Advanced Environment Customization with Configuration Files (.ebextensions)

Advanced Environment Customization with Configuration Files (.ebextensions)

You can add AWS Elastic Beanstalk configuration files (`.ebextensions`) to your web application's source code to configure your environment and customize the AWS resources that it contains. Configuration files are YAML- or JSON-formatted documents with a `.config` file extension that you place in a folder named `.ebextensions` and deploy in your application source bundle.

We recommend using YAML for your configuration files, because it's more readable than JSON. YAML supports comments, multi-line commands, several alternatives for using quotes, and more. However, you can make any configuration change in Elastic Beanstalk configuration files identically using either YAML or JSON.

> **Tip**
>
> When you are developing or testing new configuration files, launch a clean environment running the default application and deploy to that. Poorly formatted configuration files will cause a new environment launch to fail unrecoverably.

The `option_settings` section of a configuration file defines values for configuration options. Configuration options let you configure your Elastic Beanstalk environment, the AWS resources in it, and the software that runs your application. Configuration files are only one of several ways to set configuration options.

The Resources section lets you further customize the resources in your application's environment, and define additional AWS resources beyond the functionality provided by configuration options. You can add and configure any resources supported by AWS CloudFormation, which Elastic Beanstalk uses to create environments.

The other sections of a configuration file (`packages`, `sources`, `files`, `users`, `groups`, `commands`, `container_commands`, and `services`) let you configure the EC2 instances that are launched in your environment. Whenever a server is launched in your environment, Elastic Beanstalk runs the operations defined in these sections to prepare the operating system and storage system for your application.

Requirements

- **Location** – Place all of your configuration files in a single folder, named `.ebextensions`, in the root of your source bundle. Folders starting with a dot can be hidden by file browsers, so make sure that the folder is added when you create your source bundle. See Create an Application Source Bundle for instructions.
- **Naming** – Configuration files must have the `.config` file extension.
- **Formatting** – Configuration files must conform to YAML or JSON specifications. When using YAML, always use spaces to indent keys at different nesting levels. For more information about YAML, see YAML Ain't Markup Language (YAML™) Version 1.1.
- **Uniqueness** – Use each key only once in each configuration file.

> **Warning**
>
> If you use a key (for example, `option_settings`) twice in the same configuration file, one of the sections will be dropped. Combine duplicate sections into a single section, or place them in separate configuration files.

Option Settings

You can use the `option_settings` key to modify the Elastic Beanstalk configuration and define variables that can be retrieved from your application using environment variables. Some namespaces allow you to extend the number of parameters, and specify the parameter names. For a list of namespaces and configuration options, see Configuration Options.

Option settings can also be applied directly to an environment during environment creation or an environment update. Settings applied directly to the environment override the settings for the same options in configuration files. If you remove settings from an environment's configuration, settings in configuration files will take effect. See Precedence for details.

Syntax

The standard syntax for option settings is an array of objects, each having a `namespace`, `option_name` and `value` key.

```
option_settings:
  - namespace:  namespace
    option_name:  option name
    value:  option value
  - namespace:  namespace
    option_name:  option name
    value:  option value
```

The `namespace` key is optional. If you do not specify a namespace, the default used is `aws:elasticbeanstalk:application:environment`:

```
option_settings:
  - option_name:  option name
    value:  option value
  - option_name:  option name
    value:  option value
```

Examples

The following examples set a Tomcat platform-specific option in the `aws:elasticbeanstalk:container:tomcat:jvmoptions` namespace and an environment property named `MYPARAMETER`. In standard YAML format:

Example .ebextensions/options.config

```
option_settings:
  - namespace:  aws:elasticbeanstalk:container:tomcat:jvmoptions
    option_name:  Xmx
    value:  256m
  - option_name: MYPARAMETER
    value: parametervalue
```

Elastic Beanstalk also supports a shorthand syntax for option settings that lets you specify options as key-value pairs underneath the namespace:

```
option_settings:
  namespace:
    option name: option value
```

In shorthand format:

Example .ebextensions/options.config

Note that default namespace for an option is aws:elasticbeanstalk:application:environment

```
option_settings:
  aws:elasticbeanstalk:container:tomcat:jvmoptions:
    Xmx: 256m
  aws:elasticbeanstalk:application:environment:
    MYPARAMETER: parametervalue
```

In JSON:

Example .ebextensions/options.config

```
{
  "option_settings": [
    {
      "namespace": "aws:elasticbeanstalk:container:tomcat:jvmoptions",
      "option_name": "Xmx",
      "value": "256m"
    },
    {
      "option_name": "MYPARAMETER",
      "value": "parametervalue"
    }
  ]
}
```

Note : Options specified in .ebextensions have the lowest level of precedence and are overridden by settings at any other level.

Precedence

During environment creation, configuration options are applied from multiple sources with the following precedence, from highest to lowest:

- **Settings applied directly to the environment** – Settings specified during a create environment or update environment operation on the Elastic Beanstalk API by any client, including the AWS Management Console, EB CLI, AWS CLI, and SDKs. The AWS Management Console and EB CLI also apply recommended values for some options that apply at this level unless overridden.
- **Saved Configurations** – Settings for any options that are not applied directly to the environment are loaded from a saved configuration, if specified.
- **Configuration Files (.ebextensions)** – Settings for any options that are not applied directly to the environment, and also not specified in a saved configuration, are loaded from configuration files in the `.ebextensions` folder at the root of the application source bundle.
  Configuration files are executed in alphabetical order. For example, `.ebextensions/01run.config` is executed before `.ebextensions/02do.config`.
- **Default Values** – If a configuration option has a default value, it only applies when the option is not set at any of the above levels.

You can use the `Resources` key in a configuration file to create and customize AWS resources in your environment. Resources defined in configuration files are added to the AWS CloudFormation template used to launch your environment. All AWS CloudFormation resources types are supported.

For example, the following configuration file adds an Auto Scaling lifecycle hook to the default Auto Scaling group created by Elastic Beanstalk:

`~/my-app/.ebextensions/as-hook.config`

```
Resources:
  hookrole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument: {
                "Version" : "2012-10-17",
                "Statement": [ {
                   "Effect": "Allow",
                   "Principal": {
                      "Service": [ "autoscaling.amazonaws.com" ]
                   },
                   "Action": [ "sts:AssumeRole" ]
                } ]
            }
      Policies: [ {
                "PolicyName": "SNS",
                "PolicyDocument": {
                       "Version": "2012-10-17",
                       "Statement": [{
                           "Effect": "Allow",
                           "Resource": "*",
                           "Action": [
                               "sqs:SendMessage",
                               "sqs:GetQueueUrl",
                               "sns:Publish"
                           ]
                         }
                       ]
                   }
                } ]
  hooktopic:
    Type: AWS::SNS::Topic
    Properties:
      Subscription:
        - Endpoint: "my-email@example.com"
          Protocol: email
  lifecyclehook:
    Type: AWS::AutoScaling::LifecycleHook
    Properties:
      AutoScalingGroupName: { "Ref" : "AWSEBAutoScalingGroup" }
      LifecycleTransition: autoscaling:EC2_INSTANCE_TERMINATING
      NotificationTargetARN: { "Ref" : "hooktopic" }
      RoleARN: { "Fn::GetAtt" : [ "hookrole", "Arn"] }
```

Container Commands

You can use the `container_commands` key to execute commands that affect your application source code. Container commands run after the application and web server have been set up and the application version archive has been extracted, but before the application version is deployed. Non-container commands and other customization operations are performed prior to the application source code being extracted.

The specified commands run as the root user, and are processed in alphabetical order by name. Container commands are run from the staging directory, where your source code is extracted prior to being deployed to the application server. Any changes you make to your source code in the staging directory with a container command will be included when the source is deployed to its final location.

You can use `leader_only` to only run the command on a single instance, or configure a `test` to only run the command when a test command evaluates to `true`. Leader-only container commands are only executed during environment creation and deployments, while other commands and server customization operations are performed every time an instance is provisioned or updated. Leader-only container commands are not executed due to launch configuration changes, such as a change in the AMI Id or instance type.

Syntax

```
container_commands:
  name of container_command:
    command: "command to run"
    leader_only: true
  name of container_command:
    command: "command to run"
```

Example Snippet

The following is an example snippet.

```
container_commands:
  collectstatic:
    command: "django-admin.py collectstatic --noinput"
  01syncdb:
    command: "django-admin.py syncdb --noinput"
    leader_only: true
  02migrate:
    command: "django-admin.py migrate"
    leader_only: true
  99customize:
    command: "scripts/customize.sh"
```

Commands

You can use the `commands` key to execute commands on the EC2 instance. The commands run before the application and web server are set up and the application version file is extracted.

The specified commands run as the root user, and are processed in alphabetical order by name. By default, commands run in the root directory. To run commands from another directory, use the `cwd` option.

To troubleshoot issues with your commands, you can find their output in instance logs.

Syntax

```
commands:
  command name:
    command: command to run
    cwd: working directory
    env:
      variable name: variable value
    test: conditions for command
    ignoreErrors: true
```

Example Snippet

The following example snippet runs a python script.

```
commands:
  python_install:
    command: myscript.py
    cwd: /home/ec2-user
    env:
      myvarname: myvarvalue
    test: "[ -x /usr/bin/python ]"
```

Files

You can use the `files` key to create files on the EC2 instance. The content can be either inline in the configuration file, or the content can be pulled from a URL. The files are written to disk in lexicographic order.

You can use the `files` key to download private files from Amazon S3 by providing an instance profile for authorization.

Syntax

```
files:
  "target file location on disk":
     mode: "six-digit octal value"
     owner: name of owning user for file
     group: name of owning group for file
     source: URL
     authentication: authentication name:

  "target file location on disk":
     mode: "six-digit octal value"
     owner: name of owning user for file
     group: name of owning group for file
     content: |
      # this is my
      # file content
     encoding: encoding format
     authentication: authentication name:
```

Example Snippet

```
files:
  "/home/ec2-user/myfile" :
    mode: "000755"
    owner: root
    group: root
    source: http://foo.bar/myfile

  "/home/ec2-user/myfile2" :
    mode: "000755"
    owner: root
    group: root
    content: |
      this is my
      file content
```

Example using a symlink. This creates a link `/tmp/myfile2.txt` that points at the existing file `/tmp/myfile1.txt`.

```
files:
  "/tmp/myfile2.txt" :
    mode: "120400"
    content: "/tmp/myfile1.txt"
```

The following example uses the Resources key to add an authentication method named S3Auth and uses it to download a private file from an Amazon S3 bucket:

```
Resources:
  AWSEBAutoScalingGroup:
    Metadata:
      AWS::CloudFormation::Authentication:
        S3Auth:
          type: "s3"
          buckets: ["elasticbeanstalk-us-west-2-123456789012"]
          roleName:
            "Fn::GetOptionSetting":
              Namespace: "aws:autoscaling:launchconfiguration"
              OptionName: "IamInstanceProfile"
              DefaultValue: "aws-elasticbeanstalk-ec2-role"

files:
  "/tmp/data.json" :
    mode: "000755"
    owner: root
    group: root
    authentication: "S3Auth"
    source: https://s3-us-west-2.amazonaws.com/elasticbeanstalk-us-west-2-123456789012/data.json
```

Services

You can use the `services` key to define which services should be started or stopped when the instance is launched. The `services` key also allows you to specify dependencies on sources, packages, and files so that if a restart is needed due to files being installed, Elastic Beanstalk takes care of the service restart.

Syntax

```
services:
  sysvinit:
    name of service:
      enabled: "true"
      ensureRunning: "true"
      files:
        - "file name"
      sources:
        - "directory"
      packages:
        name of package manager:
          "package name[: version]"
      commands:
        - "name of command"
```

Example Snippet

The following is an example snippet:

```
services:
  sysvinit:
    myservice:
      enabled: true
      ensureRunning: true
```

Packages

You can use the `packages` key to download and install prepackaged applications and components.

Example Snippet

The following snippet specifies a version URL for rpm, requests the latest version from yum, and version 0.10.2 of chef from rubygems.

```
packages:
  yum:
    libmemcached: []
    ruby-devel: []
    gcc: []
  rpm:
    epel: http://download.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rpm
  rubygems:
    chef: '0.10.2'
```

Groups

You can use the `groups` key to create Linux/UNIX groups and to assign group IDs. To create a group, add a new key-value pair that maps a new group name to an optional group ID. The groups key can contain one or more group names. The following table lists the available keys.

Example Snippet

The following snippet specifies a group named groupOne without assigning a group ID and a group named groupTwo that specified a group ID value of 45.

```
groups:
  groupOne: {}
  groupTwo:
    gid: "45"
```

Users

You can use the `users` key to create Linux/UNIX users on the EC2 instance.

Users are created as noninteractive system users with a shell of `/sbin/nologin`. This is by design and cannot be modified.

Example Snippet

```
users:
  myuser:
    groups:
      - group1
      - group2
    uid: "50"
    homeDir: "/tmp"
```

Sources

You can use the `sources` key to download an archive file from a public URL and unpack it in a target directory on the EC2 instance.

Syntax

```
sources:
  target directory: location of archive file
```

Supported Formats

Supported formats are tar, tar+gzip, tar+bz2, and zip. You can reference external locations such as Amazon Simple Storage Service (Amazon S3) (e.g., `https://s3.amazonaws.com/mybucket/myobject`) as long as the URL is publicly accessible.

Example Snippet

The following example downloads a public .zip file from an Amazon S3 bucket and unpacks it into `/etc/myapp`:

```
sources:
  /etc/myapp: https://s3.amazonaws.com/mybucket/myobject
```

> **Note**
>
> Multiple extractions should not reuse the same target path. Extracting another source to the same target path will replace rather than append to the contents.