment.

Install the Docker engine

App2Container uses the Docker engine (Docker CE) to create container images and generate Dockerfiles that run the containers hosted on Amazon ECS, Amazon EKS, or AWS App Runner. You must install the Docker engine on the application server or worker machine that you'll use to containerize the application using the **containerize** command.

Install Docker on Linux

Use the following procedure to install Docker on Linux.

To install the Docker engine

1. **Install Docker**

   Choose your Linux distribution from the following options, and follow instructions to download and install the Docker engine, using the links provided.

Amazon Linux

To download and install the Docker engine on Amazon Linux instances, see Docker basics for Amazon ECS in the *Amazon Elastic Container Service Developer Guide*. This works with any Amazon Linux instance.

RHEL

Recent versions of RHEL do not natively support the Docker engine. However, you can still download and install the Docker engine on RHEL to create containers that will be hosted and run on Amazon ECS, Amazon EKS, or AWS App Runner. To do this, follow the instructions given for CentOS on the Docker website: Install Docker engine.

All other supported distributions (CentOS, Ubuntu)

To download and install the Docker engine for other supported Linux distributions, follow the instructions for your Linux distribution on the Docker website: Install Docker engine.

2. **Verify the Docker installation**

   To verify that your Docker installation was successful, run the following command.

   ```
   $ docker run -it hello-world
   ```

   When the command runs, it pulls the latest hello-world application from the Docker repository, if applicable. When the application has finished downloading, it displays a "Hello" message followed by information on how this command verified your installation of Docker.

Install Docker on Windows

Use the following procedure to install Docker on Windows.

To install the Docker engine

1. **Install Docker version 17.07 or later**

   To download and install the Docker engine on Windows, see Get started: Prep Windows for containers (Install Docker section).

2. **Verify the Docker installation**

   To verify that your Docker installation was successful, run the following command.

   ```
   PS> docker run -it hello-world
   ```

   When the command runs, it pulls the latest hello-world application from the Docker repository, if applicable. When the application has finished downloading, it displays a "Hello" message followed by information on how this command verified your installation of Docker.

Getting started with AWS App2Container

AWS App2Container is a tool that helps you break down the work of moving your applications into containers, and configuring them to be hosted in AWS using the Amazon ECS, Amazon EKS, or App Runner container management services. Explore the resources listed below to help you get started with containers. Or to get started using App2Container commands, skip to the tutorial for the operating system that your application runs on.

Understanding Docker containers

The following resources can help you get the most out of your application containers by understanding what goes into them.

- To learn more about Docker containers on AWS, see What is Docker?.
- Use the Docker command line reference to look up Docker commands. See Use the Docker command line.

Tutorials

These tutorials walk you through the basics of using App2Container to containerize your applications.

- Containerizing a Java application on Linux (p. 19)
- Containerizing a .NET application on Windows (p. 23)

Containerizing a Java application on Linux

This tutorial takes you through the steps to containerize a legacy Java application on Linux using App2Container, and to deploy it on Amazon ECS, Amazon EKS, or AWS App Runner. You can complete all steps on the application server, or you can perform the initial steps on the application server and perform the containerization and deployment steps on a worker machine.

Tasks

Prerequisites

Verify that you have completed the following prerequisites:

- Your application environment meets all of the requirements that are listed in the Supported applications (p. 3) section.
- You installed the AWS CLI and configured the AWS profile on your server. See Configure your AWS profile (p. 16) in the Setting up section of this user guide for more information.
- You installed the Docker engine on the server where you are running containerization and deployment steps. See Install the Docker engine (p. 17) in the Setting up section of this user guide for more information.
- There are one or more Java applications running on the application server.
- You have root access on the application server (and worker machine, if using).
- The application server (and worker machine, if using) has **tar** and 20 GB of free space.

Step 1: Install App2Container

App2Container for Linux is packaged as a tar.gz archive. The archive contains an interactive shell script that installs App2Container on your server. If you are using an application server and a worker machine, you must install App2Container on both.

To download and install App2Container for Linux

1. Download the installation file in one of the following ways:

   - Use the **curl** command to download the App2Container installation package from Amazon S3.

     ```
     $ curl -o AWSApp2Container-installer-linux.tar.gz https://app2container-release-us-east-1.s3.us-east-1.amazonaws.com/latest/linux/AWSApp2Container-installer-linux.tar.gz
     ```

   - Use your browser to download the installer from the following URL: https://app2container-release-us-east-1.s3.us-east-1.amazonaws.com/latest/linux/AWSApp2Container-installer-linux.tar.gz.

2. Extract the package to a local folder on the server.

   ```
   $ sudo tar xvf AWSApp2Container-installer-linux.tar.gz
   ```

3. Run the install script that you extracted from the package and follow the prompts.

   ```
   $ sudo ./install.sh
   ```

You can check the downloaded tar.gz installer archive for integrity by validating the MD5 and SHA256 hashes of the local file against the published hash files.

To verify the authenticity of the download

1. **Generate hashes to verify**

   From the directory where you downloaded your tar.gz installer, run the following commands to generate the hash of the downloaded tar.gz file.

   ```
   $ md5sum AWSApp2Container-installer-linux.tar.gz
   a0a1234f567bf89012345a6ce7bf89a0  AWSAppContainerization-installer-linux.tar.gz
   $ sha256sum AWSApp2Container-installer-linux.tar.gz
   01bc2d345f6789012345bd6aa789012345d67dea8b9c0f1234ee5a67890123d4
    AWSAppContainerization-installer-linux.tar.gz
   ```

2. **Verify hashes against public files**

   Download the App2Container hash files from Amazon S3 using the following links, and compare the contents to the hashes that you generated in step 1:

   - AWSApp2Container-installer-linux.tar.gz.md5.
   - AWSApp2Container-installer-linux.tar.gz.sha256.

Step 2: Initialize App2Container

On each server where you installed App2Container, run the init (p. 92) command as follows.

```
$ sudo app2container init
```

You are prompted to provide the following information. Choose *<enter>* to accept the default value.

- Workspace directory path – A local directory where App2Container can store artifacts during the containerization process. The default is `/root/app2container`.
- AWS profile – Contains information needed to run App2Container, such as your AWS access keys. For more information about AWS profiles, see Configure your AWS profile (p. 16).

  Note
  If App2Container detects an instance profile for your server, the **init** command prompts if you want to use it. If you don't specify any value, App2Container uses your AWS default profile.
- Amazon S3 bucket – You can optionally provide the name of an Amazon S3 bucket where you can extract artifacts using the **extract** command. The **containerize** command uses the extracted components to create the application container if the Amazon S3 bucket is configured. The default is no bucket.
- You can optionally upload logs and command-generated artifacts automatically to App2Container support when an app2container command crashes or encounters internal errors.
- Permission to collect usage metrics – You can optionally allow App2Container to collect information about the host operating system, application type, and the **app2container** commands that you run. The default is to allow the collection of metrics.
- Whether to enforce signed images – You can optionally require that images are signed using Docker Content Trust (DCT). The default is no.

Step 3: Analyze your application

On the application server, use the following procedure to prepare to containerize the application.

To prepare for containerization

1. Run the inventory (p. 94) command as follows to list the Java applications that are running on your server.

   ```
   $ sudo app2container inventory
   ```

   The output includes a JSON object collection with one entry for each application. Each application object will include key/value pairs as shown in the following example.

   ```
   "java-app-id": {
       "processId": pid,
       "cmdline": "/user/bin/java ...",
   ```

```
    "applicationType": "java-apptype"
}
```

2. Locate the application ID for the application to convert in the JSON output of the **inventory** command, and then run the analyze (p. 76) command as follows, replacing *java-app-id* with the application ID that you located.

   ```
   $ sudo app2container analyze --application-id java-app-id
   ```

   The output is a JSON file, `analysis.json`, stored in the workspace directory that you specified when you ran the **init** command.
3. (Optional) You can edit the information in the `containerParameters` section of `analysis.json` as needed before continuing to the next step.

Step 4: Transform your application

The transform phase depends on whether you are running all steps on the application server, or are using the application server for the analysis and a worker machine for containerization and deployment.

To containerize the application on the application server

If you are using an application server for all steps, run the containerize (p. 77) command as follows.

```
$ sudo app2container containerize --application-id java-app-id
```

The output is a set of deployment files that are stored in the workspace directory that you specified when you ran the **init** command.

To containerize the application on a worker machine

If you are using a worker machine for containerization and deployment, use the following procedure to transform the application.

1. On the application server, run the extract (p. 80) command as follows.

   ```
   $ sudo app2container extract --application-id java-app-id
   ```

2. If you specified an Amazon S3 bucket when you ran the **init** command, the archive is extracted to that location. Otherwise, you can manually copy the resulting archive file to the worker machine.
3. On the worker machine, run the containerize (p. 77) command as follows.

   ```
   $ sudo app2container containerize --input-archive /path/extraction-file.tar
   ```

   The output is a set of deployment artifacts that are stored in the workspace directory that you specified when you ran the **init** command.

Step 5: Deploy your application

Run the generate app-deployment (p. 82) command as follows to deploy the application on AWS.

```
$ sudo app2container generate app-deployment --deploy --application-id java-app-id
```

You have now deployed your application! You can find the deployment artifacts that the **generate app-deployment** command created for you in the local directory for your application.

Step 6: Clean up

To remove App2Container from your application server or worker machine, delete the `/usr/local/app2container` folder where it is installed, and then remove this folder from your path.

To clean up your AWS profile, use the **aws configure set** command. For more information, see Set and view configuration settings in the *AWS Command Line Interface User Guide*.

Containerizing a .NET application on Windows

This tutorial takes you through the steps to containerize a legacy .NET application running in IIS on Windows using App2Container, and to deploy it on Amazon ECS, Amazon EKS, or AWS App Runner. You can complete all steps on the application server, or you can perform the initial steps on the application server and perform the containerization and deployment steps on a worker machine.

Tasks

Prerequisites

Verify that you have completed the following prerequisites:

- Your application environment meets all of the requirements that are listed for Windows in the Supported applications (p. 3) section.
- You installed the AWS Tools for Windows PowerShell to configure the AWS profile on your server. See Configure your AWS profile (p. 16) in the Setting up section of this user guide for more information.
- You installed the Docker engine on the server where you are running containerization and deployment steps. See Install the Docker engine (p. 17) in the Setting up section of this user guide for more information.
- There are one or more applications running in IIS on the application server.
- You are a Windows administrator on the application server (and worker machine, if using).
- The application server or worker machine has PowerShell version 5.1 or later and at least 20-30 GB of free space.

Note
App2Container does not support applications running on Windows client operating systems, such as Windows 7 or Windows 10.

Step 1: Install App2Container

App2Container for Windows is packaged as a zip archive. The package contains a PowerShell script that installs App2Container. If you are using an application server and a worker machine, you must install App2Container on both.

To download and install App2Container for Windows

1. Download the App2Container installation package, AWSApp2Container-installer-windows.zip.
2. Extract the package to a local folder on the server and navigate to that folder.
3. Run the install script from the folder where you extracted it, and follow the prompts.

   ```
   PS> .\install.ps1
   ```
4. (Optional) To verify the authenticity of the download, use the `Get-AuthenticodeSignature` PowerShell command as follows to get the Authenticode Signature of the App2Container executable.

   ```
   PS> Get-AuthenticodeSignature C:\Users\Administrator\app2container\AWSApp2Container\bin
   \app2container.exe
   ```

Step 2: Initialize App2Container

On each server where you installed App2Container, run the init (p. 92) command as follows.

```
PS> app2container init
```

You are prompted to provide the following information. Choose *<enter>* to accept the default value.

- Workspace directory path – A local directory where App2Container can store artifacts during the containerization process. The default is `C:\Users\Administrator\AppData\Local\app2container`.
- AWS profile – Contains information needed to run App2Container, such as your AWS access keys. For more information about AWS profiles, see Configure your AWS profile (p. 16).

  Note
  If App2Container detects an instance profile for your server, the **init** command prompts if you want to use it. If you don't specify any value, App2Container uses your AWS default profile.
- Amazon S3 bucket – You can optionally provide the name of an Amazon S3 bucket where you can extract artifacts using the **extract** command. The **containerize** command uses the extracted components to create the application container if the Amazon S3 bucket is configured. The default is no bucket.
- You can optionally upload logs and command-generated artifacts automatically to App2Container support when an app2container command crashes or encounters internal errors.
- Permission to collect usage metrics – You can optionally allow App2Container to collect information about the host operating system, application type, and the **app2container** commands that you run. The default is to allow the collection of metrics.
- Whether to enforce signed images – You can optionally require that images are signed using Docker Content Trust (DCT). The default is no.

Step 3: Analyze your application

On the application server, use the following procedure to prepare to containerize the application.

To prepare for containerization

1. Run the inventory (p. 94) command as follows to list the ASP.NET applications that are running on your server.

```
PS> app2container inventory
```

The output includes a JSON object collection with one entry for each application. Each application object will include key/value pairs as shown in the following example.

```
"iis-app-id": {
    "siteName": My site name,
    "bindings": "http/*:80:",
    "applicationType": "iis",
    "discoveredWebApps": [
        "app1",
        "app2"
    ]
}
```

2. Locate the application ID for the application to convert in the JSON output of the **inventory** command, and then run the analyze (p. 76) command as follows, replacing *iis-app-id* with the application ID that you located.

   ```
   PS> app2container analyze --application-id iis-app-id
   ```

   The output is a JSON file, `analysis.json`, stored in the workspace directory that you specified when you ran the **init** command.
3. (Optional) You can edit the information in the `containerParameters` section of `analysis.json` as needed before continuing to the next step.

Step 4: Transform your application

The transform phase depends on whether you are running all steps on the application server or using the application server for the analysis and a worker machine for containerization and deployment.

To containerize the application on the application server

If you are using an application server for all steps, run the containerize (p. 77) command as follows.

```
PS> app2container containerize --application-id iis-app-id
```

The output is a set of deployment files stored in the workspace directory that you specified when you ran the **init** command.

To containerize the application on a worker machine

If you are using a worker machine for containerization and deployment, use the following procedure to transform the application.

1. On the application server, run the extract (p. 80) command as follows.

   ```
   PS> app2container extract --application-id iis-app-id
   ```

2. If you specified an Amazon S3 bucket when you ran the **init** command, the archive is extracted to that location. Otherwise, you can manually copy the resulting archive file to the worker machine.
3. On the worker machine, run the containerize (p. 77) command as follows.

   ```
   PS> app2container containerize --input-archive drive:\path\extraction-file.zip
   ```

The output is a set of deployment artifacts that are stored in the workspace directory that you specified when you ran the **init** command.

Step 5: Deploy your application

Run the generate app-deployment (p. 82) command as follows to deploy the application on AWS.

```
PS> app2container generate app-deployment --deploy --application-id iis-smarts-51d2dbf8
```

You have now deployed your application! You can find the deployment artifacts that the **generate app-deployment** command created for you in the local directory for your application.

Applications using Windows authentication
For applications using Windows authentication, you can use the `gMSAParameters` inside of the `deployment.json` file to set the gMSA-related artifacts automatically during generation of your AWS CloudFormation template.
Perform the actions in the list below once per Active Directory domain before you update the gMSA parameters.

- Set up a secret in SecretsManager that stores the Domain credentials with the following key value pairs:

Key	Value
Username	<DomainNetBIOSName>\<DomainUser>
Password	<DomainUserPassword>

- For the VPC with the Domain Controller, verify that the DHCP options are set to reach the Domain Controller. The options for `DomainName` and `DomainNameServers` must be set correctly. See DHCP options sets for more information about how to set DHCP options.

Step 6: Clean up

To remove App2Container from your application server or worker machine, delete the `C:\Users\Administrator\app2container` folder where it is installed, and then remove this folder from your path.

To clean up your AWS profile, see Removing Credential Profiles in the *AWS Tools for Windows PowerShell User Guide*.

Configuring your application

Containerizing your application and creating pipelines with App2Container requires configuration throughout the process. This section of the guide describes the files that are created by **app2container** commands, the fields that they contain, and which fields are configurable. App2Container commands primarily generate JSON files, using standard JSON notation. Field details for the files included here indicate where there are specific requirements for the values.

Creating IAM resources is covered separately, under the Security section. For more information and instructions about how to set up IAM resources for App2Container, see Identity and access management in App2Container (p. 63).

You can consolidate your containerization workload by configuring connections to your application servers to run containerization workflows remotely, using App2Container remote commands from your worker machine. Prior to running remote commands, you must configure the connections that the worker machine uses for its target application servers. For more information on configuring connections, see the remote configure (p. 98) command reference page.

Contents

Manage secrets for AWS App2Container

App2Container uses AWS Secrets Manager to manage the credentials for connecting your worker machine to application servers in order to run remote commands. Secrets Manager encrypts your secrets for storage, and provides an Amazon Resource Name (ARN) for you to access the secret. When you run the **remote configure** command, you provide the secret ARN for App2Container to use to connect to your target server when running the remote command. For more information about Secrets Manager, see What Is AWS Secrets Manager?

The secret that App2Container uses to connect to an application server varies, based on the application server's operating system platform. To see more information about storing secrets for your application server, expand the platform section that matches.

Create a remote access secret for Linux (console)

For Linux, you can store either the SSH private key or the Certificate and SSH private key in Secrets Manager. To create a secret in Secrets Manager for remote access to your application server, follow these steps:

1. Open the Secrets Manager console at https://console.aws.amazon.com/systems-manager/.
2. Choose **Store a new secret**. This takes you to the Secrets Manager console wizard.
3. In the **Select secret type** panel, choose the **Other type of secrets** option. This enables you to store a key that is used programmatically, via API calls.
4. **Specify key/value pairs to be stored in the secret**

    a. On the **Secret key/value** tab, enter your first key/value pair as follows:

- Enter **username** in the first box.
- Enter the plaintext username value to use with SSH in the second box.

b. Choose **+ Add row** to add the next key/value pair.

c. Enter the key/value pair for your SSH private key:

- Enter **key** in the first box.
- Copy the base64-encoded string representing your private key file into the second box.

Note

To base64-encode your key file, you can use the following command, where `.ssh/id_rsa` is the private key being used:

```
$ base64 .ssh/id_rsa
```

d. Choose **+ Add row** to add the next key/value pair.

e. Enter the key/value pair for your SSH Certificate (optional):

- Enter **cert** in the first box.
- Copy the base64-encoded string representing your signed certificate file into the second box.

Note

To base64-encode your signed certificate file, you can use the following command, where `.ssh/id_rsa-cert.pub` is the private key being used:

```
$ base64 .ssh/id_rsa-cert.pub
```

f. Choose **Next** to continue.

5. **Name the secret**

   a. In the **Secret name and description** panel, enter a name for your secret in the **Secret name** box. You can also enter optional information to help identify your secret, such as **Description**, or you can enter tags in the **Tags** panel.

   b. Choose **Next** to continue.

6. Additional steps are optional. Choose **Next** to continue to the review page.
7. Choose **Store** to store your secret.

Create a remote access secret for Windows (console)

For Windows, you can store the Username and Password to access the Windows application server. In most cases, this translates to a set of credentials for a domain user with access to the application servers.

To create a secret in Secrets Manager for remote access to your application server, follow these steps:

1. Open the Secrets Manager console at https://console.aws.amazon.com/systems-manager/.
2. Choose **Store a new secret**. This takes you to the Secrets Manager console wizard.
3. In the **Select secret type** panel, choose the **Other type of secrets** option. This enables you to store a key that is used programmatically, via API calls.
4. **Specify key/value pairs to be stored in the secret**

   a. On the **Secret key/value** tab, enter your first key/value pair as follows:

   - Enter **username** in the first box.
   - Enter the plaintext username value to use with connection credentials for your application server in the second box.

   b. Choose **+ Add row** to add the next key/value pair.
   c. Enter the key/value pair for your password:
      - Enter `password` in the first box.
      - Enter the password value into the second box.
   d. Choose **Next** to continue.
5. **Name the secret**
   a. In the **Secret name and description** panel, enter a name for your secret in the **Secret name** box. You can also enter optional information to help identify your secret, such as **Description**, or you can enter tags in the **Tags** panel.
   b. Choose **Next** to continue.
6. Additional steps are optional. Choose **Next** to continue to the review page.
7. Choose **Store** to store your secret.

Configuring application containers

When you run the **analyze (p. 76)** command, an `analysis.json` file is created for the application that is specified in the `--application-id` parameter. The **containerize** command uses this file to build the application container image and to generate artifacts.

You can configure the fields in the `containerParameters` section before running the **containerize** command to customize your application container. For configurable key/value pairs that do not apply to your container, set string values to an empty string, numeric values to zero, and Boolean values to false.

Contents

Java application analysis file

The Java application `analysis.json` file includes the following content:

Read-only data

- **Control fields** – Fields having to do with file creation, such as template version, and the file creation timestamp.
- **analysisInfo** – System dependencies for the application.

Configurable data

The `containerParameters` section contains the following fields:

- **imageRepository** (string, required) – The name of the repository where the application container image is stored.
- **imageTag** (string, required) – A tag for the build version of the application container image.
- **containerBaseImage** (string, required) – The base operating system image for the container build. *Must be an image name from your registry in the format <image name>[:<tag>]. Tag is optional if the repository supports "latest".*
- **appExcludedFiles** (array of strings) – Specific files and directories to exclude from the container build.
- **appSpecificFiles** (array of strings) – Specific files and directories to include in the container build.

- **applicationMode** (Boolean, required) – The approach that App2Container uses to determine which files to include in your container image. App2Container uses application mode (value=true) for supported application frameworks, and process mode (value=false) for all other configurations.

  You can override this value if necessary. For example, if your application is running on a supported framework, but App2Container did not recognize it and therefore assigned process mode, you can override the setting to use application mode instead.

  Application mode settings

  - *true* (application mode): For supported application frameworks, App2Container targets only the application files and dependencies that are needed for containerization, thereby minimizing the size of the resulting container image. Supported application frameworks include: Tomcat, TomEE, and JBoss (standalone mode).
  - *false* (process mode): If App2Container does not find a supported framework running on your application server, or if you have other dependent processes running on your server, App2Container takes a conservative approach to identifying dependencies. For process mode, all non-system files on the application server are included in the container image.

    Tip
    If your application container image includes unnecessary files, or is missing files that should be included, use the following parameters to make corrections:

    - To specify files to exclude from your application container image, use the `appExcludedFiles` parameter.
    - To add files that were missed, use the `appSpecificFiles` parameter.
- **logLocations** (array of strings) – Specific log files or log directories to be routed to `stdout`. This enables applications that write to log files on the host to be integrated with AWS services such as CloudWatch and Kinesis Data Firehose.
- **enableDynamicLogging** (Boolean, required) – Maps application logs to `stdout` as they are created. *If set to true, requires log directories to be entered in `logLocations`.*
- **dependencies** (array of strings) – A listing of all dependent processes or applications found for the application ID by the **analyze** command. You can remove specific dependencies to exclude them from the container.

The following example shows an `analysis.json` file for a Java application running on Linux.

```
{
 "a2CTemplateVersion": "",
       "createdTime": "",
       "containerParameters": {
              "_comment1": "*** EDITABLE: The below section can be edited according to the
 application requirements. Please see the analysisInfo section below for details discovered
 regarding the application. ***",
              "imageRepository": "java-tomcat-6e6f3a87",
              "imageTag": "latest",
              "containerBaseImage": "ubuntu:18.04",
              "appExcludedFiles": [],
              "appSpecificFiles": [],
              "applicationMode": true,
              "logLocations": [],
              "enableDynamicLogging": false,
              "dependencies": []
       },
       "analysisInfo": {
              "_comment2": "*** NON-EDITABLE: Analysis Results ***",
              "processId": 2065,
              "appId": "java-tomcat-6e6f3a87",
              "userId": "1000",
              "groupId": "1000",
              "cmdline": [
                     "/usr/bin/java",
                     "... list of commands",
                     "start"
              ],
              "osData": {
                     "BUG_REPORT_URL": "",
                     "HOME_URL": "",
                     "ID": "ubuntu",
                     "ID_LIKE": "debian",
                     "NAME": "Ubuntu",
                     "PRETTY_NAME": "Ubuntu 18.04.2 LTS",
                     "PRIVACY_POLICY_URL": "",
                     "SUPPORT_URL": "",
                     "UBUNTU_CODENAME": "",
                     "VERSION": "",
                     "VERSION_CODENAME": "",
                     "VERSION_ID": "18.04"
              },
              "osName": "ubuntu",
              "ports": [
                     {
                            "localPort": 8080,
                            "protocol": "tcp6"
                     },
                     {
                            "localPort": 8009,
                            "protocol": "tcp6"
                     },
                     {
                            "localPort": 8005,
                            "protocol": "tcp6"
                     }
              ],
              "Properties": {
                     "catalina.base": "<application directory>",
                     "catalina.home": "<application directory>",
                     "classpath": "<application directory>/bin/bootstrap.jar:... etc.",
                     "ignore.endorsed.dirs": "",
                     "java.io.tmpdir": "<application directory>/temp",
                     "java.protocol.handler.pkgs": "org.apache.catalina.webresources",
                     "java.util.logging.config.file": "<application directory>/conf/
logging.properties",
                     "java.util.logging.manager": "org.apache.juli.ClassLoaderLogManager",
                     "jdk.tls.ephemeralDHKeySize": "2048",
                     "jdkVersion": "11.0.7",
                     "org.apache.catalina.security.SecurityListener.UMASK": ""
              },
              "AdvancedAppInfo": {
                     "Directories": {
                            "base": "<application directory>",
                            "bin": "<application directory>/bin",
                            "conf": "<application directory>/conf",
                            "home": "<application directory>",
                            "lib": "<application directory>/lib",
                            "logConfig": "<application directory>/conf/logging.properties",
                            "logs": "<application directory>/logs",
                            "tempDir": "<application directory>/temp",
                            "webapps": "<application directory>/webapps",
                            "work": "<application directory>/work"
                     },
                     "distro": "java-tomee",
                     "flavor": "plume",
                     "jdkVersion": "11.0.7",
                     "version": "8.0.0"
              },
              "env": {
                     "HOME": "... Java Home directory",
                     "JDK_JAVA_OPTIONS": "",
                     "LANG": "C.UTF-8",
                     "LC_TERMINAL": "iTerm2",
                     "LC_TERMINAL_VERSION": "3.3.11",
                     "LESSCLOSE": "/usr/bin/lesspipe %s %s",
                     "LESSOPEN": "| /usr/bin/lesspipe %s",
                     "LOGNAME": "ubuntu",
                     "LS_COLORS": "",
                     "MAIL": "",
                     "OLDPWD": "",
                     "PATH": "... server PATH",
                     "PWD": "",
                     "SHELL": "/bin/bash",
                     "SHLVL": "1",
                     "SSH_CLIENT": "",
                     "SSH_CONNECTION": "",
                     "SSH_TTY": "",
                     "TERM": "",
                     "USER": "ubuntu",
                     "XDG_DATA_DIRS": "",
                     "XDG_RUNTIME_DIR": "",
                     "XDG_SESSION_ID": "1",
                     "_": "bin/startup.sh"
              },
              "cwd": "",
              "procUID": {
                     "euid": "1000",
                     "suid": "1000",
                     "fsuid": "1000",
                     "ruid": "1000"
              },
              "procGID": {
                     "egid": "1000",
                     "sgid": "1000",
                     "fsgid": "1000",
                     "rgid": "1000"
              },
              "userNames": {
                     "1000": "ubuntu"
              },
              "groupNames": {
                     "1000": "ubuntu"
              },
              "fileDescriptors": [
                     "<application directory>/logs/... log files",
                     "<application directory>/lib/... jar files",
                     "... etc.",
                     "/usr/lib/jvm/.../lib/modules"
              ],
              "dependencies": {}
          }
}
```

.NET application analysis file

The .NET application `analysis.json` file includes the following content:

Read-only data

- **Control fields** – Fields having to do with file creation, such as template version, and the file creation timestamp.

- **analysisInfo** – System dependencies for the application.

Configurable data

The `containerParameters` section contains the following fields:

- **containerBaseImage** (string, required) – The base operating system (OS) image for the container build. App2Container chooses the base image for your application container, based on the worker machine or application server OS where you run the containerization command. *Must be an image name from your registry in the format <image name>[:<tag>]. Tag is optional if the repository supports "latest".*
- **enableServerConfigurationUpdates** (Boolean, required) – Provides an option in the Dockerfile to restore the application configuration of the source server.
- **imageRepositoryName** (string, required) – The name of the repository where the application container image is stored.
- **imageTag** (string, required) – A tag for the build version of the application container image.
- **additionalExposedPorts** (array of numbers) – Additional port numbers that should be exposed inside of the application container.
- **appIncludedFiles** (array of strings) – Specific files and directories to include in the container build.
- **appExcludedFiles** (array of strings) – Specific files and directories to exclude from the container build.
- **enableLogging** (Boolean, required) – Enables dynamic logging, redirecting application logs to container `stdout`.
- **includedWebApps** (array of strings) – The application IDs for web applications running under the IIS site that should be included in the container image. *Applications must have been running in IIS during inventory and analysis.*
- **additionalApps** (array of strings) – For the `analysis.json` file that describes the root application in a complex Windows .NET application, these are the additional application or service components to include in the application container. You can include up to five additional application components in the array.

Examples

Your `analysis.json` file configuration can vary by the type of .NET application you are migrating, its dependencies, and whether you want it to run in a single container or in multiple containers. To see an example, expand the scenario that matches your configuration.

Simple Windows .NET application

The following example shows an `analysis.json` file for a simple .NET application running on Windows.

```
{
  "a2CTemplateVersion": "3.1",
  "createdTime": "",
  "containerParameters": {
    "_comment": "*** EDITABLE: The below section can be edited according to the application
 requirements. Please see the Analysis Results section further below for details discovered
 regarding the application. ***",
    "containerBaseImage": "mcr.microsoft.com/dotnet/framework/aspnet:4.7.2-
windowsservercore-ltsc2019",
    "enableServerConfigurationUpdates": true,
    "imageRepositoryName": "iis-smarts-51d2dbf8",
    "imageTag": "latest",
    "additionalExposedPorts": [
    ],
    "appIncludedFiles": [
    ],
    "appExcludedFiles": [
    ],
    "enableLogging": false,
    "additionalApps": [
    ]
  },
  "analysisInfo": {
    "_comment": "*** NON-EDITABLE: Analysis Results ***",
    "hostInfo": {
      "os": "...",
      "osVersion": "...",
      "osWindowsDirectory": "...",
      "arch": "..."
    },
    "appId": "iis-smarts-51d2dbf8",
    "appServerIp": "localhost",
    "appType": "IIS",
    "appName": "smarts",
    "ports": [
      {
        "localPort": 90,
        "protocol": "http"
      }
    ],
    "features": [
      "File-Services",
      "FS-FileServer",
      "Web-Http-Tracing",
      "Web-Basic-Auth",
      "Web-Digest-Auth",
      "Web-Url-Auth",
      "Web-Windows-Auth",
      "Web-ASP",
      "Web-CGI",
      "Web-Mgmt-Tools",
      "Web-Mgmt-Console",
      "Web-Scripting-Tools",
      "FS-SMB1",
      "User-Interfaces-Infra",
      "Server-Gui-Mgmt-Infra",
      "Server-Gui-Shell",
      "PowerShell-ISE"
    ],
    "appPoolName": "smarts",
    "poolIdentityType": "ApplicationPoolIdentity",
    "dotnetVersion": "v4.0",
    "iisVersion": "IIS 10.0",
    "sitePhysicalPath": "<IIS web root directory>\\smarts",
    "discoveredWebApps": [
    ],
    "reportPath": "<application output directory>\\iis-smarts-51d2dbf8\\report.txt",
    "isSiteUsingWindowsAuth": false,
    "serverBackupFile": "<application directory>\\Web Deploy Backups\\... backup zip file"
  }
}
```

Complex Windows .NET application running in one container

In this scenario, each application or service has its own `analysis.json` file, but the root application references the application ID for the service in the `additionalApps` array. This results in a single container that includes both the root application and the service when you run the **containerize** command.

- **Root application**

  The following example shows the `analysis.json` file for the root application.

```
{
 "a2CTemplateVersion": "1.0",
 "createdTime": "2021-06-25-05:18:24",
 "containerParameters": {
   "_comment": "*** EDITABLE: The below section can be edited according to the
application requirements. Please see the Analysis Results section further below for
details discovered regarding the application. ***",
   "containerBaseImage": "",
   "enableServerConfigurationUpdates": true,
   "imageRepositoryName": "iis-colormvciis-b69c09ab",
   "imageTag": "latest",
   "additionalExposedPorts": [
   ],
   "appIncludedFiles": [
   ],
   "appExcludedFiles": [
   ],
   "enableLogging": false,
   "includedWebApps": [
   ],
   "additionalApps": [
     "service-colorwindowsservice-69f90194"
   ]
 },
 "analysisInfo": {
   "_comment": "*** NON-EDITABLE: Analysis Results ***",
   "hostInfo": {
     "os": "Microsoft Windows Server 2019 Datacenter",
     "osVersion": "10.0.17763",
     "osWindowsDirectory": "C:\\Windows",
     "arch": "64-bit"
   },
   "appId": "iis-colormvciis-b69c09ab",
   "appServerIp": "localhost",
   "appType": "IIS",
   "appName": "colorMvcIIs",
   "ports": [
     {
       "localPort": 82,
       "protocol": "http"
     }
   ],
   "features": [
     "Web-Http-Redirect",
     "Web-Custom-Logging",
     "... etc."
   ],
   "appPoolName": "colorMVC",
   "poolIdentityType": "ApplicationPoolIdentity",
   "dotNetVersion": "v4.0",
   "iisVersion": "IIS 10.0",
   "sitePhysicalPath": "C:\\colorMvcIis",
   "discoveredWebApps": [
    ],
   "siteUsesWindowsAuth": false,
   "serverBackupFile": "<application directory>\\Web Deploy Backups\\... backup zip
file",
   "reportPath": "<application output directory>\\iis-colormvciis-b69c09ab\\report.txt"
 }
}
```

- **Windows service**

  The following example shows the `analysis.json` file for the Windows service that is included in the application container.

```
{
  "a2CTemplateVersion": "1.0",
  "createdTime": "2021-07-09-04:16:58",
  "containerParameters": {
    "_comment": "*** EDITABLE: The below section can be edited according to the
 application requirements. Please see the Analysis Results section further below for
 details discovered regarding the application. ***",
    "containerBaseImage": "",
    "enableServerConfigurationUpdates": true,
    "imageRepositoryName": "service-colorwindowsservice-69f90194",
    "imageTag": "latest",
    "additionalExposedPorts": [
    ],
    "appIncludedFiles": [
    ],
    "appExcludedFiles": [
    ],
    "enableLogging": false,
    "additionalApps": [
    ]
  },
  "analysisInfo": {
    "_comment": "*** NON-EDITABLE: Analysis Results ***",
    "hostInfo": {
      "os": "Microsoft Windows Server 2019 Datacenter",
      "osVersion": "10.0.17763",
      "osWindowsDirectory": "C:\\Windows",
      "arch": "64-bit"
    },
    "appId": "service-colorwindowsservice-69f90194",
    "appServerIp": "localhost",
    "appType": "service",
    "appName": "colorwindowsservice",
    "ports": [
      {
        "localPort": 33335,
        "protocol": "TCP"
      }
    ],
    "features": [
      "Web-Http-Redirect",
      "Web-Custom-Logging",
      "... etc."
    ],
    "serviceName": "colorwindowsservice",
    "serviceBinary": "ColorWindowsService.exe",
    "serviceDir": "C:\\COLORCODE\\colorservice-master\\ColorWindowsService\\bin\\Release\
\"
  }
}
```

Complex Windows .NET application running in multiple containers

In this scenario, each application or service has its own `analysis.json` file, and the `additionalApps` array is empty. To create two containers, run the **containerize** command twice – once for the root application and once for the service. For container orchestration, specify the service as a dependent application when you configure the `deployment.json` file for the root application.

- **Root application**

  The following example shows the `analysis.json` file for the root application.

```
{
 "a2CTemplateVersion": "1.0",
 "createdTime": "2021-06-25-05:18:24",
 "containerParameters": {
   "_comment": "*** EDITABLE: The below section can be edited according to the
application requirements. Please see the Analysis Results section further below for
details discovered regarding the application. ***",
   "containerBaseImage": "",
   "enableServerConfigurationUpdates": true,
   "imageRepositoryName": "iis-colormvciis-b69c09ab",
   "imageTag": "latest",
   "additionalExposedPorts": [
   ],
   "appIncludedFiles": [
   ],
   "appExcludedFiles": [
   ],
   "enableLogging": false,
   "includedWebApps": [
   ],
   "additionalApps": [
   ]
 },
 "analysisInfo": {
   "_comment": "*** NON-EDITABLE: Analysis Results ***",
   "hostInfo": {
     "os": "Microsoft Windows Server 2019 Datacenter",
     "osVersion": "10.0.17763",
     "osWindowsDirectory": "C:\\Windows",
     "arch": "64-bit"
   },
   "appId": "iis-colormvciis-b69c09ab",
   "appServerIp": "localhost",
   "appType": "IIS",
   "appName": "colorMvcIIs",
   "ports": [
     {
       "localPort": 82,
       "protocol": "http"
     }
   ],
   "features": [
     "Web-Http-Redirect",
     "Web-Custom-Logging",
     "... etc."
   ],
   "appPoolName": "colorMVC",
   "poolIdentityType": "ApplicationPoolIdentity",
   "dotNetVersion": "v4.0",
   "iisVersion": "IIS 10.0",
   "sitePhysicalPath": "C:\\colorMvcIis",
   "discoveredWebApps": [
    ],
   "siteUsesWindowsAuth": false,
   "serverBackupFile": "<application directory>\\Web Deploy Backups\\... backup zip
file",
   "reportPath": "<application output directory>\\iis-colormvciis-b69c09ab\\report.txt"
 }
}
```

- **Windows service**

The following example shows the `analysis.json` file for the Windows service that runs in a separate container.

```
{
  "a2CTemplateVersion": "1.0",
  "createdTime": "2021-07-09-04:16:58",
  "containerParameters": {
    "_comment": "*** EDITABLE: The below section can be edited according to the
 application requirements. Please see the Analysis Results section further below for
 details discovered regarding the application. ***",
    "containerBaseImage": "",
    "enableServerConfigurationUpdates": true,
    "imageRepositoryName": "service-colorwindowsservice-69f90194",
    "imageTag": "latest",
    "additionalExposedPorts": [
    ],
    "appIncludedFiles": [
    ],
    "appExcludedFiles": [
    ],
    "enableLogging": false,
    "additionalApps": [
    ]
  },
  "analysisInfo": {
    "_comment": "*** NON-EDITABLE: Analysis Results ***",
    "hostInfo": {
      "os": "Microsoft Windows Server 2019 Datacenter",
      "osVersion": "10.0.17763",
      "osWindowsDirectory": "C:\\Windows",
      "arch": "64-bit"
    },
    "appId": "service-colorwindowsservice-69f90194",
    "appServerIp": "localhost",
    "appType": "service",
    "appName": "colorwindowsservice",
    "ports": [
      {
        "localPort": 33335,
        "protocol": "TCP"
      }
    ],
    "features": [
      "Web-Http-Redirect",
      "Web-Custom-Logging",
      "... etc."
    ],
    "serviceName": "colorwindowsservice",
    "serviceBinary": "ColorWindowsService.exe",
    "serviceDir": "C:\\COLORCODE\\colorservice-master\\ColorWindowsService\\bin\\Release\
\"
  }
}
```

Note

For complex Windows .NET applications, you can also use a hybrid approach, with some components running together in a single container and other components running in separate containers.

Configuring container deployment

This topic contains information about the files that are used for configuring deployment of application containers.

Container deployment files

- deployment.json file (p. 39)

deployment.json file

When you run the **containerize (p. 77)** command, a `deployment.json` file is created for the application specified in the `--application-id` parameter. The **generate app-deployment** command uses this file, along with others, to generate application deployment artifacts. All of the fields in this file are configurable as needed so that you can customize your application container deployment before running the **generate app-deployment** command.

> **Important**
> The `deployment.json` file includes sections for both Amazon ECS and Amazon EKS. If your application is suitable for App Runner, there is a section for that too. You must set the Boolean value deployment flag for the section that matches your target container management service to **true**, and the other flags to **false**. The flag to deploy to Amazon ECS is `createEcsArtifacts`, the flag to deploy to Amazon EKS is `createEksArtifacts`, and the flag to deploy to App Runner is `createAppRunnerArtifacts`.

The application `deployment.json` file includes the following content. While all fields are configurable, the following fields should not be changed: `a2CTemplateVersion`, `applicationId`, and `imageName`. For key/value pairs that do not apply to your deployment, set string values to an empty string, numeric values to zero, and Boolean values to false.

- **exposedPorts** (array of objects, required) – An array of JSON objects representing the ports that should be exposed when the container is running. Each object consists of the following fields:
  - **localPort** (number) – A port to expose for container communication.
  - **protocol** (string) – The application protocol for the exposed port. For example, "http".
- **environment** (array of objects) – Environment variables to be passed on to the target container management deployment. For Amazon ECS deployments, the key/value pairs update the Amazon ECS task definition. For Amazon EKS deployments, the key/value pairs update the Kubernetes `deployment.yml` file.
- **ecrParameters** (object) – Contains parameters needed to register application container images in Amazon ECR.
  - **ecrRepoTag** (string, required) – The version tag to use for registering an application container image in Amazon ECR.
- **ecsParameters** (object) – Contains parameters needed for deployment to Amazon ECS. *The `createEcsArtifacts` parameter is always required. Other parameters in this section that are marked as required apply only to Amazon ECS deployment.*
  - **createEcsArtifacts** (Boolean, required) – A flag that indicates if you are targeting Amazon ECS for deployment.
  - **ecsFamily** (string, required) – An ID for the Amazon ECS family in the Amazon ECS task definition. We recommend setting this value to the application ID.
  - **cpu** (number, required*) – The hard limit for the number of vCPUs to present for the task. When the task definition is registered, the number of CPU units is determined by multiplying the number of vCPUs by 1024.

    ** This parameter is required for Linux containers, but is not supported for Windows containers.*

- **memory** (number or string, required*) – The hard limit of memory (in MiB) to present to the task. It can be expressed as an integer in the Amazon ECS task definition, using MiB, for example 1024, or as a string using GB, for example 1 GB. When the task definition is registered, a GB value is converted to an integer indicating the MiB.

  ** This parameter is required for Linux containers, but is not supported for Windows containers.*

  > **Note**
  > In the Amazon ECS task definition, task size consists of the `cpu` and `memory` parameters. The configuration for task size, in part, depends on what type of instance will host your tasks – an Amazon EC2 instance, or Fargate. For more information about setting the task size for you Amazon ECS task definition, see Task definition parameters in the *Amazon Elastic Container Service Developer Guide*.

- **dockerSecurityOption** (string) – For .NET applications, this is the gMSA Credspec location value for the Amazon ECS task definition.
- **enableCloudwatchLogging** (Boolean, required*) – A flag that sets the Amazon ECS task definition to enable CloudWatch logging for your Windows application container. *If set to true, the* `enableLogging` *field in the* `analysis.json` *file must have a valid value.*

  ** This parameter is required for Windows containers, but is not supported for Linux containers.*
- **publicApp** (Boolean, required) – A flag to configure the CloudFormation templates with a public endpoint for your application when it runs.
- **stackName** (string, required) – A name to use as a prefix to your CloudFormation stack Amazon Resource Name (ARN). We recommend using the application ID for this.
- **resourceTags** (array of objects) – Custom tags, expressed as key/value pairs that are added to resources during deployment. For Amazon ECS deployments, the key/value pairs update the Amazon ECS task definition.

  > **Note**
  > An example tag is generated when the `deployment.json` file is created. The example tag is ignored by default if it is not removed or changed prior to deployment.

- **reuseResources** (object) – Contains shared resource identifiers that can be used throughout your CloudFormation templates.
  - **vpcId** (string) – The VPC ID, if you want to bring your own VPC or to reuse an existing VPC that App2Container created for a prior deployment.
  - **reuseExistingA2cStack** (object) – Contains references that allow you to reuse AWS CloudFormation resources that App2Container has already created.
    - **cfnStackName** (string) – The name or ID (ARN) of the CloudFormation stack created with App2Container for the containerized application.
    - **microserviceUriPath** (string) – Used to create application forwarding rules in your load balancer.

      > **Note**
      > The load balancer does not strip off this prefix when it forwards traffic. Your application must be able to handle requests coming in with the prefix.
  - **sshKeyPairName** (string) – The name of the Amazon EC2 key pair to use for the instances that your container runs on.
  - **acmCertificateArn** (string) – The AWS Certificate Manager certificate ARN used to enable HTTPS on your application load balancer.

    > **Note**
    > The certificate can be imported or provisioned as follows:
    > - To import an IIS certificate into ACM, see How to import PFX-formatted certificates into AWS Certificate Manager using OpenSSL.
    > - To provision a certificate in ACM, see Issuing and Managing Certificates in the AWS Certificate Manager User Guide.

    This parameter is required if you use an HTTPS endpoint for your load balancer. For more information about ACM, see What is AWS Certificate Manager in the AWS Certificate Manager User Guide.

  - **gMSAParameters** (object) – Contains parameters used by the CloudFormation template to create gMSA-related artifacts for .NET applications.
    - **domainSecretsArn** (string) – The Secrets Manager arn for the domain credentials to join the Amazon ECS nodes to gMSA Active Directory.
    - **domainDNSName** (string) – The DNS Name of the gMSA Active Directory for Amazon ECS nodes to join.
    - **domainNetBIOSName** (string) – The Net BIOS name of the Active Directory for Amazon ECS nodes to join.
    - **createGMSA** (Boolean, required) – A flag to create a group Managed Service Account (gMSA) Active Directory security group and account, using the name supplied in the `gMSAName` field.
    - **gMSAName** (string) – The name of the gMSA Active Directory that the container should use for access.
  - **dependentApps** (array of objects) – For complex Windows applications, this array of JSON objects contains identifying details for dependent applications. App2Container does not generate this array. For complex Windows applications that incorporate dependent applications, you must add details to this array for each dependent application. You can include up to two dependent applications in the array.
    - **appId** (string, required) – The application ID that App2Container generated for this dependent application.
    - **privateRootDomain** (string, required) – The private domain name that's used for creating the hosted zone.
    - **dnsRecordName** (string, required) – The DNS record name of the application. This is combined with the privateRootDomain to construct the endpoint for the dependent application.
- **fireLensParameters** (object) – Contains parameters needed to use FireLens with your Linux application to route your application logs for Amazon ECS tasks. *The `enableFireLensLogging` parameter is always required. Other parameters in this section that are marked as required apply only when FireLens is used for log routing.*

    Note
    This section is not included for applications running on Windows.

  - **enableFireLensLogging** (Boolean, required) – A flag for using FireLens for Amazon ECS to configure application log routing for containers.
  - **logDestinations** (array of objects) – A list of unique target destinations for application log routing. If more than one destination is configured, App2Container creates a custom file that contains the FireLens configuration. Otherwise, the destination parameters are used directly in the Amazon ECS task definition and CloudFormation templates.
    - **service** (string) – The AWS service to route logs to. *Valid values are "cloudwatch", "firehose", and "kinesis".*
    - **regexFilter** (string) – A Ruby regular expression to match against log content to determine where to route the log.
    - **streamName** (string) – The name of the log delivery stream that will be created at the destination.
- **eksParameters** (object) – Contains parameters needed for deployment to Amazon EKS. *The `createEksArtifacts` parameter is always required. Other parameters in this section that are marked as required apply only to Amazon EKS deployments.*
  - **createEksArtifacts** (Boolean, required) – A flag that indicates if you are targeting Amazon EKS for deployment.
  - **stackName** (string, required) – A name to use as a prefix to your CloudFormation stack arn. We recommend using the application ID for this.

- **reuseResources** (object) – Contains shared resource identifiers that can be used throughout your CloudFormation templates.
  - **vpcId** (string) – The VPC ID, if you want to bring your own VPC or to reuse an existing VPC that App2Container created for a prior deployment. *If you are bringing a custom VPC, you must have two or more private subnets in two or more Availability Zones, and can optionally have two or more public subnets in the same two Availability Zones.*

    Note
    For each private subnet in the reused VPC, you must configure a route to the internet using a NAT gateway. For more information about cluster networking for Amazon EKS, see De-mystifying cluster networking for Amazon EKS worker nodes
  - **cfnStackName** (string) – The name or ID (ARN) of the CloudFormation stack created with App2Container for the containerized application.
  - **sshKeyPairName** (string) – The name of the Amazon EC2 key pair to use for the instances that your container runs on.
  - **resourceTags** (array of objects) – Custom tags, expressed as key/value pairs that are added to resources during deployment. For Amazon EKS deployments, the key/value pairs update the Kubernetes `deployment.yml` file.

    Note
    An example tag is generated when the `deployment.json` file is created. The example tag is ignored by default if it is not removed or changed prior to deployment.
- **gMSAParameters** (object) – Contains parameters used by the CloudFormation template to create gMSA-related artifacts for .NET applications.
  - **domainSecretsArn** (string) – The Secrets Manager arn for the domain credentials to join the Amazon EKS nodes to gMSA Active Directory.
  - **domainDNSName** (string) – The DNS Name of the gMSA Active Directory for Amazon EKS nodes to join.
  - **domainNetBIOSName** (string) – The Net BIOS name of the Active Directory for Amazon EKS nodes to join.
  - **createGMSA** (Boolean, required) – A flag to create a group Managed Service Account (gMSA) Active Directory security group and account, using the name supplied in the `gMSAName` field.
  - **gMSAAccountName** (string) – The name of the gMSA Active Directory that the container should use for access.
- **dependentApps** (array of objects) – For complex Windows applications, this array of JSON objects contains identifying details for dependent applications. App2Container does not generate this array. For complex Windows applications that incorporate dependent applications, you must add details to this array for each dependent application. You can include up to two dependent applications in the array.
  - **appId** (string, required) – The application ID that App2Container generated for this dependent application.
  - **privateRootDomain** (string, required) – The private domain name that's used for creating the hosted zone.
  - **dnsRecordName** (string, required) – The DNS record name of the application. This is combined with the privateRootDomain to construct the endpoint for the dependent application.

- **appRunnerParameters** (object) – Contains parameters needed for deployment of Linux applications to an AWS App Runner environment. *The `createAppRunnerArtifacts` parameter is always required. Other parameters in this section that are marked as required apply only to App Runner deployments.*

  Note
  This section is not included for applications running on Windows.
  - **createAppRunnerArtifacts** (Boolean, required) – A flag that indicates if you are targeting App Runner for deployment.

- **stackName** (string, required) – The name of the AWS CloudFormation stack. *We recommend including the application ID in the stack name.*
- **serviceName** (string, required) – The name of the service in App Runner. *We recommend using the application ID for the service name.*
- **autoDeploymentsEnabled** (Boolean, required) – If set to true, an update to the Amazon ECR repository also updates the service in App Runner. If set to false, you can manually update the service using the App Runner console or API, or `apprunner` commands in the AWS CLI.
- **resourceTags** (array of objects) – Custom tags, expressed as key/value pairs that are added to resources during deployment. For App Runner deployments, the key/value pairs update both of the resources that are created in the `apprunner.yml` AWS CloudFormation template.

  Note
  An example tag is generated when the `deployment.json` file is created. The example tag is ignored by default if it is not removed or changed prior to deployment.

Note
When the **containerize** command runs, it determines if your application is suitable for App Runner, and adds `appRunnerParameters` to the `deployment.json` file if it is. If your application is not suitable for App Runner, the `appRunnerParameters` are left out.

Examples

Linux Java application deployed to Amazon ECS

The following example shows a `deployment.json` file for a Java application running on Linux with default settings to deploy to an Amazon ECS environment.

```
{
	"a2CTemplateVersion": "3.1",
	"applicationId": "java-tomcat-6e6f3a87",
	"imageName": "java-tomcat-6e6f3a87",
	"exposedPorts": [
		{
			"localPort": 8080,
			"protocol": "tcp6"
		},
		{
			"localPort": 8009,
			"protocol": "tcp6"
		},
		{
			"localPort": 8005,
			"protocol": "tcp6"
		}
	],
	"environment": [],
	"ecrParameters": {
		"ecrRepoTag": "latest"
	},
	"ecsParameters": {
		"createEcsArtifacts": true,
		"ecsFamily": "java-tomcat-6e6f3a87",
		"cpu": 2,
		"memory": 4096,
		"dockerSecurityOption": "",
		"enableCloudwatchLogging": false,
		"publicApp": true,
		"stackName": "app2container-java-tomcat-6e6f3a87-ECS",
		"resourceTags": [
			{
```

```
                             "key": "example-key",
                             "value": "example-value"
                      }
               ],
               "reuseResources": {
                      "vpcId": "",
                      "reuseExistingA2cStack": {
                             "cfnStackName": "",
                             "microserviceUrlPath": ""
                      },
                      "sshKeyPairName": "",
                      "acmCertificateArn": ""
               },
               "gMSAParameters": {
                      "createGMSA": false,
                      "domainSecretsArn": "",
                      "domainDNSName": "",
                      "domainNetBIOSName": "",
                      "gMSAName": "",
                      "ADSecurityGroupName": ""
               },
               "deployTarget": "FARGATE"
        },
        "fireLensParameters": {
              "enableFireLensLogging": true,
              "logDestinations": [
                     {
                             "service": "cloudwatch",
                             "matchRegex": "^.*INFO.*$",
                             "streamName": "Info"
                     },
                     {
                             "service": "cloudwatch",
                             "matchRegex": "^.*WARN.*$",
                             "streamName": "Warn"
                     }
              ]
        },
        "eksParameters": {
               "createEksArtifacts": false,
               "applicationName": "",
               "stackName": "java-tomcat-6e6f3a87",
               "reuseResources": {
                      "vpcId": "",
                      "cfnStackName": "",
                      "sshKeyPairName": ""
               },
               "gMSAParameters": {
                      "createGMSA": false,
                      "domainSecretsArn": "",
                      "domainDNSName": "",
                      "domainNetBIOSName": "",
                      "gMSAAccountName": "",
                      "ADSecurityGroupName": ""
               }
        },
        "appRunnerParameters": {
              "createAppRunnerArtifacts": false,
              "stackName": "a2c-java-tomcat-6e6f3a87-AppRunner",
              "autoDeploymentsEnabled": true,
              "resourceTags": [
                    {
                          "key": "example-key",
                          "value": "example-value"
                    }
              ]
        }
}
```

Complex Windows .NET application deployed to Amazon ECS

For complex Windows .NET web applications that consist of a root application and up to two dependent applications, each application is defined separately, and each has its own deployment.json file.

The following example shows the `deployment.json` file for the root application in a complex .NET web service running on Windows, followed by `deployment.json` files for the two dependent applications that it refers to. The applications are deployed to an Amazon ECS environment running together in the same VPC.

- **Root application example**

```
{
        "a2CTemplateVersion": "3.1",
        "applicationId": "iis-smarts-51d2dbf8",
        "imageName": "iis-smarts-51d2dbf8",
        "exposedPorts": [
                {
                        "localPort": 8080,
                        "protocol": "http"
                }
        ],
        "environment": [],
        "ecrParameters": {
                "ecrRepoTag": "latest"
        },
        "ecsParameters": {
                "createEcsArtifacts": true,
                "ecsFamily": "iis-smarts-51d2dbf8",
                "cpu": 2,
                "memory": 4096,
                "dockerSecurityOption": "",
                "enableCloudwatchLogging": false,
                "publicApp": true,
                "stackName": "iis-smarts-51d2dbf8-ECS",
                "resourceTags": [
                        {
                                "key": "example-key",
                                "value": "example-value"
                        }
                ],
                "reuseResources": {
                        "vpcId": "vpc-0abc1defa2345b67c",
                        "reuseExistingA2cStack": {
                                "cfnStackName": "",
                                "microserviceUrlPath": ""
                        },
                        "sshKeyPairName": "",
                        "acmCertificateArn": ""
                },
                "gMSAParameters": {
                        "createGMSA": false,
                        "domainSecretsArn": "",
                        "domainDNSName": "",
                        "domainNetBIOSName": "",
                        "gMSAName": ""
                },
                "dependentApps" : [
                  {
                      "appId":"iis-appB-ab800cde",
                      "privateRootDomain": "dependent-app1.test1.com",
                    "dnsRecordName":"appB"
                },
                {
                    "appId":"service-appC-9fghi90j",
                    "privateRootDomain": "dependent-app2.test1.com",
                    "dnsRecordName":"appC"
                }
            ]
        },
        "eksParameters": {
                "createEksArtifacts": false,
                "applicationName": "",
                "stackName": "iis-smarts-51d2dbf8",
                "reuseResources": {
                        "vpcId": "",
                        "reuseExistingA2cStack": {
                                "cfnStackName": "",
                                "microserviceUrlPath": ""
                        },
                        "sshKeyPairName": "",
                        "acmCertificateArn": ""
                },
                "gMSAParameters": {
                        "createGMSA": false,
                        "domainSecretsArn": "",
                        "domainDNSName": "",
                        "domainNetBIOSName": "",
                        "gMSAAccountName": ""
                },
                "dependentApps" : []
        }
}
```

- **Dependent application B**

```
{
        "a2CTemplateVersion": "3.1",
        "applicationId": "iis-appB-ab800cde",
        "imageName": "iis-appB-ab800cde",
        "exposedPorts": [
                {
                        "localPort": 8080,
                        "protocol": "http"
                }
        ],
        "environment": [],
        "ecrParameters": {
                "ecrRepoTag": "latest"
        },
        "ecsParameters": {
                "createEcsArtifacts": true,
                "ecsFamily": "iis-appB-ab800cde",
                "cpu": 2,
                "memory": 4096,
                "dockerSecurityOption": "",
                "enableCloudwatchLogging": false,
                "publicApp": true,
                "stackName": "iis-appB-ab800cde-ECS",
                "resourceTags": [
                        {
                                "key": "example-key",
                                "value": "example-value"
                        }
                ],
                "reuseResources": {
                        "vpcId": "vpc-0abc1defa2345b67c",
                        "reuseExistingA2cStack": {
                                "cfnStackName": "",
                                "microserviceUrlPath": ""
                        },
                        "sshKeyPairName": "",
                        "acmCertificateArn": ""
                },
                "gMSAParameters": {
                        "createGMSA": false,
                        "domainSecretsArn": "",
                        "domainDNSName": "",
                        "domainNetBIOSName": "",
                        "gMSAName": ""
                },
                "dependentApps" : []
        },
        "eksParameters": {
                "createEksArtifacts": false,
                "applicationName": "",
                "stackName": "iis-appB-ab800cde",
                "reuseResources": {
                        "vpcId": "",
                        "reuseExistingA2cStack": {
                                "cfnStackName": "",
                                "microserviceUrlPath": ""
                        },
                        "sshKeyPairName": ""
                },
                "gMSAParameters": {
                        "createGMSA": false,
                        "domainSecretsArn": "",
                        "domainDNSName": "",
                        "domainNetBIOSName": "",
                        "gMSAAccountName": ""
                },
                "dependentApps" : []
        }
}
```

- **Dependent application C**

```
{
        "a2CTemplateVersion": "3.1",
        "applicationId": "service-appC-9fghi90j",
        "imageName": "service-appC-9fghi90j",
        "exposedPorts": [
                {
                        "localPort": 8080,
                        "protocol": "http"
                }
        ],
        "environment": [],
        "ecrParameters": {
                "ecrRepoTag": "latest"
        },
        "ecsParameters": {
                "createEcsArtifacts": true,
                "ecsFamily": "service-appC-9fghi90j",
                "cpu": 2,
                "memory": 4096,
                "dockerSecurityOption": "",
                "enableCloudwatchLogging": false,
                "publicApp": true,
                "stackName": "service-appC-9fghi90j-ECS",
                "resourceTags": [
                        {
                                "key": "example-key",
                                "value": "example-value"
                        }
                ],
                "reuseResources": {
                        "vpcId": "vpc-0abc1defa2345b67c",
                        "reuseExistingA2cStack": {
                                "cfnStackName": "",
                                "microserviceUrlPath": ""
                        },
                        "sshKeyPairName": "",
                        "acmCertificateArn": ""
                },
                "gMSAParameters": {
                        "createGMSA": false,
                        "domainSecretsArn": "",
                        "domainDNSName": "",
                        "domainNetBIOSName": "",
                        "gMSAName": ""
                },
                "dependentApps" : []
        },
        "eksParameters": {
                "createEksArtifacts": false,
                "applicationName": "",
                "stackName": "service-appC-9fghi90j",
                "reuseResources": {
                        "vpcId": "",
                        "reuseExistingA2cStack": {
                                "cfnStackName": "",
                                "microserviceUrlPath": ""
                        },
                        "sshKeyPairName": ""
                },
                "gMSAParameters": {
                        "createGMSA": false,
                        "domainSecretsArn": "",
                        "domainDNSName": "",
                        "domainNetBIOSName": "",
                        "gMSAAccountName": ""
                },
                "dependentApps" : []
        }
}
```

Configuring container pipelines

This topic contains information about the files that are used to configure and deploy application container pipelines using AWS CodeStar services.

Pipeline configuration files

- pipeline.json file (p. 48)

pipeline.json file

When you run the **generate app-deployment (p. 82)** command, a `pipeline.json` file is created for the application specified in the `--application-id` parameter. The **generate pipeline** command uses this file, along with others, to generate pipeline deployment artifacts. All of the fields in this file are configurable as needed to customize your application container pipeline before running the **generate pipeline** command .

The application `pipeline.json` file includes the following content. While all fields are configurable, the `a2CTemplateVersion` field should not be changed. For key/value pairs that do not apply to your pipeline, set string values to an empty string, numeric values to zero, and Boolean values to false.

- **sourceInfo** (object) – Contains JSON objects for AWS CodeStar configuration.
  - **CodeCommit** (object) – Contains parameters needed for AWS CodeCommit configuration.
    - **repositoryName** (string, required) – The name of the AWS CodeCommit repository to use or create.
    - **branch** (string, required) – The name of the code branch in the AWS CodeCommit repository to commit to.
- **imageInfo** (object) – Contains parameters needed for Amazon ECR configuration.
  - **image** (string, required) – The full repository name of the application container image to store in Amazon ECR. *Must be in the format <application ID>.<repository name>:<tag>.*
- **releaseInfo** (object) – Contains JSON objects with parameters needed to create a pipeline for your target deployment environments.
  - **ECS | EKS | AppRunner** (object) – Contains JSON objects representing the environments to target for deployment. The key name specifies the container management service that you are targeting for your application container pipeline. *Key must be "ECS", "EKS", or "AppRunner". At least one of the pipeline environments must be enabled.*
    - **beta** (object) –
      - **clusterName** (string, required*) – The name of the Amazon ECS or Amazon EKS cluster to set up in the AWS CloudFormation stack.
      - **serviceName** (string, required*) – The name of the Amazon ECS service to set up in the AWS CloudFormation stack.

        ** Applies only to Amazon ECS pipelines.*
      - **enabled** (Boolean, required) – A flag indicating whether a beta environment should be configured.

        Note
        Beta environments are not supported for App Runner.
    - **prod** (object) –
      - **clusterName** (string, required*) – The name of the Amazon ECS or Amazon EKS cluster to set up in the AWS CloudFormation stack.

        ** Does not apply to App Runner.*
      - **serviceName** (string, required*) – The name of the Amazon ECS service to set up in the AWS CloudFormation stack.

        ** Applies only to Amazon ECS pipelines.*
      - **enabled** (Boolean, required) – A flag indicating whether a prod environment should be configured.

The following example shows a `pipeline.json` file for a Java application running on Linux. As you can see in the file, the application is running in a beta environment, and there is no prod environment configured yet.

```
{
    "a2CTemplateVersion": "3.1",
    "sourceInfo": {
        "CodeCommit": {
            "repositoryName": "app2container-java-tomcat-6e6f3a87-ecs",
            "branch": "master"
        }
    },
    "releaseInfo": {
        "ECS": {
            "beta": {
                "clusterName": "app2container-java-tomcat-6e6f3a87-ECS-Cluster",
                "serviceName": "app2container-java-tomcat-6e6f3a87-ECS-LBWebAppStack-etc.",
                "enabled": true
            },
            "prod": {
                "clusterName": "",
                "serviceName": "",
                "enabled": false
            }
        }
    }
}
```

Product and service integrations for AWS App2Container

AWS App2Container integrates with an array of AWS services and partner products and services. Use the information in the following sections to help you configure App2Container to integrate with the products and services that you use.

Integrations

Deploy application containers to AWS App Runner with AWS App2Container

AWS App Runner is an AWS service that provides a way for existing container images or source code to run directly as web services in AWS. App Runner uses Fargate as its underlying environment, but has its own management layer on top. With App Runner, you can access your application through an assigned web service URL, via HTTP requests.

Considerations for deploying to App Runner using App2Container:

- App Runner is not available in all Regions. To see the Regions and service endpoints for App Runner, refer to App Runner Service endpoints in the *AWS General Reference*.
- Resources that are created by App Runner reside in the multi-tenant App Runner service account. With other container hosting services, you might access resources such as an Amazon EC2 instance that your container runs on, or an Amazon EBS volume attached to your container instance, using the standard access methods for those resources directly. With App Runner you access resources that App Runner creates for your application through the App Runner service, using the App Runner console, API, SDKs, or by using **apprunner** commands in the AWS CLI.
- App Runner supports continuous integration and deployment from the Amazon ECR repository that App2Container creates on your behalf. When continuous deployment is configured, an update to the container image in the Amazon ECR repository automatically initiates an update in App Runner.

  You can turn this on or off in the `deployment.json` file. For more information, see Configure deployment (p. 39).
- App Runner integrates with Amazon CloudWatch and AWS CloudTrail to provide logging and monitoring support for your application. App Runner creates the following log groups for each App Runner service:
  - An application group, which contains `stdout` from your containers.
  - A service group, which contains high-level logs from App Runner to notify you about service-related events, such as new deployments or health check failures.

  These logs can also be viewed from the App Runner console, or by using the App Runner API, SDKs, or by using **apprunner** commands in the AWS CLI.
- App Runner enforces limits for the application containers that it hosts, such as the number of concurrent requests, the size of the application, and the amount of memory it can use. To learn more about Service Quotas for App Runner, see App Runner Service quotas in the *AWS General Reference*.
- Application state is not guaranteed to be maintained between requests.

For more information about using App Runner to host your application container, see What is AWS App Runner in the *AWS App Runner Developer Guide*.

Prerequisites

To configure an App Runner integration for your application container with App2Container, your application must meet the following criteria:

- Your application runs on Linux. [*Windows applications are not currently supported.*]
- Your application meets all of the requirements that are listed in the Supported applications (p. 3) section for Linux.
- Your application container size is less than 3 GB.
- Your application must not be dependent on background processing. App Runner heavily throttles container CPU when requests are not actively being processed.

App Runner integration for App2Container workflow

Setting up application containers for hosting in App Runner integrates smoothly with the App2Container workflow. All of The initial steps for App2Container are the same for all applications:

1. Install and set up the App2Container environment, as described in the Setting up AWS App2Container (p. 14) section.
2. Complete the initialization phase for your App2Container environment with the **init** command, and **remote configure**, if applicable. To learn more about what is included in all of the App2Container containerization phases, see the Command reference (p. 73).
3. Complete the analyze phase for each application that you want to containerize.
   - If you are running commands directly on application servers, use the **inventory** and **analyze** commands.
   - If you are running a remote workflow on a worker machine, use the **remote inventory** and **remote analyze** commands.
4. Integration begins with the containerization step.
   - When you run the **containerize** command, App2Container generates the `deployment.json` file, which provides configurable parameters for all supported container hosting service options that could apply to your application container.
   - Parameters for Amazon ECS and Amazon EKS are always included. Parameters for App Runner are also included if your application container meets the App2Container criteria for hosting in App Runner (see Prerequisites (p. 52)).
   - Each container hosting service has its own section in the `deployment.json` file, and each section has a flag to indicate which container hosting service is the destination for your application container. Only one section can have its flag set to **true** – all others must be set to **false**.

     Amazon ECS is configured as the destination by default, but if your application is suitable for App Runner, you can set the `createAppRunnerArtifacts` flag in the `appRunnerParameters` section to **true**, and the `createEcsArtifacts` flag in the `ecsParameters` section to **false**. For more information about configuring the `deployment.json` file, see Configure deployment (p. 39).
5. The deployment step generates an AWS CloudFormation template and `pipeline.json` file that are targeted for the App Runner container hosting service, based on the settings in the `deployment.json` file, where the `createAppRunnerArtifacts` flag is set to **true**.
   - When you run the **generate app-deployment** command, App2Container validates the properties in the `deployment.json` file, and pushes the container image to Amazon ECR. This is the standard workflow.

- The command generates an AWS CloudFormation template for App Runner deployment that contains the IAM role that App Runner uses to pull your application container images from Amazon ECR, and the App Runner service definition.
- The command generates the `pipeline.json` file to support creating a pipeline to deploy updates to your application container in Amazon ECR.
- If you use the `--deploy` option for the **generate app-deployment** command, App2Container deploys the AWS CloudFormation stack that creates the App Runner service for the containerized application, using the configuration values in the AWS CloudFormation template that it generates. To customize the configuration, run the command without the `--deploy` option, and then manually deploy using the AWS CLI when you are ready.

6. The pipeline step generates an AWS CloudFormation template for the pipeline that is targeted for the App Runner container hosting service, based on the settings in the `pipeline.json` file.
   - When you run the **generate pipeline** command, App2Container validates the properties in the `pipeline.json` file, and verifies that initial deployment to App Runner has been completed, and your application is active.
   - The command generates an AWS CloudFormation template to create a two-step pipeline:
     a. **Code commit** – Creates or updates an AWS CodeCommit repository that contains the Dockerfile and application artifacts that are required to create your application container image.
     b. **Code build** – Builds the Docker image for your application container, and pushes the updated image to the Amazon ECR repository that you configured for your application.
     c. If you use the `--deploy` option for the **generate pipeline** command, App2Container deploys the pipeline with the configuration values in the AWS CloudFormation template it generates. To customize the configuration, run the command without the `--deploy` option, and then manually deploy using the AWS CLI when you are ready.

       Note
       If you have automatic deployments configured for App Runner, an update to your application container image in Amazon ECR automatically kicks off an update for your application in App Runner.
       To configure automatic deployments, use the following settings in the `deployment.json` file:
       - Set `autoDeploymentsEnabled` to **true** to automatically deploy updates to App Runner when you deploy updates to Amazon ECR. *This is the default setting.*
       - Set `autoDeploymentsEnabled` to **false** if you want to update App Runner manually, using the App Runner service console, API, SDKs, or AWS CLI.

Setting up FireLens log file routing for containers with AWS App2Container

When you set up your application containers to use FireLens for Amazon ECS you can route your application logs to CloudWatch, Kinesis Data Streams, or Kinesis Data Firehose for log storage and analytics. After you have configured the FireLens settings in your application analysis and deployment JSON files, App2Container creates the artifacts that you need to deploy your application to Amazon EC2 or AWS Fargate. This includes:

- Creation of initial Kinesis Data Streams or Kinesis Data Firehose streams, if applicable
- Creation of an IAM role with the permissions needed to enable FireLens log routing to the destinations that you have specified
- Deployment artifacts that contain the FireLens parameters that you specified in your JSON configuration files, including the Amazon ECS task definition and AWS CloudFormation template files

For more information about using FireLens for Amazon ECS, see Custom log routing in the *Amazon Elastic Container Service Developer Guide*.

> **Note**
> App2Container initially supports FireLens log file routing for Amazon ECS for Linux containers only.

Contents

FireLens log routing for Linux

Before starting these configuration steps, you should have an understanding of the App2Container containerization phases – Initialize, Analyze, Transform, and Deploy. To learn more about the containerization phases and the commands that run during each phase, see the App2Container command reference (p. 73) in this user guide.

Follow these steps to set up log file routing with FireLens for Amazon ECS for your Linux application containers:

FireLens configuration

Prerequisites

Prior to setting up FireLens log routing for your application, you must have completed the following prerequisites:

- You have root access on the application server (and worker machine, if using).
- You successfully completed all of the steps from the Setting up AWS App2Container (p. 14) section of this user guide.
- You have initialized the App2Container environment by successfully running the **init (p. 92)** command.
- The application must be running on the application server, and must have a valid application ID assigned by the **inventory (p. 94)** command.

Step 1: Identify log locations for the container

Run the **analyze (p. 76)** command for your application, and then update the following parameters in your `analysis.json` file:

- Update the `logLocations` array to include a list of log files or directory locations where log files can be picked up for routing with FireLens.
- Set the `enableDynamicLogging` parameter to *true* to map application logs to `stdout` as they are created. If your application appends to specific log files such as `info.log` or `error.log`, set the `enableDynamicLogging` parameter to *false*.

The `analysis.json` file is stored in the application folder, for example: `/root/app2container/java-tomcat-9e8e4799`. For more information on `analysis.json` fields and configuration, see Configuring application containers (p. 29) in the **Configuring your application** section of this user guide.

Example:

The following example shows container parameters in the `analysis.json` file for logging.

```
"containerParameters": {
    ...
    "logFiles": ["error.log", "info.log"],
    "logDirectory": "/var/app/logs/",
    "logLocations": ["error.log", "info.log", "/var/app/logs/"],
    "enableDynamicLogging": true,
    ...
},
```

Step 2: Configure log deployment parameters

Run the **containerize (p. 77)** command, and then edit the `deployment.json` file to set the `fireLensParameters`. The `deployment.json` file is stored in the application folder, for example: `/root/app2container/java-tomcat-9e8e4799`.

There must be at least one valid log destination defined for the `logDestinations` array, with valid values for each of the parameters it contains. For more information on `deployment.json` fields and configuration, including how to target deployment to AWS Fargate with the `deployTarget` parameter, see Configuring container deployment (p. 39) in the **Configuring your application** section of this user guide.

- Set `enableFirelensLogging` to *true*.
- Configure one or more valid `logDestinations` as follows:
  - **service** – the AWS service to route logs to. *Valid values are "cloudwatch", "firehose", and "kinesis".*
  - **regexFilter** (string) – the pattern to match against log content using a Ruby regular expression to determine where to route the log.

    Note
    Ruby regular expressions begin and end with a forward slash, with the pattern to match specified in between the slashes. Patterns often begin with a caret (^), which starts matching at the beginning of the line, and end with a dollar sign ($), which stops matching at the end of the line.
    The `regexFilter` parameter in the `deployment.json` file represents only the matching pattern. Be sure to test your matching pattern using one of the many applications available for your desktop or online, such as Rubular. For more information about Ruby regular expressions, see Mastering Ruby Regular Expressions.
  - **streamName** (string) – the name of the log delivery stream that will be created at the destination.

Examples:

The following example shows FireLens parameters in the `deployment.json` file for logging to a single destination - CloudWatch – using a Ruby regular expression.

```
"fireLensParameters": {
    "enableFireLensLogging": true,
    "logDestinations": [
         {
                "service": "cloudwatch",
                "regexFilter": "^.*INFO.*$",
                "streamName": "Info"
            }
    ]
},
```

This example shows FireLens parameters in the `deployment.json` file for logging to a single destination – Kinesis Data Firehose – using a Ruby regular expression.

```
"fireLensParameters": {
    "enableFireLensLogging": true,
    "logDestinations": [
       {
              "service": "firehose",
              "regexFilter": "^.*INFO.*$",
              "streamName": "Info"
       }
    ]
},
```

This example shows FireLens parameters in the `deployment.json` file for routing separate log files to different destinations in CloudWatch, using Ruby regular expressions.

```
"fireLensParameters": {
    "enableFireLensLogging": true,
    "logDestinations": [
       {
              "service": "cloudwatch",
              "regexFilter": "^.*INFO.*$",
              "streamName": "Info"
       },
       {
              "service": "cloudwatch",
              "regexFilter": "^.*WARNING.*$",
              "streamName": "Warning"
       }
    ]
},
```

Step 3: Validate deployment artifacts

The last step before deployment is to ensure that your Amazon ECS task definitions and AWS CloudFormation templates are configured as expected after running the **generate app-deployment** command, and that your log destinations were created, if applicable.

Note

- Deployment artifacts are stored in the Amazon ECS or Amazon EKS deployment folder within the application folder that App2Container created for you. For example: `/root/app2container/java-tomcat-9e8e4799`
- If you are routing to CloudWatch, your routing destination is not created prior to deployment.

1. Run the **generate app-deployment (p. 82)** command to generate container deployment artifacts.
2. Verify that the Amazon ECS task definitions include the parameters that you specified and that the values are correct. For an example of FireLens parameters in an Amazon ECS task definition, see Example: Amazon ECS task definition FireLens parameters (p. 57)
3. Verify that the AWS CloudFormation template includes the parameters that you specified and that the values are correct. For an example of FireLens parameters in an AWS CloudFormation

template, expand the following section: Example: AWS CloudFormation template FireLens parameters (p. 58)

4. If you are routing logs to Kinesis Data Streams or Kinesis Data Firehose, verify that the streams have been created for you by using the AWS Management Console.

   a. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.
   b. From the Amazon Kinesis dashboard, choose **Data streams** or **Delivery streams** from the navigation pane.
   c. Verify that your stream **Status** is `Active`.

Example: Amazon ECS task definition FireLens parameters

This example shows excerpts from an Amazon ECS task definition file that was generated for logging to CloudWatch.

```
"executionRoleArn": arn:aws:iam::
 &lt;YOUR_ACCOUNT_ID&gt;:role/A2CEcsFirelensRole",
"containerDefinitions": [
    {
      ...
      "logConfiguration": {
        "logDriver": "awsfirelens",
        "secretOptions": null,
        "options": {
          "include-pattern": "^.*INFO.*$",
          "log_group_name": "java-tomcat-c770eed9-logs",
          "log_stream_name": "java-tomcat-c770eed9-Info",
          "auto_create_group": "true",
          "region": "us-east-1",
          "Name": "cloudwatch"
        }
      },
      ...
      "name": "java-tomcat-c770eed9"
    },
    {
      "dnsSearchDomains": null,
      "environmentFiles": null,
      "logConfiguration": {
        "logDriver": "awslogs",
        "secretOptions": null,
        "options": {
          "awslogs-group": "/ecs/containerization",
          "awslogs-region": "us-east-1",
          "awslogs-create-group": "true",
          "awslogs-stream-prefix": "firelens"
        }
      },
      ...
      "firelensConfiguration": {
        "type": "fluentbit",
        "options": null
      },
      ...
      "name": "java-tomcat-c770eed9-log-router"
    }
  ],
  ...
  "taskRoleArn": arn:aws:iam::
 &lt;YOUR_ACCOUNT_ID&gt;:role/A2CEcsFirelensRole",
  "compatibilities": [
    "EC2",
    "FARGATE"
  ],
  ...
  "requiresAttributes": [
    {
      "targetId": null,
      "targetType": null,
      "value": null,
      "name": "ecs.capability.execution-role-awslogs"
    },
    ...
    {
      "targetId": null,
      "targetType": null,
      "value": null,
      "name": "com.amazonaws.ecs.capability.logging-driver.awsfirelens"
    },
    ...
    {
      "targetId": null,
      "targetType": null,
      "value": null,
      "name": "com.amazonaws.ecs.capability.logging-driver.awslogs"
    },
    ...
    {
      "targetId": null,
      "targetType": null,
      "value": null,
      "name": "ecs.capability.firelens.fluentbit"
    }
  ],
```

Example: AWS CloudFormation template FireLens parameters

This example shows excerpts from an AWS CloudFormation template file that was generated for logging to CloudWatch.

```
Metadata:
  AWS::CloudFormation::Interface:
    ParameterGroups:
...
- Label:
          default: Logging Parameters for the application being deployed, check ecs-lb-
webapp.yml for usage
        Parameters:
          - TaskLogDriver
          - MultipleDests
          - SingleDestName
          - IncludePattern
          - LogGrpName
          - LogStrmName
          - AutoCrtGrp
          - FirehoseStream
          - KinesisStream
          - KinesisAppendNewline
          - FirelensName
          - FirelensImage
          - ConfigType
          - ConfigPath
          - UsingCloudwatchLogs
          - UsingFirehoseLogs
          - UsingKinesisLogs
```

```
...
Parameters:
...
# Firelens Parameters for the application being deployed
  TaskLogDriver:
    Type: String
    Default: awsfirelens
  MultipleDests:
    Type: String
    AllowedValues: [true, false]
    Default: false
  SingleDestName:
    Type: String
    Default: cloudwatch
  IncludePattern:
    Type: String
    Default: ^.*INFO.*$
  LogGrpName:
    Type: String
    Default: java-tomcat-c770eed9-logs
  LogStrmName:
    Type: String
    Default: java-tomcat-c770eed9-Info
  AutoCrtGrp:
    Type: String
    Default: true
  FirehoseStream:
    Type: String
    Default: ""
  KinesisStream:
    Type: String
    Default: ""
  KinesisAppendNewline:
    Type: String
    Default: ""
  FirelensName:
    Type: String
    Default: java-tomcat-c770eed9-log-router
  FirelensImage:
    Type: String
    Default: 906394416424.dkr.ecr.us-east-1.amazonaws.com/aws-for-fluent-bit:latest
  ConfigType:
    Type: String
    Default: ""
  ConfigPath:
    Type: String
    Default: ""
  UsingCloudwatchLogs:
    Type: String
    Default: true
  UsingFirehoseLogs:
    Type: String
    Default: false
  UsingKinesisLogs:
    Type: String
    Default: false
...
Rules:
  FirelensSingleCloudwatch:
    RuleCondition: !And
      - !Equals [ !Ref MultipleDests, 'false']
      - !Equals [ !Ref UsingCloudwatchLogs, 'true']
    Assertions:
      - AssertDescription: You cannot use any other firelens destination if a single
 cloudwatch stream is desired
        Assert: !And
```

```
          - !Equals [ !Ref UsingFirehoseLogs, 'false']
          - !Equals [ !Ref UsingKinesisLogs, 'false']
          - !Equals [ !Ref SingleDestName, "cloudwatch" ]
          - !Not [ !Equals [ !Ref LogGrpName, "" ]]
          - !Not [ !Equals [ !Ref LogStrmName, "" ]]
          - !Not [ !Equals [ !Ref AutoCrtGrp, "" ]]
  FirelensSingleFirehose:
    RuleCondition: !And
      - !Equals [ !Ref MultipleDests, 'false']
      - !Equals [ !Ref UsingFirehoseLogs, 'true']
    Assertions:
      - AssertDescription: You cannot use any other firelens destination if a single
 firehose stream is desired
        Assert: !And
          - !Equals [ !Ref UsingCloudwatchLogs, 'false']
          - !Equals [ !Ref UsingKinesisLogs, 'false']
          - !Equals [ !Ref SingleDestName, "firehose" ]
          - !Not [ !Equals [ !Ref FirehoseStream, "" ]]
  FirelensSingleKinesis:
    RuleCondition: !And
      - !Equals [ !Ref MultipleDests, 'false']
      - !Equals [ !Ref UsingKinesisLogs, 'true']
    Assertions:
      - AssertDescription: You cannot use any other firelens destination if a single
 kinesis stream is desired
        Assert: !And
          - !Equals [ !Ref UsingCloudwatchLogs, 'false']
          - !Equals [ !Ref UsingFirehoseLogs, 'false']
          - !Equals [ !Ref SingleDestName, "kinesis" ]
          - !Not [ !Equals [ !Ref KinesisStream, "" ]]
          - !Not [ !Equals [ !Ref KinesisAppendNewline, "" ]]
  MultipleDestinations:
    RuleCondition: !Equals [ !Ref MultipleDests, 'true']
    Assertions:
      - AssertDescription: You must supply a configuration file location and filepath if
 multiple firelens destinations are being used
        Assert: !And
          - !Not [ !Equals [ !Ref ConfigType, "" ] ]
          - !Not [ !Equals [ !Ref ConfigPath, "" ] ]
          - !Equals [ !Ref SingleDestName, ""]
          - !Equals [ !Ref IncludePattern, ""]
          - !Equals [ !Ref LogGrpName, ""]
          - !Equals [ !Ref LogStrmName, ""]
          - !Equals [ !Ref AutoCrtGrp, ""]
          - !Equals [ !Ref FirehoseStream, ""]
          - !Equals [ !Ref KinesisStream, ""]
          - !Equals [ !Ref KinesisAppendNewline, ""]
...
Conditions:
...
Resources:
 PrivateAppStack:
    Type: AWS::CloudFormation::Stack
    Condition: DoNotCreatePublicLoadBalancer
    Properties:
      TemplateURL: !Sub 'https://${S3Bucket}.s3.${S3Region}.${AWS::URLSuffix}/
${S3KeyPrefix}/ecs-private-app.yml'
      Tags:
        - Key: "a2c-generated"
          Value: !Sub 'ecs-app-${AWS::StackName}'
      Parameters:
...
        TaskLogDriver: !Ref TaskLogDriver
        MultipleDests: !Ref MultipleDests
        SingleDestName: !Ref SingleDestName
        IncludePattern: !Ref IncludePattern
        LogGrpName: !Ref LogGrpName
        LogStrmName: !Ref LogStrmName
        AutoCrtGrp: !Ref AutoCrtGrp
        FirehoseStream: !Ref FirehoseStream
        KinesisStream: !Ref KinesisStream
        KinesisAppendNewline: !Ref KinesisAppendNewline
        FirelensName: !Ref FirelensName
        FirelensImage: !Ref FirelensImage
        ConfigType: !Ref ConfigType
        ConfigPath: !Ref ConfigPath
        UsingCloudwatchLogs: !Ref UsingCloudwatchLogs
        UsingFirehoseLogs: !Ref UsingFirehoseLogs
        UsingKinesisLogs: !Ref UsingKinesisLogs
...
```

Step 4: Deploy your application to Amazon ECS

Deploy your application using the generate app-deployment (p. 82) command with the `--deploy` option.

```
$ sudo app2container generate app-deployment --deploy --application-id java-tomcat-9e8e4799
# AWS prerequisite check succeeded
# Docker prerequisite check succeeded
# Created ECR Repository
# Registered ECS Task Definition with ECS
# Uploaded CloudFormation resources to S3 Bucket: app2container-example
# Generated CloudFormation Master template at: /root/app2container/java-tomcat-9e8e4799/
EcsDeployment/ecs-master.yml
# Initiated CloudFormation stack creation. This may take a few minutes. Please visit the
 AWS CloudFormation Console to track progress.
ECS deployment successful for application java-tomcat-9e8e4799

The URL to your Load Balancer Endpoint is:
<your endpoint>.us-east-1.elb.amazonaws.com
Successfully created ECS stack app2container-java-tomcat-9e8e4799-ECS. Check the AWS
 CloudFormation Console for additional details.
```

Alternatively, you can deploy your application's AWS CloudFormation template using the AWS CLI as follows.

```
$ sudo aws cloudformation deploy --template-file /root/app2container/java-tomcat-9e8e4799/
EcsDeployment/ecs-master.yml --capabilities CAPABILITY_NAMED_IAM --stack-name
 app2container-java-tomcat-9e8e4799-ECS
```

Step 5: Verify log routing

After you deploy your application to Amazon ECS, you can verify that your logs are routing to their intended destinations.

Security in AWS App2Container

Security at AWS is the highest priority. As an AWS customer using AWS App2Container and tools such as Amazon ECR, Amazon ECS, and Amazon EKS, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon EC2, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

Contents

Data protection in App2Container

The AWS shared responsibility model applies to data protection in AWS App2Container. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form fields such as a **Name** field. This includes when you work with App2Container or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data encryption

App2Container communicates with AWS services using standard APIs when retrieving artifacts from Amazon S3 or pushing Docker containers to service endpoints in the AWS container management suite (Amazon ECR, Amazon ECS, and Amazon EKS). It works with AWS CloudFormation and AWS CodeStar services to generate and deploy relevant container and lifecycle artifacts using their standard APIs.

Encryption at rest

- App2Container installation packages are kept in a private Amazon S3 bucket with encryption enabled.
- Application artifacts can optionally be uploaded into Amazon S3 buckets. Enable encryption for your Amazon S3 bucket to enforce data encryption.

Encryption in transit

- App2Container installation packages are kept in a private Amazon S3 bucket, which requires secure download using the HTTPS protocol using links provided for each package.
- App2Container uses standard AWS APIs for the services it interacts with, including Amazon ECR, Amazon ECS, Amazon EKS, AWS CloudFormation, CodePipeline, and Amazon S3. AWS APIs use HTTPS as their default communication protocol.

Internetwork traffic privacy

App2Container does not store passwords, keys, or other secrets or customer-sensitive material. App2Container also ensures that no sensitive fields are contained in application logs.

Identity and access management in App2Container

Your AWS security credentials identify you to AWS and grant you access to your AWS resources. For example, they can allow you to access artifacts saved to an Amazon S3 bucket. You can use features of AWS Identity and Access Management (IAM) to allow other users, services, and applications to use specific resources in your AWS account without sharing your security credentials. You can choose to allow full use or limited use of your AWS resources.

If you are the owner of the AWS account and use AWS as the root user, we strongly recommend that you create an IAM admin user to use for access to your AWS resources. See Creating Your First IAM Admin User and Group in the *IAM User Guide* to set up your own access before setting up any other IAM users who need to use App2Container.

By default, IAM users don't have permission to create or modify resources. To allow IAM users to create or modify resources and perform tasks, you must create IAM policies that grant permission to use the specific resources and API actions that they need. For more information about IAM policies, see Policies and Permissions in the *IAM User Guide*.

IAM groups and roles are a flexible way to manage permissions across multiple users. When you assign a user to a group or when your user assumes a role, that user inherits the group's or role's permissions, and is allowed or denied permission to perform the specified tasks on the specified resources. You can assign multiple users to the same group, and a role can be assumed by authorized users. While groups and roles both serve the purpose of granting access to resources, roles are more task-oriented, and assuming a role provides you with temporary security credentials for your role session.

> **IAM security best practices**
>
> Follow these top four security best practices when setting up your IAM resources. For more information and additional best practices, see Security Best Practices in IAM in the *IAM User Guide*.
>
> 1. **Lock away your AWS account root user access keys**
>
>   Protect your root user access key like you would your credit card numbers or any other sensitive secret, and only use your root user account for necessary account and service management tasks.
> 2. **Create individual IAM users**
>
>   Don't use your AWS account root user credentials to access AWS, and don't give your credentials to anyone else. Instead, create individual users for anyone who needs access to your AWS account.
> 3. **Use groups or roles to assign permissions to IAM Users**
>
>   Instead of defining permissions for individual IAM users, it's usually more convenient to create groups that relate to job functions (administrators, developers, accounting, etc.) or roles that relate to specific tasks.
> 4. **Grant least privilege**
>
>   When you create IAM policies, follow the standard security advice of granting *least privilege*, or granting only the permissions required to perform a task. Determine what users (and roles) need to do and then craft policies that allow them to perform only those tasks.

We recommend that you create a general purpose IAM group that can run all of the commands *except* commands that are run with the `--deploy` option.

If you plan to use App2Container to deploy your containers or create pipelines, then you should create a separate IAM user for deployments. The deployment user needs to be able to create or update AWS objects for container management services (Amazon ECR, Amazon ECS, Amazon EKS, and App Runner), and to create pipelines with AWS CodeStar services. This requires elevated permissions that should only be used for deployment.

Set up IAM resources for App2Container

- Create IAM resources for general use (p. 64)
- Create IAM resources for deployment (p. 71)

Create IAM resources for general use

Follow best practices by using the following steps to create an IAM group with access to perform specific tasks, using specific resources, and to assign users to the group.

> **Note**
>
> Alternatively, you can create an IAM role and EC2 instance profile to grant permissions to applications that run on an Amazon EC2 instance. For more information about using instance profiles, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

1. **Create a customer managed IAM policy**

   You can create a customer managed IAM policy for your general purpose user or group, using one of the example policies (p. 65) on this page after you have customized the JSON to refer to your resources. To create a policy using the AWS console, see Creating policies on the JSON tab in the *IAM User Guide*. To create a policy using the AWS CLI, use the **create-policy** command.

> **Tip**
> Review your policy periodically, to add actions required for newer features, and to ensure that the policy continues to meet your needs.

2. **Create IAM users and a group**

   Every user who will run **app2container** commands needs to have an IAM user created for accessing AWS resources under your account. To follow best practices, you can create an IAM group with the policy attached, and assign users to it.

   To create an IAM user, see Creating an IAM User in Your AWS Account in the *IAM User Guide*. Be sure to select programmatic access to AWS when you create the IAM user.

   Perform the following steps to create an IAM group and assign users to it.

   a. To create an IAM group, see Creating IAM Groups in the *IAM User Guide*.
   b. Ensure that every person who will run **app2container** commands has an IAM user defined for AWS access.
   c. To assign the users to the group that you created in step 1a, see Adding Permissions to a User (Console), or Adding and Removing a User's Permissions (AWS CLI or AWS API) in the *IAM User Guide*.

3. **Save your AWS access keys**

   Save the access keys for your new or existing IAM user in a safe place. You'll need them to configure your AWS profile (p. 16) as part of getting set up for App2Container.

4. **Attach or assign the policy**

   Use one of the following methods to assign permissions to your IAM users.

   - **Attach the policy to the IAM group**

     Attach the policy that you created in step 1 to the group that you created in step 2. See Attaching a Policy to an IAM Group in the *IAM User Guide*.
   - **Embed the policy inline for an IAM user**

     Embed the policy that you created in step 1 inline for your IAM user. See the section that begins with "To embed an inline policy" in Adding Permissions to a User (Console), or Adding and Removing a User's Permissions (AWS CLI or AWS API) in the *IAM User Guide*.

Example IAM policies

You can use one of the policy templates in this section as a starting point to configure the access that App2Container uses on your behalf to generate the deployment artifacts for your application containers.

Choose the policy resources and actions that you need

The following sections in the example policies depend on choices you've made for your containerization environment and workflow:

- **SectionForS3Access** and **SectionForS3ReadAccess** – if you set up an Amazon S3 bucket for application or deployment artifacts, you must grant access to your bucket in the policy.

  You must also ensure that only authorized users can access the bucket. We recommend that you use server-side encryption for your bucket. See Protecting data using server-side encryption in the *Amazon Simple Storage Service User Guide* for more information about how to set it up.

- **SectionForByoVPC** – if you specify your own VPC or want to reuse an existing VPC that App2Container created for a prior deployment, you must grant access to associated describe actions in the policy.
- **SectionForMetricsService** – if you gave consent for App2Container to collect and export application usage metrics when you ran the **init** command, you must grant access to upload the metric data.
- **SectionForUploadSupportBundleService** – if you chose to have App2Container logs and command-generated artifacts uploaded automatically for failed commands when you ran the **init** command, you must grant access to upload the application support bundles.
- **SectionForSecretManagerAccess** – if you configured your environment to run remote workflows, App2Container requires you to use Secrets Manager for connection secrets to access application servers from the worker machine. You must grant access to retrieve secrets in the policy.
- **SectionForCodeCommitAccess** – if you use App2Container to generate a container pipeline, you must grant access to interact with your code repository.

Other policy sections in the examples are required for App2Container to generate application deployment artifacts.

IAM policy for Amazon ECS

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SectionForS3Access",
            "Action": [
                "s3:DeleteObject",
                "s3:GetBucketAcl",
                "s3:GetBucketLocation",
                "s3:GetObject",
                "s3:GetObjectAcl",
                "s3:ListAllMyBuckets",
                "s3:ListBucket",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Effect": "Allow",
            "Resource": "<user-provided-bucket-ARN>"
        },
        {
            "Sid": "SectionForS3ReadAccess",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:GetBucketAcl"
            ],
            "Resource": "arn:aws:s3:::*"
        },
        {
            "Sid": "SectionForECRAccess",
            "Action": [
                "ecr:BatchCheckLayerAvailability",
                "ecr:BatchDeleteImage",
                "ecr:BatchGetImage",
                "ecr:CompleteLayerUpload",
                "ecr:CreateRepository",
                "ecr:DeleteRepository",
                "ecr:DescribeImages",
                "ecr:DescribeRepositories",
                "ecr:GetAuthorizationToken",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:InitiateLayerUpload",
                "ecr:ListImages",
                "ecr:PutImage",
                "ecr:TagResource",
                "ecr:UntagResource",
                "ecr:UploadLayerPart"
            ],
            "Effect": "Allow",
            "Resource": "<resource-ARNs>"
        },
        {
            "Sid": "SectionForECSWriteAccess",
            "Action": [
                "ecs:CreateCluster",
                "ecs:CreateService",
                "ecs:CreateTaskSet",
                "ecs:DeleteCluster",
                "ecs:DeleteService",
                "ecs:DeleteTaskSet",
                "ecs:DeregisterTaskDefinition",
                "ecs:Poll",
                "ecs:RegisterContainerInstance",
                "ecs:RegisterTaskDefinition",
                "ecs:RunTask",
                "ecs:StartTask",
                "ecs:StopTask",
                "ecs:SubmitContainerStateChange",
                "ecs:SubmitTaskStateChange",
                "ecs:UpdateContainerInstancesState",
                "ecs:UpdateService",
                "ecs:UpdateServicePrimaryTaskSet",
                "ecs:UpdateTaskSet"
            ],
            "Effect": "Allow",
            "Resource": "<resource-ARNs>"
        },
        {
            "Sid": "SectionForPassRoleToECS",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "<ARN for ecsTaskExecutionRole>"
        },
        {
            "Sid": "SectionForECSReadAccess",
            "Action": [
                "ecs:DescribeClusters",
                "ecs:DescribeContainerInstances",
                "ecs:DescribeServices",
                "ecs:DescribeTaskDefinition",
                "ecs:DescribeTaskSets",
                "ecs:DescribeTasks",
                "ecs:ListClusters",
                "ecs:ListContainerInstances",
                "ecs:ListServices",
                "ecs:ListTaskDefinitionFamilies",
                "ecs:ListTaskDefinitions",
                "ecs:ListTasks"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "SectionForCodeCommitAccess",
            "Effect": "Allow",
            "Action": [
                "codecommit:GetRepository",
                "codecommit:GetBranch",
                "codecommit:CreateRepository",
                "codecommit:CreateCommit",
                "codecommit:TagResource"
            ],
            "Resource": "arn:aws:codecommit:*:*:*"
        },
        {
            "Sid": "SectionForByoVPC",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInternetGateways",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs"
            ],
            "Resource": "*"
        },
        {
          "Sid": "SectionForEC2",
          "Effect": "Allow",
          "Action": [
             "ec2:DescribeKeyPairs",
             "ec2:CreateKeyPair",
             "ec2:DescribeAvailabilityZones"
          ],
          "Resource": "*"
        },
        {
            "Sid": "SectionForMetricsService",
            "Effect": "Allow",
            "Action": "execute-api:invoke",
            "Resource": "arn:aws:execute-api:us-east-1:*:*/prod/POST/put-metric-data"
        },
        {
          "Sid": "SectionForUploadSupportBundleService",
          "Effect": "Allow",
          "Action": "execute-api:invoke",
          "Resource": "arn:aws:execute-api:us-east-1:*:*/prod/POST/put-log-data"
        },
        {
            "Sid": "SectionForSecretManagerAccess",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:secretsmanager:<user's region>:<user's account
 ID>:secret:a2c/*"
        }
    ]
}
```

IAM policy for Amazon EKS

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SectionForS3Access",
            "Action": [
                "s3:DeleteObject",
                "s3:GetBucketAcl",
                "s3:GetBucketLocation",
                "s3:GetObject",
                "s3:GetObjectAcl",
                "s3:ListAllMyBuckets",
                "s3:ListBucket",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Effect": "Allow",
            "Resource": "<user-provided-bucket-ARN>"
        },
        {
            "Sid": "SectionForS3ReadAccess",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:GetBucketAcl"
            ],
            "Resource": "arn:aws:s3:::*"
        },
        {
            "Sid": "SectionForECRAccess",
            "Action": [
                "ecr:BatchCheckLayerAvailability",
                "ecr:BatchDeleteImage",
                "ecr:BatchGetImage",
                "ecr:CompleteLayerUpload",
                "ecr:CreateRepository",
                "ecr:DeleteRepository",
                "ecr:DescribeImages",
                "ecr:DescribeRepositories",
                "ecr:GetAuthorizationToken",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:InitiateLayerUpload",
                "ecr:ListImages",
                "ecr:PutImage",
                "ecr:TagResource",
                "ecr:UntagResource",
                "ecr:UploadLayerPart"
            ],
            "Effect": "Allow",
            "Resource": "<resource-ARNs>"
        },
        {
            "Sid": "SectionForEKS",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "lambda:GetFunction"
            ],
            "Resource": [
                "arn:aws:iam::*:role/eks-quickstart-ResourceReader",
                "arn:aws:lambda:<target Region>:*:function:eks-quickstart-ResourceReader"
            ]
        },
        {
            "Sid": "SectionForCodeCommitAccess",
            "Effect": "Allow",
            "Action": [
                "codecommit:GetRepository",
                "codecommit:GetBranch",
                "codecommit:CreateRepository",
                "codecommit:CreateCommit",
                "codecommit:TagResource"
            ],
            "Resource": "arn:aws:codecommit:*:*:*"
        },
        {
            "Sid": "SectionForByoVPC",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInternetGateways",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs"
            ],
            "Resource": "*"
        },
        {
          "Sid": "SectionForEC2",
          "Effect": "Allow",
          "Action": [
             "ec2:DescribeKeyPairs",
             "ec2:CreateKeyPair",
             "ec2:DescribeAvailabilityZones"
          ],
          "Resource": "*"
        },
        {
            "Sid": "SectionForMetricsService",
            "Effect": "Allow",
            "Action": "execute-api:invoke",
            "Resource": "arn:aws:execute-api:us-east-1:*:*/prod/POST/put-metric-data"
        },
        {
          "Sid": "SectionForUploadSupportBundleService",
          "Effect": "Allow",
          "Action": "execute-api:invoke",
          "Resource": "arn:aws:execute-api:us-east-1:*:*/prod/POST/put-log-data"
        },
        {
            "Sid": "SectionForSecretManagerAccess",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:secretsmanager:<user's region>:<user's account
 ID>:secret:a2c/*"
        }
    ]
}
```

IAM policy for AWS App Runner

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SectionForAppRunnerAccess",
            "Action": [
                "apprunner:List*",
                "apprunner:Describe*"
            ],
            "Effect": "Allow",
            "Resource": "<resource-ARNs>"
        },
        {
            "Sid": "SectionForECRAccess",
            "Action": [
                "ecr:BatchCheckLayerAvailability",
                "ecr:BatchDeleteImage",
                "ecr:BatchGetImage",
                "ecr:CompleteLayerUpload",
                "ecr:CreateRepository",
                "ecr:DeleteRepository",
                "ecr:DescribeImages",
                "ecr:DescribeRepositories",
                "ecr:GetAuthorizationToken",
                "ecr:GetDownloadUrlForLayer",
                "ecr:GetRepositoryPolicy",
                "ecr:InitiateLayerUpload",
                "ecr:ListImages",
                "ecr:PutImage",
                "ecr:TagResource",
                "ecr:UntagResource",
                "ecr:UploadLayerPart"
            ],
            "Effect": "Allow",
            "Resource": "<resource-ARNs>"
        },
        {
            "Sid": "SectionForCodeCommitAccess",
            "Effect": "Allow",
            "Action": [
                "codecommit:GetRepository",
                "codecommit:GetBranch",
                "codecommit:CreateRepository",
                "codecommit:CreateCommit",
                "codecommit:TagResource"
            ],
            "Resource": "arn:aws:codecommit:*:*:*"
        },
        {
            "Sid": "SectionForMetricsService",
            "Effect": "Allow",
            "Action": "execute-api:invoke",
            "Resource": "arn:aws:execute-api:us-east-1:*:*/prod/POST/put-metric-data"
        },
        {
            "Sid": "SectionForUploadSupportBundleService",
            "Effect": "Allow",
            "Action": "execute-api:invoke",
            "Resource": "arn:aws:execute-api:us-east-1:*:*/prod/POST/put-log-data"
        },
        {
            "Sid": "SectionForSecretManagerAccess",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:secretsmanager:us-east-1:*:secret:a2c/*"
        }
    ]
}
```

Create IAM resources for deployment

The **AdministratorAccess** policy grants an IAM user full access to AWS. Therefore, IAM users with this policy can deploy a containerized application using any of the AWS services for deployment that are supported by App2Container.

1. **Create an IAM user**

   You can create an IAM user with full access to AWS API actions and resources. Be sure to grant the user programmatic access to AWS and to attach the **AdministratorAccess** policy. For more information, see Creating IAM users in the *IAM User Guide*.
2. **Save your AWS access keys**

   Save the access keys for the IAM user in a safe place. You'll need them to configure your AWS profile (p. 16) as part of getting set up for App2Container.

Update management for App2Container

App2Container detects what version of the CLI you are using when you run a command. It notifies you if there are published updates available. You can install the latest version of App2Container using the upgrade (p. 103) command.

App2Container command reference

AWS App2Container is a command line tool that transforms supported legacy applications running on physical servers or virtual machines into applications that run in Docker containers on Amazon ECS, Amazon EKS, or AWS App Runner.

Note
Running App2Container commands on a Linux server requires elevated permissions. Prefix the command syntax with **sudo**, or run the **sudo su** command one time when you log in before running the commands as shown in the syntax for the commands linked below.

Containerization phases

The containerization process has several phases.

Phases

- Initialize (p. 73)
- Analyze (p. 73)
- Transform (p. 74)
- Deploy (p. 75)

Initialize

The **init** command performs one-time initialization tasks for App2Container. This interactive command prompts for the information required to set up the local App2Container environment. Run this command before you run any other App2Container commands. If you are using a worker machine to run commands remotely on application servers, you must also run the **remote configure** command on the worker machine.

init (p. 92)

Run the **init** command to configure the AWS App2Container workspace on your application servers and worker machines. If you are using a worker machine, and running commands remotely, the **init** command is only required on the worker machine.

remote configure (p. 98)

After setting up remote access for the worker machine on your application server (see Enable remote access for a worker machine (optional) (p. 15)), run the **remote configure** command on the worker machine to configure the connections needed to run remote workflows on application servers. This interactive command prompts for the required information for each application server that you enter.

Analyze

After you have completed setup and initialization tasks on your servers, you can begin the analyze phase. Run the version of these commands that applies to your server setup:

Run commands directly on application servers

inventory (p. 94)

Run the **inventory** command to produce an inventory of applications that are running on your application servers, and to assign each one a unique ID to use when you run other commands.

analyze (p. 76)

Run the **analyze** command to analyze your running applications and to identify dependencies that are required for containerization. This command creates the `analysis.json` file that feeds into the Transform phase commands.

Run commands remotely from a worker machine

remote inventory (p. 102)

Run the **remote inventory** command from your worker machine to produce an inventory of applications that are running on your target application server and to assign each one a unique ID to use when you run other commands.

remote analyze (p. 96)

Run the **remote analyze** command from your worker machine to analyze the applications running on your target application server, and to identify dependencies that are required for containerization. This command creates the `analysis.json` file that feeds into the Transform phase commands.

Transform

The transform phase creates containers for your applications that have gone through analysis. Run the version of these commands that applies to your server setup:

Run the extract directly on application servers, or run the remote extract from a worker machine

extract (p. 80)

Run the **extract** command on your application server to generate an application archive based on the `analysis.json` file, created by the **analyze** command. Transfer the archive to the worker machine for the remaining steps that require the operating system to support containers.

remote extract (p. 100)

Run the **remote extract** command from your worker machine to generate an application archive for the applications running on your target application server, based on the `analysis.json` file that was created by the **analyze** command.

Run all remaining commands directly on application servers or on a worker machine

containerize (p. 77)

Run the **containerize** command for the application specified in the `--application id` parameter to do the following:

- Extract application artifacts or read from an extract archive for the specified application. For complex, multi-component Windows applications, this also applies to any additional applications or services that run in the same container.

- Generate Docker container artifacts, including a Dockerfile and container image, based on the application artifacts, and the application settings in the `analysis.json` file.
- Create the `deployment.json` file for input to the **generate app-deployment** command

Deploy

The deploy phase consists of deploying an application to your target container management environment (Amazon ECR with Amazon ECS, Amazon EKS, or AWS App Runner), and optionally creating a CI/CD pipeline to automate future deployments.

generate app-deployment (p. 82)

Option 1: Generate deployment artifacts and deploy directly

Run the **generate app-deployment** command with the `--deploy` option to generate container deployment artifacts and to deploy them to your target environment all in one step.

Option 2: Generate deployment artifacts and customize

- Run the **generate app-deployment** command without the deployment option to generate deployment artifacts.
- Review and customize the generated Amazon ECS, Amazon EKS, or AWS App Runner deployment artifacts.
- Deploy to your target environment using the AWS CLI or AWS console.

generate pipeline (p. 87) (optional)

Option 1: Generate CI/CD pipeline artifacts and deploy directly

Run the **generate pipeline** command with the `--deploy` option to generate CI/CD pipeline artifacts and to deploy them with AWS CodePipeline all in one step.

Option 2: Generate CI/CD pipeline artifacts and customize

- Run the **generate pipeline** command without the deployment option to generate pipeline artifacts.
- Review and customize the generated pipeline artifacts.
- Deploy to your target environment using the AWS CLI or AWS console.

Utility commands

The following additional commands help you maintain AWS App2Container in your environment.

upgrade (p. 103)

Run the **upgrade** command to upgrade your existing installation of App2Container. This command checks if there is a newer version of App2Container available, and automatically upgrades if doing so will not break backwards compatibility with previously generated container artifacts.

upload-support-bundle (p. 104)

Run the **upload-support-bundle** command for assistance from the AWS App2Container team for troubleshooting a command failure. This command securely uploads App2Container logs and supporting artifacts, and an optional message for your troubleshooting request to the AWS App2Container team.

app2container analyze command

Analyzes the specified application and generates a report.

Note
If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.
When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.
Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container analyze --application-id id [--help]
```

Parameters and options

Parameters

--application-id *id*

The application ID *(required)*. After you run the inventory (p. 94) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

Options

--help

Displays the command help.

Output

The **analyze** command creates files and directories for each application. Output varies slightly, depending on your application language and the application server operating system.

The application directory is created in the output location that you specified when you ran the **init** command. Each application has its own directory named for the application ID. The directory contains analysis output and editable application configuration files. The files are stored in subdirectories that match the application structure on the server.

An `analysis.json` file is created for the application that is specified in the `--application-id` parameter. The file contains information about the application found during analysis, and configurable fields for container settings. See Java application analysis file (p. 29), or .NET application analysis file (p. 32) for more information about configurable fields, and for an example of what the file looks like.

For .NET applications, App2Container detects connection strings and produces the `report.txt` file. The report location is specified in the `analysis.json` file, in the `reportPath` attribute of the `analysisInfo` section. You can use this report to identify the changes that you need to make in application configuration files to connect your application container to new database endpoints, if needed. The report also contains the locations of other configuration files that might need changes.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **analyze** command with the `--application-id` parameter and no additional options.

```
$ sudo app2container analyze --application-id java-tomcat-9e8e4799
# Created artifacts folder /root/app2container/java-tomcat-9e8e4799
# Generated analysis data in /root/app2container/java-tomcat-9e8e4799/analysis.json
Analysis successful for application java-tomcat-9e8e4799
Please examine the application analysis file at /root/app2container/java-
tomcat-9e8e4799/analysis.json,
make appropriate edits and initiate containerization using "app2container containerize
 --application-id java-tomcat-9e8e4799
```

Windows

The following example shows the **analyze** command with the `--application-id` parameter and no additional options.

```
PS> app2container analyze --application-id iis-smarts-51d2dbf8
# Created artifacts folder C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8
# Generated analysis data in C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8\analysis.json
Analysis successful for application iis-smarts-51d2dbf8
Please examine the application analysis file at C:\Users\Administrator\AppData\Local
\app2container\iis-smarts-51d2dbf8\analysis.json,
make appropriate edits and initiate containerization using "app2container containerize
 --application-id iis-smarts-51d2dbf8
```

app2container containerize command

When you run this command, it creates a Docker container for your application. The container is based on the parameters in the `analysis.json` file that is generated by the analyze (p. 76) command, along with any customizations you have made.

See Configuring application containers (p. 29) for more information about configuring the `analysis.json` file.

Note
If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.
When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.

Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container containerize {--application-id id | --input-archive extraction-file} [--help]
```

Parameters and options

Parameters

--application-id *id*

The application ID *(required)*. After you run the inventory (p. 94) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

--input-archive *extraction-file*

The file path or Amazon S3 key (for example, s3://*bucket*/*archive-key*) for the application archive. If you specify an application archive, the command downloads and opens the archive, and then builds the container image.

--profile *admin-profile*

Use this option to specify a *named profile* to run this command. For more information about named profiles in AWS, see Named profiles in the AWS Command Line Interface User Guide

Options

--build-only

Builds container images based on the existing `Dockerfile` and artifacts.

--force

Bypasses the disk space prerequisite check.

--help

Displays the command help.

Output

This command generates a `Dockerfile`, a container image, and a `deployment.json` file that you can use with the generate app-deployment (p. 82) command.

It also generates a `Dockerfile.update` file that you can use to make updates to your containerized application. The generate pipeline (p. 87) command adds this Dockerfile to your CodeCommit repository and deploys updates to your CodePipeline infrastructure.

See deployment.json file (p. 39) for more information about configuration, and for an example of what the `deployment.json` file looks like.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **containerize** command with the `--application-id` parameter and no additional options.

```
$ sudo app2container containerize --application-id java-tomcat-9e8e4799
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Extracted container artifacts for application
# Entry file generated
# Dockerfile generated under /root/app2container/java-tomcat-9e8e4799/Artifacts
# Generated dockerfile.update under /root/app2container/java-tomcat-9e8e4799/Artifacts
# Generated deployment file at /root/app2container/java-tomcat-9e8e4799/deployment.json
Containerization successful. Generated docker image java-tomcat-9e8e4799

You're all set to test and deploy your container image.

Next Steps:
1. View the container image with \"docker images\" and test the application.
2. When you're ready to deploy to AWS, please edit the deployment file as needed at /
root/app2container/java-tomcat-9e8e4799/deployment.json.
3. Generate deployment artifacts using app2container generate app-deployment --
application-id java-tomcat-9e8e4799
Please use "docker images" to view the generated container image.
```

The following example shows the **containerize** command with the `--input-archive` option.

```
$ sudo app2container containerize --input-archive /var/aws/java-tomcat-9e8e4799/java-
tomcat-9e8e4799-extraction.tar
```

Windows

The following example shows the **containerize** command with the `--application-id` parameter and no additional options.

```
PS> app2container containerize --application-id iis-smarts-51d2dbf8
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Extracted container artifacts for application
# Entry file generated
# Dockerfile generated under C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8\Artifacts
# Generated dockerfile.update under C:\Users\Administrator\AppData\Local\app2container
\iis-smarts-51d2dbf8\Artifacts
# Generated deployment file at C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8\deployment.json
Containerization successful. Generated docker image iis-smarts-51d2dbf8

You're all set to test and deploy your container image.

Next Steps:
1. View the container image with \"docker images\" and test the application.
2. When you're ready to deploy to AWS, please edit the deployment file as needed at C:
\Users\Administrator\AppData\Local\app2container\iis-smarts-51d2dbf8\deployment.json.
3. Generate deployment artifacts using app2container generate app-deployment --
application-id iis-smarts-51d2dbf8
```

```
Please use "docker images" to view the generated container image.
```

The following example shows the **containerize** command with the `--input-archive` option.

```
PS> app2container containerize --input-archive archive C:\Users\Administrator\Downloads
\iis-smarts-51d2dbf8.zip
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Dockerfile generated under C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8\Artifacts
# Generated dockerfile.update under C:\Users\Administrator\AppData\Local\app2container
\iis-smarts-51d2dbf8\Artifacts
# Generated deployment file at C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8\deployment.json
Containerization successful. Generated docker image iis-smarts-51d2dbf8

You're all set to test and deploy your container image.

Next Steps:
1. View the container image with \"docker images\" and test the application.
2. When you're ready to deploy to AWS, please edit the deployment file as needed at C:
\Users\Administrator\AppData\Local\app2container\iis-smarts-51d2dbf8\deployment.json.
3. Generate deployment artifacts using app2container generate app-deployment --
application-id iis-smarts-51d2dbf8
To have gMSA related artifacts generated with CloudFormation, please edit gMSAParams
 inside deployment file.
Otherwise look at C:\Users\Administrator\AppData\Local\app2container\iis-
smarts-51d2dbf8\Artifacts\WindowsAuthSetupInstructions.md for setup instructions on
 Windows Authentication
Please use "docker images" to view the generated container image.
```

app2container extract command

Generates an application archive for the specified application. Before you call this command, you must call the analyze (p. 76) command.

Note

If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.

When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.

Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container extract --application-id id [--output s3] [--help]
```

Parameters and options

Parameters

--application-id *id*

The application ID *(required)*. After you run the inventory (p. 94) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

--profile *`admin-profile`*

Use this option to specify a *named profile* to run this command. For more information about named profiles in AWS, see Named profiles in the AWS Command Line Interface User Guide

Options

--output s3

If specified, this option writes the archive file to the Amazon S3 bucket that you specified when you ran the **init** command.

--force

Bypasses the disk space prerequisite check.

--help

Displays the command help.

Output

This command creates an archive file. When you use the `--output s3` option, the archive is written to the Amazon S3 bucket that you specified when you ran the **init** command. Otherwise, the archive is written to the output location that you specified when you ran the **init** command.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **extract** command with the `--application-id` parameter and no additional options.

```
$ sudo app2container extract --application-id java-tomcat-9e8e4799
# Extracted container artifacts for application
# Application archive file created at: /root/app2container/java-tomcat-9e8e4799/java-
tomcat-9e8e4799-extraction.tar
Extraction successful for application java-tomcat-9e8e4799

Please transfer this tar file to your worker machine and run, "app2container
 containerize --input-archive <extraction-tar-filepath>"
```

Windows

The following example shows the **extract** command with the `--application-id` parameter and no additional options.

```
PS> app2container extract --application-id iis-smarts-51d2dbf8
# Extracted container artifacts for application
```

```
Extraction successful for application iis-smarts-51d2dbf8
```

app2container generate app-deployment command

When you run this command, it generates the artifacts needed to deploy your application container in AWS. App2Container pre-fills key values in the artifacts based on your profile, the application analysis, your App2Container workflow, and best practices.

> **Note**
> For Windows applications, App2Container chooses the base image for your application container and Amazon ECS cluster, based on the worker machine or application server OS where you run the containerization command. Windows application containers running on Amazon EKS use Windows Server Core 2019 for the base image.

You have two options for deployment to your target container management environment (Amazon ECR with Amazon ECS, Amazon EKS, or App Runner):

- You can use the `--deploy` option to deploy directly.
- You can review and customize deployment artifacts, and then deploy using the AWS CLI or AWS console.

This command accesses AWS resources to generate and deploy artifacts to your target environment. The IAM user with administrator access that you created during security setup is required to run the command with the `--deploy` option. See Identity and access management in App2Container (p. 63) for more information about setting up IAM users for App2Container.

The command uses the `deployment.json` file that is generated by the containerize (p. 77) command. You can edit the `deployment.json` file to specify parameters for your deployment, such as:

- An image repository name for Amazon ECR
- Task definition parameters for Amazon ECS
- The Kubernetes app name
- The App Runner service name

See Configuring container deployment (p. 39) for more information about configuring the `deployment.json` file.

> **Note**
> If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.
> When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.
> Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container generate app-deployment --application-id id [--deploy] [--profile admin-
profile] [--help]
```

Parameters and options

Parameters

--application-id *id*

The application ID *(required)*. After you run the inventory (p. 94) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

--profile *admin-profile*

Use this option to specify a *named profile* to run this command. For more information about named profiles in AWS, see Named profiles in the AWS Command Line Interface User Guide

Options

--deploy

Use this option to deploy directly to your target container management environment (Amazon ECR with Amazon ECS, Amazon EKS, or App Runner).

> **Note**
> When you use the `--deploy` option to deploy directly to target environments, we recommend that you use the `--profile` option to specify a *named profile* that has elevated permissions.

--help

Displays the command help.

Output

You have two options for deploying a container to your target environment using the **generate app-deployment** command.

- You can use the `--deploy` option to deploy directly.
- You can review and customize deployment artifacts, and then deploy using the AWS CLI or AWS console.

Generate container deployment artifacts for customization

```
app2container generate app-deployment --application-id id
```

To see the steps App2Container performs and the artifacts that it creates to generate an application deployment for your target container management service, choose the tab that matches your environment:

Amazon ECS

App2Container performs the following tasks and creates artifacts for deployment to Amazon ECS:

> **Note**
> The `createEcsArtifacts` parameter in the `deployment.json` file must be set to `true` in order to generate Amazon ECS artifacts. See Configuring container deployment (p. 39) for more information about configuring the `deployment.json` file.

- Checks for AWS and Docker prerequisites.
- Creates an Amazon ECR repository.
- Pushes the container image to the Amazon ECR repository.
- Generates an Amazon ECS task definition template.
- Generates a `pipeline.json` file.

Amazon EKS

App2Container performs the following tasks and creates artifacts for deployment to Amazon EKS:

> **Note**
> The `createEksArtifacts` parameter in the `deployment.json` file must be set to `true` in order to generate Amazon EKS artifacts. See Configuring container deployment (p. 39) for more information about configuring the `deployment.json` file.

- Checks for AWS and Docker prerequisites.
- Creates an Amazon ECR repository.
- Pushes the container image to the Amazon ECR repository.
- Generates a Kubernetes `deployment.yaml` AWS CloudFormation template that you can use with Amazon EKS or the **kubectl** command.
- Generates a `pipeline.json` file.

AWS App Runner

App2Container performs the following tasks and creates artifacts for deployment to AWS App Runner:

> **Note**
> The `createAppRunnerArtifacts` parameter in the `deployment.json` file must be set to `true` in order to generate App Runner artifacts. See Configuring container deployment (p. 39) for more information about configuring the `deployment.json` file.
> *App Runner deployment is currently available for Linux applications only.*

- Checks for AWS and Docker prerequisites.
- Creates an Amazon ECR repository.
- Pushes the container image to the Amazon ECR repository.
- Generates the `apprunner.yaml` AWS CloudFormation template.
- Generates a `pipeline.json` file.

Deploy directly to target environments

```
app2container generate app-deployment --application-id id --deploy --profile admin-profile
```

When you use the `--deploy` option to deploy directly to your target container management service, App2Container performs all of the same steps and generates all of the artifacts that it does for customization. Additionally, it performs the following steps to complete the deployment:

- Uploads AWS CloudFormation resources to an Amazon S3 bucket, if configured
- Creates an AWS CloudFormation stack

See pipeline.json file (p. 48) for more information about pipeline configuration, and for an example of the `deployment.json` file.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **generate app-deployment** command with the `--application-id` parameter.

```
$ sudo app2container generate app-deployment --application-id java-tomcat-9e8e4799
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Created ECR Repository
# Registered ECS Task Definition with ECS
# Uploaded CloudFormation resources to S3 Bucket: app2container/-example
# Generated CloudFormation Master template at: /root/app2container/java-
tomcat-9e8e4799/EcsDeployment/ecs-master.yml
ECS CloudFormation templates and additional deployment artifacts generated successfully
 for application java-tomcat-9e8e4799

You're all set to use AWS CloudFormation to manage your application stack.

Next Steps:
1. Edit the CloudFormation template as necessary.
2. Create an application stack using the AWS CLI or the AWS Console. AWS CLI command:
aws cloudformation deploy --template-file /root/app2container/java-tomcat-9e8e4799/
EcsDeployment/ecs-master.yml --capabilities CAPABILITY_NAMED_IAM --stack-name
 app2container-java-tomcat-9e8e4799-ECS
3. Set up a pipeline for your application stack using app2container:
app2container generate pipeline --application-id java-tomcat-9e8e4799
```

The following example shows the **generate app-deployment** command with the `--application-id` parameter for an application that is deployed to AWS App Runner.

```
$ sudo app2container generate app-deployment --application-id java-tomcat-9e8e4799
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Created ECR repository 123456789012.dkr.ecr.us-west-2.amazonaws.com/java-
tomcat-9e8e4799 already
# Pushed docker image to 123456789012.dkr.ecr.us-west-2.amazonaws.com/java-
tomcat-9e8e4799:latest to ECR repository
# Generated AWS App Runner CloudFormation template at /root/app2container/java-
tomcat-9e8e4799/AppRunnerDeployment/apprunner.yml

CloudFormation templates and additional deployment artifacts generated successfully for
 application java-tomcat-9e8e4799

You're all set to use AWS CloudFormation to manage your application stack.

Next Steps:
1. Edit the CloudFormation template as necessary.
2. Create an application stack using the AWS CLI or the AWS Console. AWS CLI command:

aws cloudformation deploy --template-file /root/app2container/java-tomcat-9e8e4799/
AppRunnerDeployment/apprunner.yml --capabilities CAPABILITY_IAM --stack-name a2c-java-
tomcat-9e8e4799-AppRunner

3. Set up a pipeline for your application stack using app2container:
app2container generate pipeline --application-id java-tomcat-9e8e4799
```

The following example shows the **generate app-deployment** command with the `--application-id` parameter and the `--deploy` option.

```
$ sudo app2container generate app-deployment --deploy --application-id java-
tomcat-9e8e4799 --profile admin-profile
# AWS prerequisite check succeeded
# Docker prerequisite check succeeded
# Created ECR Repository
# Registered ECS Task Definition with ECS
# Uploaded CloudFormation resources to S3 Bucket: app2container-example
# Generated CloudFormation Master template at: /root/app2container/java-
tomcat-9e8e4799/EcsDeployment/ecs-master.yml
# Initiated CloudFormation stack creation. This may take a few minutes. Please visit
 the AWS CloudFormation Console to track progress.
ECS deployment successful for application java-tomcat-9e8e4799

The URL to your Load Balancer Endpoint is:
<your endpoint>.us-east-1.elb.amazonaws.com
Successfully created ECS stack app2container-java-tomcat-9e8e4799-ECS. Check the AWS
 CloudFormation Console for additional details.
```

The following example shows the **generate app-deployment** command with the `--application-id` parameter and the `--deploy` option for an application that is deployed to AWS App Runner.

```
$ sudo app2container generate app-deployment --application-id java-tomcat-9e8e4799 --
deploy
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Created ECR repository 123456789012.dkr.ecr.us-west-2.amazonaws.com/java-
tomcat-9e8e4799
# Pushed docker image to 123456789012.dkr.ecr.us-west-2.amazonaws.com/java-
tomcat-9e8e4799:latest to ECR repository
# Generated AWS App Runner CloudFormation template at /root/app2container/java-
tomcat-9e8e4799/AppRunnerDeployment/apprunner.yml

Deployment successful for application java-tomcat-9e8e4799
Access your newly deployed App Runner service at the following URL:
https://xyz123abc4.us-west-2.awsapprunner.com
Stack deployed successfully!

Set up a pipeline for your application stack using app2container:
app2container generate pipeline --application-id java-tomcat-9e8e4799
```

Windows

The following example shows the **generate app-deployment** command with the `--application-id` parameter.

```
PS> app2container generate app-deployment --application-id iis-smarts-51d2dbf8
# AWS pre-requisite check succeeded
# Docker pre-requisite check succeeded
# Created ECR Repository
# Registered ECS Task Definition with ECS
# Uploaded CloudFormation resources to S3 Bucket: app2container\-testing
# Generated CloudFormation Master template at: C:\Users\Administrator\AppData\Local
\app2container\iis-smarts-51d2dbf8\EcsDeployment\ecs-master.yml
ECS CloudFormation templates and additional deployment artifacts generated successfully
 for application iis-smarts-51d2dbf8

You're all set to use AWS CloudFormation to manage your application stack.

Next Steps:
1. Edit the CloudFormation template as necessary.
```

```
2. Create an application stack using the AWS CLI or the AWS Console. AWS CLI command:
aws cloudformation deploy --template-file C:\Users\Administrator\AppData\Local
\app2container\iis-smarts-51d2dbf8\EcsDeployment\ecs-master.yml --capabilities
 CAPABILITY_NAMED_IAM --stack-name app2container-iis-smarts-51d2dbf8-ECS
3. Set up a pipeline for your application stack using app2container:
app2container generate pipeline --application-id iis-smarts-51d2dbf8
```

The following example shows the **generate app-deployment** command with the `--application-id` parameter and the `--deploy` option.

```
PS> app2container generate app-deployment --deploy --application-id iis-smarts-51d2dbf8
 --profile admin-profile
# AWS prerequisite check succeeded
# Docker prerequisite check succeeded
# Created ECR Repository
# Registered ECS Task Definition with ECS
# Uploaded CloudFormation resources to S3 Bucket: app2container-example
# Generated CloudFormation Master template at: C:\Users\Administrator\AppData\Local
\app2container\iis-smarts-51d2dbf8\EcsDeployment\ecs-master.yml
# Initiated CloudFormation stack creation. This may take a few minutes. Please visit
 the AWS CloudFormation Console to track progress.
ECS deployment successful for application iis-smarts-51d2dbf8

The URL to your Load Balancer Endpoint is:
<your endpoint>.us-east-1.elb.amazonaws.com
Successfully created ECS stack app2container-iis-smarts-51d2dbf8-ECS. Check the AWS
 CloudFormation Console for additional details.
```

app2container generate pipeline command

When you run this command, it generates the artifacts needed to create a CI/CD pipeline with CodePipeline, based on your application deployment settings and artifacts.

Note
For Windows applications, App2Container chooses the base image for your application container and Amazon ECS cluster, based on the worker machine or application server OS where you run the containerization command. Windows application containers running on Amazon EKS use Windows Server Core 2019 for the base image.

You have two options for creating your pipeline:

- You can use the `--deploy` option to create your pipeline directly.
- You can review and customize pipeline artifacts, and then create your pipeline using the AWS CLI or the AWS console.

This command accesses AWS resources when it generates artifacts and creates CI/CD pipelines. The IAM user with administrator access that you created during security setup is required to run the command with the `--deploy` option. See Identity and access management in App2Container (p. 63) for more information about setting up IAM users for App2Container.

The command uses the `pipeline.json` file that is generated by the generate app-deployment (p. 82) command. You can edit the `pipeline.json` file to specify your container repository and target environments for Amazon ECS or Amazon EKS. See Configuring container pipelines (p. 48) for more information about configuring the `pipeline.json` file.

Note
If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.

When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.
Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container generate pipeline --application-id id [--deploy] [--profile admin-profile]
 [--help]
```

Parameters and options

Parameters

--application-id *id*

The application ID *(required)*. After you run the inventory (p. 94) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

--profile *admin-profile*

Use this option to specify a *named profile* to run this command. For more information about named profiles in AWS, see Named profiles in the AWS Command Line Interface User Guide

Options

--deploy

Use this option to create your CI/CD pipeline directly.

Note
When you use the `--deploy` option to create your CI/CD pipeline directly, we recommend that you use the `--profile` option to specify a *named profile* that has elevated permissions.

--help

Displays the command help.

Output

You have two options for creating your CI/CD pipeline using the **generate pipeline** command.

- You can use the `--deploy` option to create your pipeline directly.
- You can review and customize pipeline artifacts, and then create your pipeline using the AWS CLI or the AWS console.

The lists below show what artifacts are generated and what tasks are performed when you run the **generate pipeline** command.

Generate pipeline artifacts for customization

- Generate CI/CD artifacts: **generate pipeline `--application-id` *`id`***
  - Checks for AWS and Docker prerequisites
  - Creates a CodeCommit repository, if one does not exist already
  - Generates a buildspec file
  - Generates AWS CloudFormation templates

Create pipeline directly

- Generate CI/CD artifacts and create pipeline: **generate pipeline `--application-id` *`id`* `--deploy` `--profile` *`admin-profile`***
  - Checks for AWS and Docker prerequisites
  - Creates a CodeCommit repository, if one does not exist already
  - Commits files to a CodeCommit repository
  - Creates an AWS CloudFormation stack

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **generate pipeline** command with the `--application-id` parameter.

```
$ sudo app2container generate pipeline --application-id java-tomcat-9e8e4799
# Created CodeCommit repository
# Generated buildspec file(s)
# Generated CloudFormation templates
# Committed files to CodeCommit repository
Pipeline resource template generation successful for application java-tomcat-9e8e4799

You're all set to use AWS CloudFormation to manage your pipeline stack.

Next Steps:
1. Edit the CloudFormation template as necessary.
2. Create a pipeline stack using the AWS CLI or the AWS Console. AWS CLI command:

aws cloudformation deploy --template-file /root/app2container/java-tomcat-9e8e4799/
Artifacts/Pipeline/CodePipeline/ecs-pipeline-master.yml --capabilities
 CAPABILITY_NAMED_IAM --stack-name app2container-java-tomcat-9e8e4799-ecs-pipeline-
stack
```

The following example shows the **generate pipeline** command with the `--application-id` parameter and the `--deploy` option.

```
$ sudo app2container generate pipeline --deploy --application-id java-tomcat-9e8e4799
# Generated buildspec file(s)
# Generated CloudFormation templates
# Committed files to CodeCommit repository
# Initiated CloudFormation stack creation. This may take a few minutes. Please visit
 the AWS CloudFormation Console to track progress.
# Deployed pipeline through CloudFormation
Pipeline deployment successful for application --application-id java-tomcat-9e8e4799
```

```
Successfully created AWS CodePipeline stack 'app2container---application-id java-
tomcat-9e8e4799-ecs-pipeline-stack' for application. Check the AWS CloudFormation
 Console for additional details.
```

The following example shows the **generate pipeline** command with the `--application-id` parameter and the `--deploy` option for an application that is deployed to AWS App Runner.

```
$ sudo app2container generate pipeline --deploy --application-id java-tomcat-9e8e4799
# Created CodeCommit repository
# Generated buildspec file(s)
# Generated CloudFormation templates
# Committed files to CodeCommit repository
# Initiated CloudFormation stack creation. This may take a few minutes. To track
 progress, open the AWS CloudFormation console.
# Deployed pipeline through CloudFormation
Pipeline deployment successful for application --application-id java-tomcat-9e8e4799

Successfully created AWS CodePipeline stack 'a2c---application-id java-tomcat-9e8e4799-
ecs-pipeline-stack' for application. Check the AWS CloudFormation Console for
 additional details.
```

Windows

The following example shows the **generate pipeline** command with the `--application-id` parameter.

```
PS> app2container generate pipeline --application-id iis-smarts-51d2dbf8
# Created CodeCommit repository
# Generated buildspec file(s)
# Generated CloudFormation templates
# Committed files to CodeCommit repository
Pipeline resource template generation successful for application --application-id iis-
smarts-51d2dbf8

You're all set to use AWS CloudFormation to manage your pipeline stack.

Next Steps:
1. Edit the CloudFormation template as necessary.
2. Create a pipeline stack using the AWS CLI or the AWS Console. AWS CLI command:

aws cloudformation deploy --template-file C:\Users\Administrator\AppData\Local
\app2container\--application-id iis-smarts-51d2dbf8\Artifacts\Pipeline\CodePipeline
\ecs-pipeline-master.yml --capabilities CAPABILITY_NAMED_IAM --stack-name
 app2container---application-id iis-smarts-51d2dbf8-652becbe-ecs-pipeline-stack
```

The following example shows the **generate pipeline** command with the `--application-id` parameter and the `--deploy` option.

```
PS> app2container generate pipeline --deploy --application-id iis-smarts-51d2dbf8
# Generated buildspec file(s)
# Generated CloudFormation templates
# Committed files to CodeCommit repository
# Initiated CloudFormation stack creation. This may take a few minutes. Please visit
 the AWS CloudFormation Console to track progress.
# Deployed pipeline through CloudFormation
Pipeline deployment successful for application --application-id iis-smarts-51d2dbf8

Successfully created AWS CodePipeline stack 'app2container---application-id iis-
smarts-51d2dbf8-ecs-pipeline-stack' for application. Check the AWS CloudFormation
 Console for additional details.
```

app2container help command

Lists the commands for App2Container, grouped into the phases where they would normally run.

Note
Commands are shown in alphabetical order within the phases where they run. For example, in the *Analyze* phase, you would run the **inventory** command first, then the **analyze** command. Utility commands are included after the containerization phases.

Syntax

```
app2container help
```

Parameters and options

None

Output

The list of app2container commands.

Examples

```
app2container help
App2Container is an application from Amazon Web Services (AWS),
that provides commands to discover and containerize applications.

Commands
  Getting Started
    init                    Sets up workspace for artifacts

  Analyze
    analyze                 Analyzes the selected application to identify dependencies
 required for containerization
    inventory               Lists all applications that can be containerized

  Transform
    containerize            Generates Dockerfile, container images, and deployment metadata
    extract                 Creates an archive of application artifacts for containerization

  Deploy
    generate                Generates ECS, EKS, or Pipeline artifacts

  Settings
    upgrade                 Upgrades app2container CLI to latest version
    upload-support-bundle Uploads user's app2container logs and supporting artifacts to the
 support team

Flags
      --debug     enable debug logging
  -h, --help      help for app2container
      --version   version for app2container
```

app2container init command

The **init** command performs one-time initialization tasks for App2Container. This interactive command prompts for the information required to set up the local App2Container environment. Run this command before you run any other App2Container commands.

> **Note**
> If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.
> When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.
> Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container init [--advanced] [--help]
```

Parameters and options

Options

--advanced

This option allows you to use features that are in the experimental phase, if any exist.

--help

Displays the command help.

Output

The **init** command prompts you for the information that it needs for initialization.

You must provide a local directory for application containerization artifacts. Ensure that only authorized users can access the local directory. If you do not specify a local directory, one is created for you at the default output location. The default locations are as follows:

- **Linux:** `/root/app2container`
- **Windows:** `C:\Users\Administrator\AppData\Local\app2container`

You can optionally provide an Amazon S3 bucket for application containerization artifacts. If you choose to set up an Amazon S3 bucket, you must ensure that only authorized users can access the bucket. We recommend that you use server-side encryption for your bucket. See Protecting data using server-side encryption in the *Amazon Simple Storage Service User Guide* for more information about how to set it up.

You can optionally upload logs and command-generated artifacts automatically to App2Container support when an app2container command crashes or encounters internal errors. Log files are retained for 90 days.

You can optionally consent to allow App2Container to collect and export the following metrics to AWS each time that you run an **app2container** command:

- Host OS name
- Host OS version
- Application stack type
- Application stack version
- JDK version (Linux only, for Java applications)
- App2Container CLI version
- Command that ran
- Command status
- Command duration
- Command features and flags
- Command errors
- Container base image

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **init** command with no additional options.

```
$ sudo app2container init
Please enter a workspace directory path to use for artifacts[default: /root/
app2container]:
Please enter an AWS Profile to use. (The same can be configured with 'aws configure --
profile <name>')[default: default]:
Please provide an S3 bucket to store application artifacts (Optional):
Automatically upload logs and App2Container generated artifacts on crashes and internal
 errors? (Y/N):
Please confirm permission to report usage metrics to AWS (Y/N)[default: y]:
Would you like to enforce the use of only signed images using Docker Content Trust
 (DCT)? (Y/N)[default: n]:
All application artifacts will be created under the above workspace. Please ensure that
 the folder permissions are secure.
Init configuration saved
```

The following example shows the **init** command with the `--advanced` option and default values.

```
PS> sudo app2container init --advanced
Please enter a workspace directory path to use for artifacts[default: /root/
app2container]:
Please enter an AWS Profile to use. (The same can be configured with 'aws configure --
profile <name>')[default: default]:
Please provide an S3 bucket to store application artifacts (Optional):
Automatically upload logs and App2Container generated artifacts on crashes and internal
 errors? (Y/N):
Please confirm permission to report usage metrics to AWS (Y/N)[default: y]:
Would you like to enforce the use of only signed images using Docker Content Trust
 (DCT)? (Y/N)[default: n]:
Would you like to enable experimental features? (Y/N)[default: n]:
All application artifacts will be created under the above workspace. Please ensure that
 the folder permissions are secure.
Init configuration saved
```

Windows

The following example shows the **init** command with no additional options.

```
PS> app2container init
Please enter a workspace directory path to use for artifacts[default: C:\Users
\Administrator\AppData\Local\app2container]:
Please enter an AWS Profile to use. (The same can be configured with 'aws configure --
profile <name>')[default: default]:
Please provide an S3 bucket to store application artifacts (Optional):
Automatically upload logs and App2Container generated artifacts on crashes and internal
 errors? (Y/N):
Please confirm permission to report usage metrics to AWS (Y/N)[default: y]:
Would you like to enforce the use of only signed images using Docker Content Trust
 (DCT)? (Y/N)[default: n]:
All application artifacts will be created under the above workspace. Please ensure that
 the folder permissions are secure.
Init configuration saved
```

The following example shows the **init** command with the `--advanced` option and default values.

```
PS> app2container init --advanced
Please enter a workspace directory path to use for artifacts[default: C:\Users
\Administrator\AppData\Local\app2container]:
Please enter an AWS Profile to use. (The same can be configured with 'aws configure --
profile <name>')[default: default]:
Please provide an S3 bucket to store application artifacts (Optional):
Automatically upload logs and App2Container generated artifacts on crashes and internal
 errors? (Y/N):
Please confirm permission to report usage metrics to AWS (Y/N)[default: y]:
Would you like to enforce the use of only signed images using Docker Content Trust
 (DCT)? (Y/N)[default: n]:
Please enter if we can enable checking for upgrades automatically (Y/N)[default: y]:
Would you like to enable experimental features? (Y/N)[default: n]:
All application artifacts will be created under the above workspace. Please ensure that
 the folder permissions are secure.
Init configuration saved
```

app2container inventory command

Records all Java processes (Linux) or all IIS websites and Windows services (Windows) that are running on the application server.

Syntax

```
app2container inventory --type [iis | service | java] [--nofilter] [--help]
```

Parameters and options

Parameters

--type [*iis* | *service* | *java*]

Use this parameter to specify the application type *(required)*. For .NET applications running on Windows, you can specify an IIS web application (`iis`), or a Windows service (`service`). For Java applications running on Linux, you must specify `java`.

Options

--nofilter

For applications running on Windows, this option prevents App2Container from filtering out default system services when building the inventory output. This can be used for complex Windows .NET applications that have dependent web apps that need to be included in the container.

--help

Displays the command help.

Output

Information about the Java processes or IIS websites is saved to the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

The application ID that is used by other App2Container commands is the key for each application object in the JSON file. The application objects are slightly different depending on your application language and the application server operating system. Choose the operating system tab for your application in the Examples section below to see the differences.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

Each Java process has a unique application ID (for example, java-tomcat-9e8e4799). You can use this application ID with other AWS App2Container commands. Inventory information is saved to `/root/inventory.json`.

The following example shows the **inventory** command with no additional options.

```
$ sudo app2container inventory
{
    "java-jboss-5bbe0bec": {
        "processId": 27366,
        "cmdline": "java ...",
        "applicationType": "java-jboss"
    },
    "java-tomcat-9e8e4799": {
        "processId": 2537,
        "cmdline": "/usr/bin/java ...",
        "applicationType": "java-tomcat"
    }
}
```

Windows

Each IIS website has a unique application ID (for example, iis-smarts-51d2dbf8). You can use this application ID with other AWS App2Container commands. Inventory information is saved to C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json.

The following example shows the **inventory** command with no additional options.

```
PS> app2container inventory
{
    "iis-smarts-51d2dbf8": {
        "siteName": "Default Web Site",
        "bindings": "http/*:80:,net.tcp/808:*",
        "applicationType": "iis",
        "discoveredWebApps": []
    },
    "iis-smart-544e2d61": {
        "siteName": "smart",
        "bindings": "http/*:82:",
        "applicationType": "iis",
        "discoveredWebApps": []
    },
    "service-colorwindowsservice-69f90194": {
                "serviceName": "colorwindowsservice",
                "applicationType": "service"
 }
}
```

app2container remote analyze command

Run this command from a worker machine to analyze the specified application on the target application server, and generate a report. The target application server is specified by its IP address or Fully Qualified Domain Name (FQDN).

Note
If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.
When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.
Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container remote analyze --application-id id --target IP/FQDN [--help]
```

Parameters and options

Parameters

--application-id *id*

The application ID *(required)*. After you run the remote inventory (p. 102) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `<workspace>/remote/<target server IP or FQDN>/inventory.json`
- **Windows:** `<workspace>\remote\<target server IP or FQDN>\.app2container-config\inventory.json`

--target *IP/FQDN*

Specifies the IP address or FQDN of the application server targeted for the inventory *(required)*.

Options

--help

Displays the command help.

Output

The **remote analyze** command creates files and directories on the worker machine for the specified application on the target application server. Each application has its own directory, named for the application ID. Output varies slightly, depending on your application language and the application server operating system.

The application directory contains analysis output and editable application configuration files. The files are stored in subdirectories that match the application structure on the application server.

An `analysis.json` file is created for the application that is specified in the `--application-id` parameter. The file contains information about the application found during analysis, and configurable fields for container settings. See Java application analysis file (p. 29), or .NET application analysis file (p. 32) for more information about configurable fields, and for an example of what the file looks like.

For .NET applications, App2Container detects connection strings and produces the `report.txt` file. The report location is specified in the `analysis.json` file, in the `reportPath` attribute of the `analysisInfo` section. You can use this report to identify the changes that you need to make in application configuration files to connect your application container to new database endpoints, if needed. The report also contains the locations of other configuration files that might need changes.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **remote analyze** command with the `--target` and `--application-id` parameters and no additional options.

```
$ sudo app2container remote analyze --target  192.0.2.0 --application-id java-
tomcat-9e8e4799
  Analysis successful for application java-tomcat-9e8e4799

   Next Steps:
1. View the application analysis file at <workspace>/remote/<target server IP or FQDN>/
java-tomcat-9e8e4799/analysis.json.
2. Edit the application analysis file as needed.
3. Start the extraction process using the following command: app2container remote
 extract --target  192.0.2.0 java-tomcat-9e8e4799
```

Windows

The following example shows the **remote analyze** command with the `--target` and `--application-id` parameters and no additional options.

```
PS> app2container remote analyze --target  192.0.2.0 --application-id iis-
smarts-51d2dbf8
  Analysis successful for application iis-smarts-51d2dbf8;

   Next Steps:
```

```
1. View the application analysis file at <workspace>\remote\<target server IP or FQDN>/
iis-smarts-51d2dbf8/analysis.json.
2. Edit the application analysis file as needed.
3. Start the extraction process using the following command: app2container remote
 extract --target  192.0.2.0 iis-smarts-51d2dbf8
```

app2container remote configure command

Run this command from a worker machine to configure the connections needed to run remote workflows on application servers. This interactive command prompts for the required information for each application server that you enter, or you can provide a JSON input file with your connection information by specifying the `--input-json` parameter when you run the command.

Note
For the remote configure command prompts, if you specify the Fully Qualified Domain Name (FQDN), the server IP address is optional and is not used by App2Container.

Syntax

```
app2container remote configure [--input-json myhosts.json] [--help]
```

Parameters and options

Parameters

--input-json

Uses the provided JSON file as input to configure connections to application servers for the worker machine to run remote commands.

Options

--help

Displays the command help.

Input

To see the input file format, choose the system platform that matches your configuration. For key/value pairs that do not apply to your configuration, set string values to an empty string.

Linux remote hosts file

The Linux `remote_hosts.json` file contains an array of Linux platform hosts, with connection information. The key for each host is the host IP address or FQDN, with an array of strings for the connection information. Each host includes the following content:

- **Fqdn** (string, conditionally required) – the fully qualified domain name of the host, used as the identifier for connecting. *If an IP address is used as the host identifier, this must be empty. If the FQDN has a value, the IP address is ignored.*
- **Ip** (string, conditionally required) – the IP address of the host, used as the identifier for connecting. *Required if the FQDN is empty.*

- **SecretArn** (string, required) – the Amazon Resource Name (ARN) that identifies the Secrets Manager secret to use for credentials.
- **AuthMethod** (string, required) – the authentication method used to connect to the host. *Valid values include "cert" and "key".*

The following example shows a `remote_hosts.json` file for a Java application running on Linux.

```
{
        "10.10.10.10": {
                "Fqdn": "",
                "Ip": "10.10.10.10",
                "SecretArn": "arn:aws:secretsmanager:us-west-2:123456789012:secret:linux-
cert-Abcdef",
                "AuthMethod": "cert"
        },
        "myhost.mydomain.com": {
                "Fqdn": "myhost.mydomain.com",
                "Ip": "",
                "SecretArn": "arn:aws:secretsmanager:us-west-2:987654321098:secret:linux-
cert-Ghijkl",
                "AuthMethod": "key"
        }
}
```

Windows remote hosts file

The Windows `remote_hosts.json` file contains an array of Windows Server platform hosts, with connection information. The key for each host is the host IP address or FQDN, with an array of strings for the connection information. Each host includes the following content:

- **fqdn** (string, conditionally required) – the fully qualified domain name of the host, used as the identifier for connecting. *If an IP address is used as the host identifier, this must be empty. If the FQDN has a value, the IP address is ignored.*
- **ip** (string, conditionally required) – the IP address of the host, used as the identifier for connecting. *Required if the FQDN is empty.*
- **secretArn** (string, required) – the Amazon Resource Name (ARN) that identifies the Secrets Manager secret to use for credentials.

The following example shows a `remote_hosts.json` file for a .NET application running on Windows Server.

```
{
        "10.10.10.10": {
                "fqdn": "",
                "ip": "10.10.10.10",
                "secretArn": "arn:aws:secretsmanager:us-west-2:123456789012:secret:windows-
cred-Abcdef"
        }
}
```

Output

This command does not produce a configurable output file. For troubleshooting purposes, or if you need to verify what was entered during the interactive command dialog, you can find the entries in the `remote_hosts.json` file by searching the folder structure on the server where you ran the command.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **remote configure** command with no additional options.

```
$ sudo app2container remote configure
Server IP address: 10.10.10.10
Server FQDN (Fully Qualified Domain Name):
Authentication method to be used key/cert: cert
Secret ARN for remote connection credentials: arn:aws:secretsmanager:us-
west-2:123456789012:secret:linux-cert-Abcdef
Continue to configure servers? (y/N)[default: n]: y
Server IP address:
Server FQDN (Fully Qualified Domain Name): fqdn2
Authentication method to be used key/cert: key
Secret ARN for remote connection credentials: arn:aws:secretsmanager:us-
west-2:987654321098:secret:linux-cert-Ghijkl
Continue to configure servers? (y/N)[default: n]: n
```

Windows

The following example shows the **remote configure** command with no additional options.

```
PS> app2container remote configure
Server IP address: 10.10.10.10
Server FQDN (Fully Qualified Domain Name):
Authentication method to be used key/cert: cert
Secret ARN for remote connection credentials: arn:aws:secretsmanager:us-
west-2:123456789012:secret:windows-cred-Abcdef
Continue to configure servers? (y/N)[default: n]: y
Server IP address:
Server FQDN (Fully Qualified Domain Name): fqdn2
Authentication method to be used key/cert: key
Secret ARN for remote connection credentials: arn:aws:secretsmanager:us-
west-2:987654321098:secret:windows-cred-Ghijkl
Continue to configure servers? (y/N)[default: n]: n
```

app2container remote extract command

Run this command from a worker machine to generate an application archive for the specified application on the target application server. The target application server is specified by its IP address or Fully Qualified Domain Name (FQDN). Before you call this command, you must call the remote analyze (p. 96) command.

Note

If the command fails, an error message is displayed in the console, followed by additional messaging to help you troubleshoot.
When you ran the **init** command, if you chose to automatically upload logs to App2Container support if an error occurs, App2Container notifies you of the success of the automatic upload of your application support bundle.
Otherwise, App2Container messaging directs you to upload application artifacts by running the upload-support-bundle (p. 104) command for additional support.

Syntax

```
app2container remote extract --application-id id --target IP/FQDN [--output s3] [--help]
```

Parameters and options

--application-id *id*

The application ID *(required)*. After you run the remote inventory (p. 102) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `<workspace>/remote/<target server IP or FQDN>/inventory.json`
- **Windows:** `<workspace>\remote\<target server IP or FQDN>\.app2container-config\inventory.json`

--target *IP/FQDN*

Specifies the IP address or FQDN of the application server targeted for the inventory *(required)*.

--profile *admin-profile*

Use this option to specify a *named profile* to run this command. For more information about named profiles in AWS, see Named profiles in the AWS Command Line Interface User Guide

Options

--output s3

If specified, this option writes the archive file to an Amazon S3 bucket that you specified when you ran the **init** command.

--force

Bypasses the disk space prerequisite check.

--help

Displays the command help.

Output

This command creates an archive file. When you use the `--output s3` option, the archive is written to the Amazon S3 bucket that you specified when you ran the **init** command. Otherwise, the archive is written to the output location that you specified when you ran the **init** command.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

The following example shows the **remote extract** command with the `--target` and `--application-id` parameters and no additional options.

```
$ sudo app2container extract --target  192.0.2.0 --application-id java-tomcat-9e8e4799
```

```
Extraction successful for application java-tomcat-9e8e4799
   Next Steps:
1. Please initiate containerization using "app2container containerize --input-
archive <workspace>/remote/<target server IP or FQDN>/java-tomcat-9e8e4799/java-
tomcat-9e8e4799-extraction.tar"
```

Windows

The following example shows the **remote extract** command with the `--target` and `--application-id` parameters and no additional options.

```
PS> app2container extract --target  192.0.2.0 --application-id iis-smarts-51d2dbf8
Extraction successful for application iis-smarts-51d2dbf8

   Next Steps:
1. Please initiate containerization using "app2container containerize --input-
archive <workspace>\remote\<target server IP or FQDN>/iis-smarts-51d2dbf8/iis-
smarts-51d2dbf8.zip"
```

app2container remote inventory command

Run this command from a worker machine to retrieve an inventory of all running Java processes (Linux) or all IIS websites (Windows) on the application server specified in the `--target` parameter. The target application server is specified by its IP address or Fully Qualified Domain Name (FQDN). The inventory details are captured in the `inventory.json` file and stored on the worker machine under the target server folder.

Syntax

```
app2container remote inventory --target IP/FQDN [--help]
```

Parameters and options

Parameters

--target *IP/FQDN*

Specifies the IP address or FQDN of the application server targeted for the inventory *(required)*.

Options

--help

Displays the command help.

Output

Information about the Java processes or IIS websites is saved to the `inventory.json` file in one of the following locations:

- **Linux:** `<workspace>/remote/<target server IP or FQDN>/inventory.json`

- **Windows:** `<workspace>\remote\`*`<target server IP or FQDN>`*`\.app2container-config\inventory.json`

The application ID that is used by other App2Container commands is the key for each application object in the JSON file. The application objects are slightly different depending on your application language and the application server operating system. Choose the operating system tab for your application in the Examples section below to see the differences.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

Each Java process has a unique application ID (for example, java-tomcat-9e8e4799). You can use this application ID with other AWS App2Container commands. Inventory information is saved to `/root/inventory.json`.

The following example shows the **remote inventory** command with the `--target` parameter.

```
$ sudo app2container remote inventory --target IP/FQDN
: Retrieving inventory from remote server 192.0.2.0
# Server inventory has been stored under <workspace>/remote/<target server IP or FQDN>/
inventory.json
##Remote inventory retrieved successfully
```

Windows

Each IIS website has a unique application ID (for example, iis-smarts-51d2dbf8). You can use this application ID with other AWS App2Container commands. Inventory information is saved to C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json.

The following example shows the **remote inventory** command with the `--target` parameter.

```
PS> app2container remote inventory --target IP/FQDN
: Retrieving inventory from remote server 192.0.2.0
# Server inventory has been stored under <workspace>\remote\<target server IP or
 FQDN>\inventory.json
##Remote inventory retrieved successfully
```

app2container upgrade command

Run this command to upgrade your existing installation of App2Container.

If a newer version of AWS App2Container will break backwards compatibility with previously generated container artifacts when you do an upgrade, the upgrade command notifies you and requests permission to continue. If you choose to continue with the upgrade, you will be required to restart any ongoing analysis and containerization workflows for your applications.

Syntax

```
app2container upgrade [--help]
```

Options

--help

Displays the command help.

Output

Console output is included in the Examples section for this command.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

Run the command shown below to upgrade your existing App2Container for Linux.

```
$ sudo app2container upgrade
Using version 1.0
Version 1.1 available for download
Starting Download...
Starting Installation...
Installation successful!
```

Windows

Run the command shown below to upgrade your existing App2Container for Windows.

```
PS> app2container upgrade
Using version 0.0
Version 2.0 available for download  Starting Download...
Starting Installation...Installation successful!
```

app2container upload-support-bundle command

For assistance with troubleshooting, run this command to securely upload App2Container logs and supporting artifacts to the AWS App2Container support team. The following list shows the types of files that you can upload with the **upload-support-bundle** command:

- App2Container logs
- The `analysis.json` file
- The `Dockerfile`
- The `deployment.json` file
- The `EcsDeployment.yml` or `ecs-master.yml` deployment artifacts

Syntax

```
app2container upload-support-bundle [--application-id id] [--support-message "message"] [--
help]
```

Options

--application-id *id*

The application ID *(required)*. After you run the inventory (p. 94) command, you can find the application ID in the `inventory.json` file in one of the following locations:

- **Linux:** `/root/inventory.json`
- **Windows:** `C:\Users\Administrator\AppData\Local\.app2container-config\inventory.json`

--support-message

Include a message for the App2Container support team with your bundle.

--help

Displays the command help.

Output

Console output is included in the Examples section for this command.

Examples

Choose the operating system tab for the application server or worker machine where you run the command.

Linux

Run the following command to upload a support bundle from a Linux operating system, including the application ID and a message for the support team.

```
$ sudo app2container upload-support-bundle --application-id java-tomcat-9e8e4799 --
support-message "I ran into an issue during deployment ..."
Support Message: I ran into an issue during deployment ...
[displays while bundle is uploading] Uploading logs and supporting artifacts to
 App2Container support
Support bundle upload successful
```

Windows

Run the following command to upload a support bundle from a Windows operating system, including the application ID and a message for the support team.

```
PS> app2container upload-support-bundle --application-id iis-smarts-51d2dbf8 --support-
message "I ran into an issue during deployment ..."
Support Message: I ran into an issue during deployment ...
[displays while bundle is uploading] Uploading logs and supporting artifacts to
 App2Container support
Support bundle upload successful
```

Troubleshooting App2Container issues

The following documentation can help you troubleshoot problems that you might have with the App2Container CLI.

Contents

Access App2Container logs on your server

A common first step in troubleshooting issues with any application is reviewing application logs. App2Container logs contain a history of the information and error messages that are produced by the commands that you run. If you opted out of metrics during initialization, the metrics messages are also logged in the local application log file.

Review log files in one of the following locations, depending on where you are running the command that needs troubleshooting:

Application logs

- **Linux:** `/root/app2container/log/app2container.log`
- **Windows:** `C:\Users\Administrator\AppData\Local\app2container\log\app2container.log`

Upgrade logs

- **Linux:** `/usr/local/app2container/log/app2container_upgrade.log`
- **Windows:** `C:\Users\Administrator\app2container\log\app2container_upgrade.log`

If there is more than one log file, it means that the first log file reached its maximum size, and a new log file was created to continue logging. Choose the most recent log file to troubleshoot.

Access application logs inside of a running container

You can access application logs on your running container by running a command shell from the container host that attaches to your container. Choose the tab that matches your container operating system to see the command.

Linux

From the host server, run an interactive bash shell on your running container.

```
$ docker exec -it container-id bash
```

Using the bash shell, you can then navigate to the location where your application logs are stored.

Windows

From the host server, run an interactive PowerShell session attached to your running container.

```
PS> docker exec -it container-id powershell.exe
```

Using the PowerShell session, you can then navigate to the location where your application logs are stored.

To look up Docker commands, use the Docker command line reference. See Use the Docker command line.

AWS resource creation fails for the generate command

Description

When you run the **generate app-deployment** or **generate pipeline** command, you receive an error message saying AWS resource creation has failed.

Cause

App2Container requires permission to access and create AWS resources when it generates and deploys application containers or pipelines. If the permission has not been configured in your IAM policy, or if you are using the default AWS profile for a command using the `--deploy` option, the command will fail.

Solution

Verify your IAM resources and AWS profile settings and adjust as necessary, depending on the command that failed and the details shown in your error message. For more information and instructions about how to set up IAM resources for App2Container, see Identity and access management in App2Container (p. 63).

Troubleshoot Java applications on Linux

This section contains issues you might have with using App2Container for Java applications running on Linux servers.

Application container image size is very large

Description

Your application container image is much larger than expected.

Cause

The application container image includes a kernel image with the application bits layered on top. The size of the image depends on both the size of the container operating system and the size of the application.

To catch all potential dependencies for Java applications on Linux that are not using JBoss or Tomcat frameworks, the container initially includes everything except the files that are already included in the kernel image.

Solution

Follow these steps to reduce the size of your application container image.

1. Use the `appExcludedFiles` section in your `analysis.json` file to exclude specific file and directory paths from the containerization process, and save the file when you are done.
2. Run the **containerize (p. 77)** command again to create a new application container image with the updates that you specified.

You can repeat this process as needed to further reduce the size.

Error: Insufficient disk space

Description

When you run the **containerize** command, it fails with the following error message: Error: Insufficient disk space.

Cause

For Java applications running on Linux, App2Container calculates the disk space that is required to generate the application container, and produces this error message if there is not enough free space. The calculation includes the space needed for the application archive (including all non-system files on the server), plus the space needed for **docker build** actions.

Solution

The error message generated by the **containerize** command includes the estimated space it needs to run successfully. There are many ways to address an insufficient space issue on Linux.

One way to ensure that your **containerize** command runs successfully is to reduce the size of the container that you are creating. Follow these steps to reduce the size of your application container image.

1. Use the `appExcludedFiles` section in your `analysis.json` file to exclude specific file and directory paths from the containerization process, and save the file when you are done.
2. Run the **containerize (p. 77)** command again to create a new application container image with the updates that you specified.

You can repeat this process as needed to further reduce the size.

Troubleshoot .NET applications on Windows

This section contains issues you might have with using App2Container for .NET applications running in IIS on Windows servers.

Application container image size is very large

Description

Your application container image is much larger than expected.

Cause

The application container image includes a kernel image with the application bits layered on top. The size of the image depends on both the size of the container operating system and the size of the application. The Windows Server Core image can be quite large, especially for versions prior to Windows Server Core 2019.

Solution

We recommend that you use Windows Server Core 2019 for your container operating system to create the smallest base container size possible.

Follow these steps to reduce the size of your application container image if you are not currently using Windows Server Core 2019 as your base image. To ensure that you get the correct version, specify the version tag as shown below. The repository for Windows base images does not support the concept of "latest" to target the most recent image version.

1. Use the `containerBaseImage` section in your `analysis.json` file to target the Windows Server Core 2019 base image tagged as `ltsc2019` and save the file when you are done.

   The `containerBaseImage` value includes both the image name and the `ltsc2019` tag, separated by a colon (:). For example: `"containerBaseImage": "mcr.microsoft.com/dotnet/framework/aspnet:4.7.2-windowsservercore-ltsc2019"`.
2. Run the **containerize (p. 77)** command again to create a new application container image. It will use the container operating system image that you specify in the `containerBaseImage` of your `analysis.json` file to build a new application container image.

Release notes for AWS App2Container

The following table describes the release history for AWS App2Container in descending date order:

Release date	Version	Details
July 26, 2021	1.5	• Added support for containerizing complex Windows applications. • Added support for Amazon EKS tagging. • Added support for pipeline tagging. • Bug fixes, including: • Reclassified DockerInvalidImageError for clarity. • Fixed App Runner deployment error that occurs when the local App Runner container is still running at deployment time. • Fixed containerization error when analysis.json has escaped characters.
May 20, 2021	1.4	• Added support for deployments to AWS App Runner. • Added support for reuse of existing Active Directory security groups with gMSA. • Bug fixes, including: • Fixed containerization of Windows applications that use a secondary drive mount. • Clarified some messaging on actionable errors. • Fixed an issue that resulted in incorrect analysis of RHEL Java processes. • Fixed issues related to symbolic links in the application server that resulted in larger than necessary image sizes.
March 29, 2021	1.3	• Added support for Amazon EC2 instance profiles. • Added support for Amazon EC2 Nitro instance types in App2Container deployments for Windows applications. • Added support for Windows Server Core Version 2004 base images for containerized Windows applications. • The container base image for Windows applications now defaults to match the OS version for the server that runs containerization. • The base image for Windows Amazon ECS deployment artifacts matches the container base image. • Bug fixes, including: • Fixed issue related to symbolic links during containerization. • Fixed issue related to spaces in paths in Dockerfiles. • Fixed issues related to the Windows remote setup script.
December 21, 2020	1.2	• Added capability to run commands remotely. • Added custom tag support for deployment resources. • Added support for HTTPS endpoints and ACM-based certificate management for Amazon ECS deployments.

Release date	Version	Details
		• Bug fixes, including: • Fixed containerization issue on Linux when unidentified base image uses default image. • Added exclusion for AWS credentials when containerizing Linux applications.
November 24, 2020	1.1	• Added support for Active Directory authenticated Windows application deployments to Amazon EKS using gMSA. • Added support for named profile overrides to commands that interact with AWS. • Enabled automatic log upload and adjusted console messaging when errors occur (requires IAM policy update). • Added capability to manually upload logs and other artifacts with the **upload-support-bundle** command (requires IAM policy update). • Bug fixes, including: • Updated CloudFormation templates for Amazon EKS deployments to address previous issues. • Fixed internal/user error classification. • Fixed upgrade errors to point to the correct log file. • Added an explicit check to validate use of Docker version 17.07 or above.
September 15, 2020	1.0.2	• Added FireLens logging support. • Added container image validation to pipeline generation. • Bug fixes, including: • Removed execution role in template if Windows is specified. • Fixed template to reflect CloudFormation API change. • Fixed autocomplete installation bug. • Fixed dark font for Windows errors. • Improved error messaging for command execution errors. • Fixed containerize error where included file is not valid.

Release date	Version	Details
August 5, 2020	1.0.1	• Improved memory usage while archiving on Windows. • Added support for containerizing individual applications running in Tomcat and JBoss standalone frameworks. • Added schema version and unhealthy version checks. • Bug fixes, including: • Fixed handling for .NET Windows app running on alternative drives (not C). • Fixed COPY command failure in DockerFile. • Access denied error now throws user error. • Added automatic removal of characters that are not allowed in AppId. • Optimized Windows container image size for websites with multiple apps. • Fixed error handling for input arguments validation. • Fixed Dockerfile generation failure when dynamic logging is enabled. • EKS CloudFormation templates are now compatible with the new CloudFormation custom resource API.
June 30, 2020	1.0.0	• Initial release

Document history for AWS App2Container

The following table describes important changes to the documentation by date. For notification about updates to this documentation, you can subscribe to an RSS feed.

update-history-change	update-history-description	update-history-date
Minor release – v1.5 (p. 113)	This minor release includes the following changes, along with miscellaneous bug fixes: support for containerizing complex Windows applications, and tagging support for Amazon EKS and pipelines.	July 26, 2021
Minor release – v1.4 (p. 113)	This minor release includes the following changes, along with miscellaneous bug fixes: support for deployments to AWS App Runner, and support for reuse of existing Active Directory security groups with gMSA.	May 20, 2021
Minor release – v1.3 (p. 113)	This minor release includes the following changes, along with miscellaneous bug fixes: support for Amazon EC2 instance profiles, and enhancements for Windows application containers (support for Amazon EC2 Nitro instance types in App2Container deployments, support for Windows Server Core Version 2004 base images, container base image defaults to match the OS version for the server that runs containerization, and Amazon ECS deployment artifacts to match the container base image).	March 29, 2021
Docs-only: applicationMode settings (p. 113)	Describe container configuration applicationMode settings in more detail.	March 19, 2021
Docs-only: IAM policy sections (p. 113)	Add content to describe optional sections of the IAM policy templates.	February 18, 2021
Docs-only: IAM update (p. 113)	Update IAM policy examples for Amazon EKS and Amazon	January 12, 2021

	ECS to reflect recent changes and adjust S3 section to remove problematic permission.	
Minor release – v1.2 (p. 113)	This minor release includes the following changes, along with miscellaneous bug fixes: capability to run commands remotely, custom tag support for deployment resources, support for HTTPS endpoints and ACM-based certificate management for Amazon ECS deployments, and exclusion of AWS credentials when containerizing Linux applications. Note: remote command capability requires an IAM policy update.	December 21, 2020
Minor release – v1.1 (p. 113)	This minor release includes the following changes, along with miscellaneous bug fixes: Added support for Amazon EKS gMSA, introduced named profile overrides for commands that interact with AWS, enabled automatic log uploads for command failures, and added a command to upload a support bundle for help with troubleshooting from App2Container support. Note: uploads for log and support file bundles require an IAM policy update.	November 24, 2020
Patch release – v1.0.2 (p. 113)	Added FireLens logging support, plus patches for AWS App2Container version 1.0.2.	September 15, 2020
Patch release – v1.0.1 (p. 113)	Added Release notes page with version 1.0.1 changes for AWS App2Container.	August 5, 2020
Docs-only: configuration and IAM updates (p. 113)	A chapter was added to describe configurable fields in files generated by App2Container commands, and the security section was updated with an IAM best practices summary and guidance for setting up IAM general use resources for App2Container.	August 1, 2020
Initial release (p. 113)	This release introduces AWS App2Container.	June 30, 2020