

AWS Elastic Beanstalk with a Java Application
Stephanie Dube
CSCI-E90: Cloud Computing
Harvard University Extension School

Final Project: AWS Elastic Beanstalk with a Java Application

Problem: The aim of this project will be to create a dynamic web application (specifically, a simple browser game) using Java and AWS Elastic Beanstalk.

Description: Elastic Beanstalk is an Amazon Web Service for deploying, managing and scaling web applications. It leverages AWS services including EC2, S3, and SNS, and automatically handles the details of load balancing, scaling, capacity provisioning and application monitoring. Elastic Beanstalk uses manages containers that support environments such as .NET, PHP, Python, Ruby, Docker, and Java. This project will focus on using an Elastic Beanstalk environment configured for a Java application running on an Apache Tomcat server.

Benefits: Deploying an application is quick and simple, with minimal infrastructure and resource configuration on the part of the application developer. This allows the developer to focus on the app itself, rather than managing servers, load balancers, database, etc. The option for auto scaling and load balancing enables the app to handle various levels of traffic. Developers are also able to change settings and manage the application environment's infrastructure directly as needed.

Operating System: Windows 10

Software: Java JDK 8, Apache Tomcat 8, Eclipse for Java EE Developers, AWS Eclipse Toolkit

Overview of steps:

1. Install AWS Eclipse Toolkit.
2. Create new AWS Java Web Project, using classes, jsps and web content files, using AWS's "Java Web Project" template as a starting point.
3. Create a WAR file from the Java source code.
4. Create a new Elastic Beanstalk application and environment using a WAR file.
5. Make configuration changes to the environment using the AWS web console.
6. Import the environment into Eclipse and deploy a new application version to the environment.
7. Manage application versions.

Links to YouTube videos:

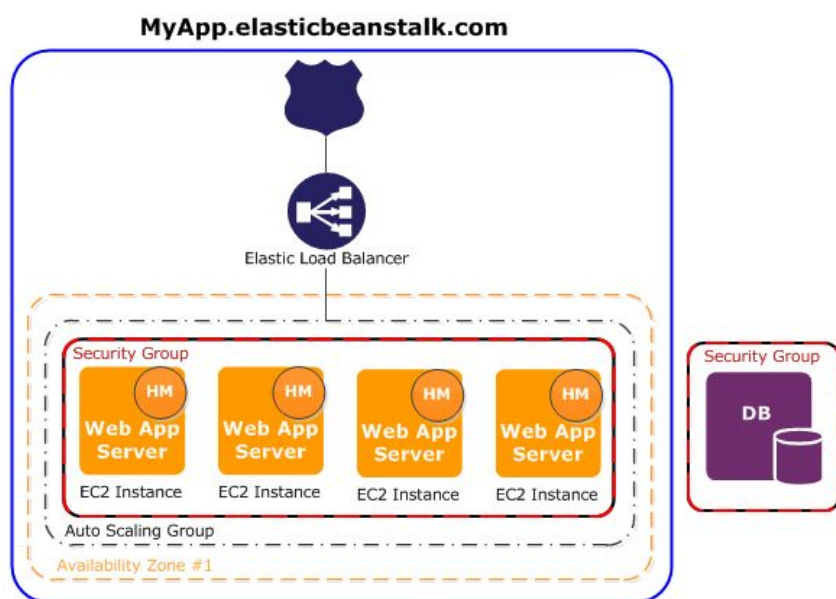
long: https://youtu.be/p4P_XoRFfwQ

short: <https://youtu.be/-awq5G42WO4>

Overview of AWS Elastic Beanstalk

Elastic Beanstalk is an Amazon web service for deploying and managing applications in the AWS cloud. AWS provides numerous tools for launching and controlling servers, storing and managing content, and monitoring resources. Elastic Beanstalk combines several of these services to create one streamlined process, including Elastic Cloud Compute, Simple Storage Service, Simple Notification Service, CloudWatch, RDS, Auto Scaling, and Load Balancing. When a user deploys a new web application with Elastic Beanstalk, many steps for setting up the infrastructure for application health monitoring, load balancing, and auto scaling are handled automatically. Environment configuration changes, such as changing the minimum or maximum number of EC2 instances to be used by the application, can be done through the Elastic Beanstalk console, or with AWS SDK or the Elastic Beanstalk CLI.

When source code is successfully deployed using Elastic Beanstalk, the resulting application running on AWS resources is called an *environment* (AWS, 2015). The environment's behavior is defined through selecting various settings and parameters, called the *environment configuration* (AWS, 2015). The *application version* is the zip or WAR file containing the source code for the application, which is stored in an Amazon S3 object called (AWS, 2015). If you make changes to your application and deploy an updated version to the same environment, the new version will be stored as another S3 object, along with any previous versions. A Beanstalk environment can be configured for and associated with multiple versions of an application, but each environment can run only one version can run at a time. An application version, however, can be run simultaneously on separate environments. Each web server environment has a live URL where the running application version can be viewed.



The left figure from the AWS site shows the architecture of a Web Server Tier environment with Auto Scaling, Load Balancing and an RDS database. The environment, outlined in blue, is made up of multiple EC2 instances, which share a security group with the database. The instances also belong to an auto scaling group which creates or terminates instances of the

application as needed. The application's URL points to the load balancer, which routes traffic to different instances, keeping any one server instance from being overloaded with heavy traffic. This configuring and provisioning of this kind of infrastructure is largely handled automatically when an application is deployed to Elastic Beanstalk.

Creating a Java Web Application

AWS Eclipse Toolkit

AWS Eclipse Toolkit is a plugin for the Eclipse IDE which includes the AWS software development kit API for Java. It supports Elastic Beanstalk development with an interface in Eclipse for creating, managing and deploying code to environments. The AWS SDK also provides sample templates for various types of applications, including dynamic web applications. To install the AWS Toolkit from Eclipse, click "Help" and select "Install New Software." In the window that opens, find the text box labeled "Work with" and type "<http://aws.amazon.com>". This should generate a list including "AWS Toolkit for Eclipse." Select it and step through Eclipse's installation dialogue.

The simplest way to begin a new Java web application is to use a template. Click the AWS Toolkit icon from the Eclipse menu bar, select "New AWS Java Web Project." In the dialogue box that opens, enter a name for the project. Make sure the option "Basic Java Application" is selected and not "Basic Amazon Elastic Beanstalk Worker Tier Application," as this project is going to be a Web Server Tier application, not a Worker Tier application (more on that difference later). Click "Finish."

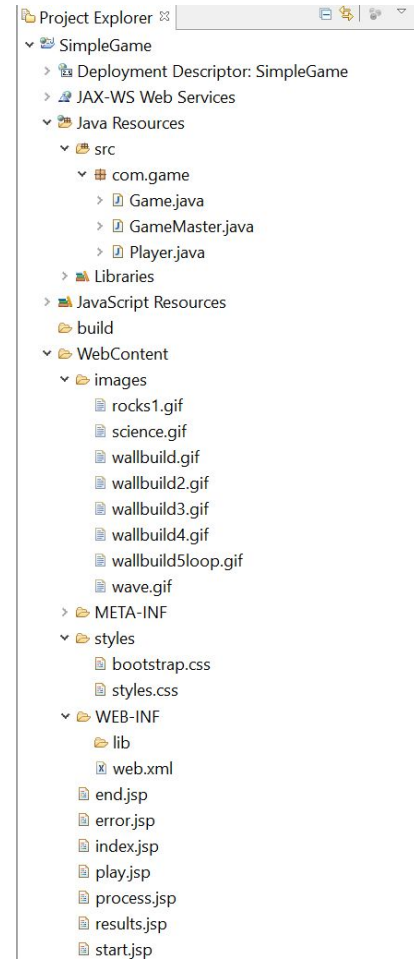
The file structure created for this project is essentially the same as that of the default "Dynamic Web Project" provided by Eclipse, but it includes the AWS SDK library in its Build Path by default, along with a sample HTML page, a file containing your AWS credentials, and a sample JavaServer page (jsp). The jsp demonstrates how to connect to your Amazon S3 buckets, DynamoDB tables, and EC2 instances. If your application does not need to interact directly with any AWS services, you can remove the credentials file from the project. The sample jsp file also explains in a comment that Elastic Beanstalk health checks default to sending HEAD requests to the default root page. HEAD requests can therefore simply return, confirming that the application server is running, without rendering the rest of the application. You can replace these sample files with new, custom jsp and html files as needed for your application.

An Example Web Server Application

My web application is a simple turn-based, single-player browser game. The application consists of a handful of jsp files which interact with server-side classes to dynamically render html and images (animated gifs made for the game) according to game events and client requests. Each EC2 server running the application will have a Singleton Java object called GameMaster which contains a hashmap of Game objects. A new Game object is created when a

user sends a POST request to the server to start the game; the Game is stored in the GameMaster hashmap with the user's session ID as the key. The jsp files are then able to use the session ID to retrieve and update the game state, responding to user input and game object methods. Each Game includes two Players, a user and a "computer," instantiated upon the start of a Game, which contain each player's game data. When a game ends, the game object is removed from the GameMaster hashmap (to be destroyed by Java's garbage collection). The file structure is essentially unchanged from the default template. A package called "com.game" contains "GameMaster.java", "Game.java" and "Player.java" in the "Java Resources" directory.

The game's working title is "Rock Wall Bomb" because of its resemblance to the classic "Rock Paper Scissors" (although it is actually much closer to a similar children's hand-game called "Shotgun"). The rules are, currently, as follows. A player starts a game against a computer. Both start with health at level 3, walls at 0, and bombing power at 0. The player has 4 actions to choose from: "Throw a rock," "Build a wall," "Develop a bomb," or "Offer a truce." If a player throws a rock, their opponent's health is decreased, unless the opponent has a wall, in which case the wall's power is decreased. Bombs do more damage, reducing walls of any strength to 0 and instantly defeating an opponent with insufficiently strong walls. Bomb require multiple turns of development. The game ends when a player's health reaches 0. If both players use a bomb on the same turn and neither has a wall large enough to withstand a bomb, the game is over and both players lose. If both players offer a truce at the same time, the game is over and both players win.



Preparing To Deploy

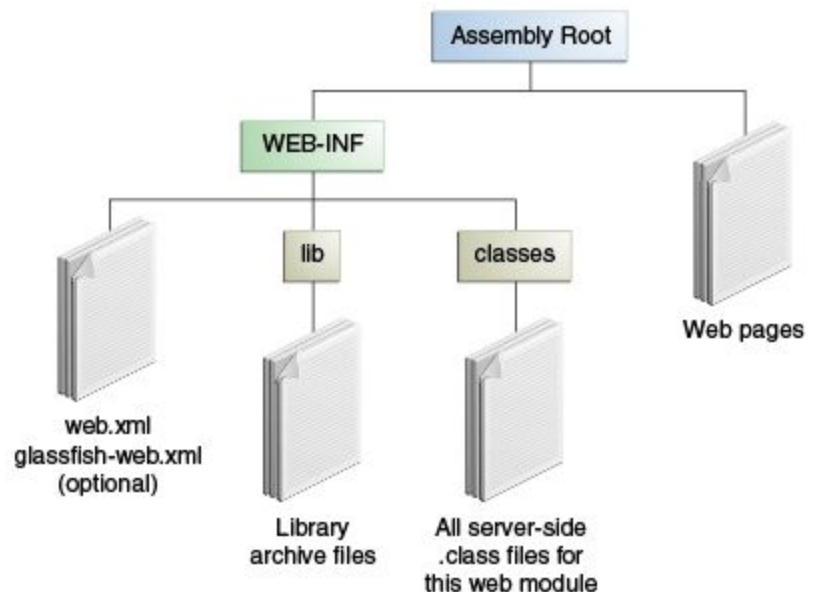
There is more than one way to deploy a web project to Elastic Beanstalk. You can [install EB CLI](#), a command line interface for Elastic Beanstalk, navigate to the project root directory, and deploy the project to a new environment. You can also deploy straight from Eclipse. To do this, select the project root directory in the Project Explorer Window in Eclipse; then click "Run" in the top menu bar and select "Run as server" from the dropdown. In the dialogue box that opens, choose "Manually define a new server" and type "aws" or "Amazon" into the text-box labeled "Select the server type." This should generate a list of servers including "AWS Elastic Beanstalk for Tomcat" (versions 6, 7 and 8). Select the version that matches your JRE run-time

configuration and click “Next.” A screen (shown left) will open to configure a new application and environment. An alternate route is to create a WAR file and upload it to AWS. A WAR file can be easily created using Eclipse (click “Export,” select “WAR” as the format, and specify where to save it) but I will instead walk through the process of creating a WAR file with the command line.

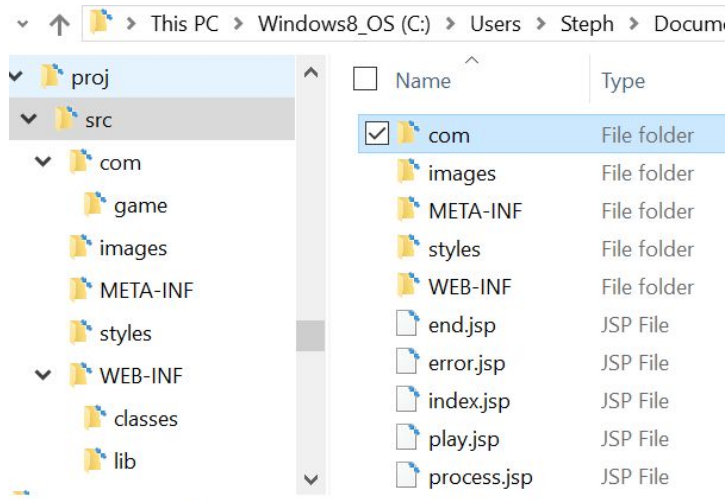
A compiled Java web application, or WAR file, must be structured according to certain standards in order to run on a Tomcat server (Oracle, 2015). The

document root should contain any stylesheets, html pages, client-side classes, and static web resources such as image files. It also contains a directory called WEB-INF, where parts of the application that will not be publically accessible are located, including things like server-side classes, servlets, libraries, and deployment descriptors (Oracle, 2015). The figure to the left, from the Oracle Java EE tutorial site, illustrates the correct hierarchy. If a Java project is deployed to Elastic Beanstalk using AWS Eclipse Toolkit, it is automatically compiled into a correctly structured .war file (AWS, 2015). Still, following these guidelines seems wise, both to follow good practice and to make compiling efficient and simple.

To begin, I copied the project files to a new project directory in Windows Explorer, and altered the directory tree slightly, following the Oracle guideline. The directory “src” is now the assembly root for all of the source files in my project. The package “com/game” contains the Java files which will need to be compiled into class files and stored in WEB_INF/classes (a currently empty directory). The “images” and “styles” directories contain image and css files; these exist in the public



part of the project structure, along with all of the jsp pages. (This newly structured copy of the project is shown in the image to the left.) To create a WAR file from this project, I open a



terminal and navigate to proj/src. I compile the java files, saving them to the WEB-INF/classes directory, with the command “javac -d WEB-INF/classes com/game*.java”. To make sure these classes can reference each other and any necessary library files, I use the command, “javac -classpath WEB-INF/lib/*:WEB-INF/classes -d WEB-INF/classes com/game/*.java”.

```

C:\Users\Steph\Documents\cloud-computing\projpics\proj>cd src

C:\Users\Steph\Documents\cloud-computing\projpics\proj\src>javac -d WEB-INF/classes com/game/*.java

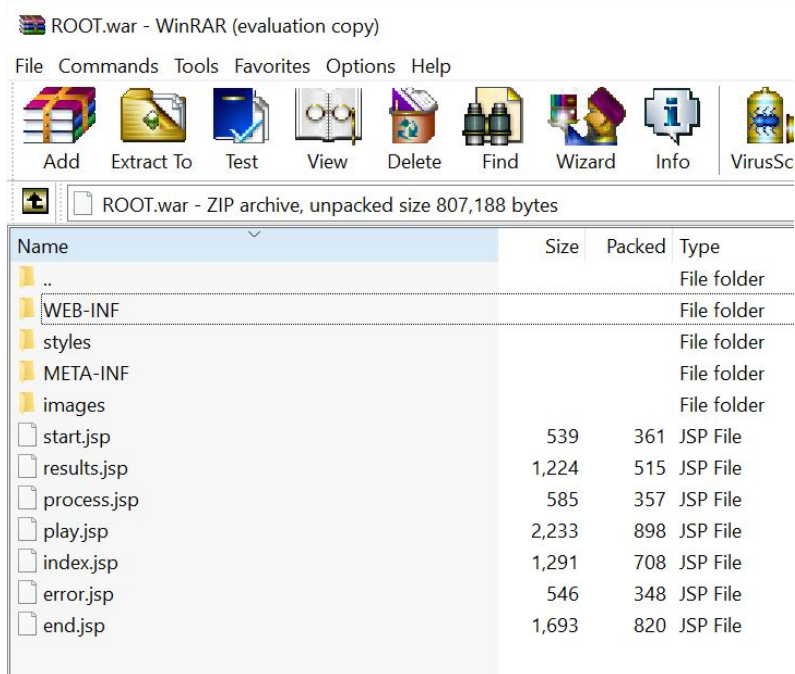
C:\Users\Steph\Documents\cloud-computing\projpics\proj\src>javac -classpath WEB-INF/lib/*:WEB-INF/classes -d WEB-INF/classes com/game/*.java
  
```

Finally, to create the WAR file, I use the command “jar -cvf ROOT.war *.jsp images styles WEB-INF” followed by all the files and directories to be included in the WAR file. These commands can also be made into a build script for more convenient rebuilds. The commands I used are based on an example build script from a tutorial in the AWS Elastic Beanstalk Developer Guide.

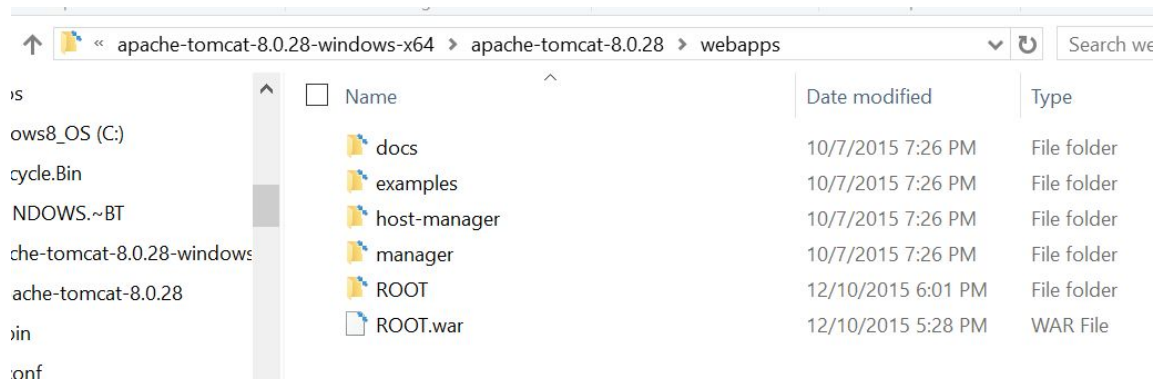
```

C:\Users\Steph\Documents\cloud-computing\projpics\proj\src>jar -cvf ROOT.war *.jsp images styles WEB-INF
added manifest
adding: end.jsp(in = 1693) (out= 820)(deflated 51%)
adding: error.jsp(in = 546) (out= 348)(deflated 36%)
adding: index.jsp(in = 1291) (out= 708)(deflated 45%)
adding: play.jsp(in = 2233) (out= 898)(deflated 59%)
adding: process.jsp(in = 585) (out= 357)(deflated 38%)
adding: results.jsp(in = 1224) (out= 515)(deflated 57%)
adding: start.jsp(in = 539) (out= 361)(deflated 33%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/rocks1.gif(in = 85507) (out= 38206)(deflated 55%)
  
```

The WAR file should now have been created, visible in the root assembly directory for the project. Move it out of the “src” folder, a level up to the project folder, (this is where a “build.sh” file could also be stored). Examine the WAR file to see that it contains all the expected files, in the expected directory tree. If you are using Windows, a simple way to do this by sending the WAR file to WinRAR and opening the WinRAR Archive.



The contents of ROOT.war are correct. META-INF shows the com.game package with its compiled class files. Before deploying it to Elastic Beanstalk, I will test it on my local server.



I place a copy of ROOT.war into Tomcat's webapps directory, and then start the server.

```
C:\apache-tomcat-8.0.28-windows-x64\apache-tomcat-8.0.28\bin>startup.bat
Using CATALINA_BASE:  "C:\apache-tomcat-8.0.28-windows-x64\apache-tomcat-8.0.28"
Using CATALINA_HOME: "C:\apache-tomcat-8.0.28-windows-x64\apache-tomcat-8.0.28"
```

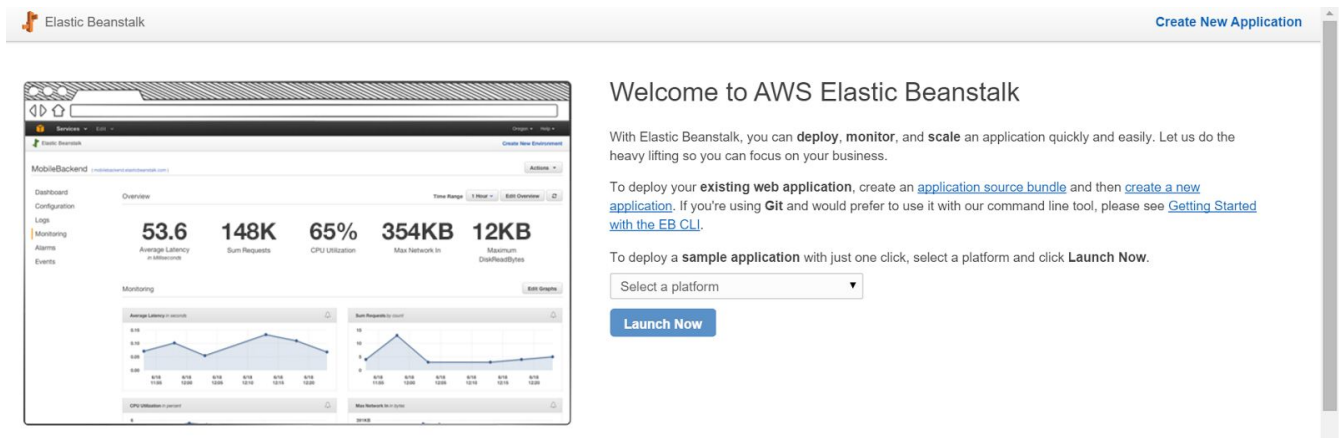
When the server is up, I visit my localhost.



The page displays what a new visitor to the root URL should see (the “index.jsp” page). I started a new game and tried a few different scenarios to verify that everything is in order. The WAR file is now ready to be deployed on a live environment.

Creating a Beanstalk Environment

There are multiple ways to go about creating a new environment, using the AWS Command Line Tool for Elastic Beanstalk, the AWS Eclipse Toolkit, or the Elastic Beanstalk console. For this example, I will describe the steps for creating an environment using the console. To begin with, navigate to aws.amazon.com/elasticbeanstalk/.



From here, there are two options: 1) create an environment using a sample application, or 2) create an environment with a new application. You could launch an environment with a sample application using the platform suited to your own application and then, once the environment is ready, deploy your application to it as a new version. You could also make desired configuration changes to the environment after it has been launched. But deploying to and updating an environment can take several minutes, so I will instead set up the environment with the application and settings I want from the outset. Click “Create New Application” in the upper-right corner to begin the setup process. On the next page, enter a name for the application (I have called mine “simple-game”) and, optionally, a description. Clicking “next” will bring you to the page to select an environment tier.

Selecting Environment Tier, Platform, and Load Balancing

When an environment is launched, Elastic Beanstalk automatically provisions resources for handling certain tasks and behaviors. Different environment tiers require different resource provisioning. An application which processes HTTP(S) requests will use a web server tier environment, while an application which runs background processing jobs, interfacing with SQS and CloudWatch, will use a worker tier environment (AWS, 2015). My project is a web application which responds directly to web requests, so I will click “Create Web Server Tier.”

The next step is to select the platform to use for the environment. Clicking the platform selection dropdown shows a list of several platform configurations which support applications created for different languages and containers. For Java applications, the choice is between different versions of Java with a Tomcat web container, and Java Standard Edition 7 and 8 (without Tomcat). A Java SE platform can be used to run JAR files. If there are multiple JAR files in the root directory, the application must include a build file or a procfile, and can use build tools included with the platform configuration, including javac, Apache Ant, Apache Maven, and Gradle (AWS, 2015). I developed my application using a local installation of a Tomcat server, so I will select a Tomcat platform (the default is Tomcat 8 on a 64-bit Amazon Linux machine).

Environment Type

Choose the platform and type of environment to launch.

Predefined configuration: Tomcat ▼ Looking for a different platform? [Let us know.](#)

AWS Elastic Beanstalk will create an environment running Tomcat 8 Java 8 on 64bit Amazon Linux 2015.09 v2.0.4. [Change platform version.](#)

Environment type: Load balancing, auto scaling ▼ [Learn more](#)

- Load balancing, auto scaling
- Single instance
- Load balancing, auto scaling

[Cancel](#) [Previous](#) [Next](#)

Once you have launched an environment with a certain platform, you will be able to upgrade to new versions of that same platform configuration, but not switch to a different platform type.

This page also offers a choice between a load-balancing or a single-instance application. Load-balancing distributes traffic to the site over multiple instances, instead of just one, and helps prevent the site from being overloaded and slowing or shutting down. For this project, I will select “load balancing” from the dropdown selection here, and click “Next.”

Uploading an Application Source Bundle

The next step in the process of creating a Beanstalk environment is to specify the source of the application version to be run. Again, there is an option to choose a sample application that can run in the type of environment specified; or, if an application source bundle has previously been uploaded to an S3 bucket, that application can be used here by providing the S3 URL. I will demonstrate the third option and upload a new application source bundle here. The bundled application must be a ZIP or WAR of 512 MB or smaller, with no top-level parent folder (AWS, 2015). I chose the ROOT.war file created in a previous step.

Application Version

Select a source for your application version.

Source: ☐ Sample application

☒ Upload your own ([Learn more](#))

ROOT.war

☐ S3 URL

(e.g. <https://s3.amazonaws.com/s3Bucket/s3Key>)

Deployment Limits

Elastic Beanstalk will update your application in batches so as to avoid downtime when deploying. [Learn more](#)

Batch size: ☒ Percentage

% of the fleet at a time

☐ Fixed

instances at a time

Further down on this same page, there are setting options for handling the deployment of new application versions. By default, rolling deployment is enabled. This means that Elastic Beanstalk handles application updates in batches. One batch of instances will be disconnected from the load balancer while the application is set up on each of them, which will usually involve restarting the server (AWS, 2015). If health checks are configured for the environment, Beanstalk will wait until an instance URL returns 200 before reattaching it to the load balancer (AWS, 2015). You can define how many instances will be taken offline and updated at a time here. This rolling deployment can also be disabled altogether once the environment is running, in which case deploying a new version will actually result in the creation of a new environment, which then swaps its CNAME or environment URL with the original environment (AWS, 2015).

After choosing settings for application version deployment and uploading the application source bundle, you will come to the “Environment Information” page. Enter a name for your environment. This will be the URL for your application. The default format is “yourApplication-env”. I have named my environment “simpleGame-env”.


Additional Resources



Next, there is the option to include additional resources, including a Relational Database Service, and a Virtual Private Cloud. An RDS database can be added to the environment later as well, after the environment has been launched. A Java application can use MySQL Connector to

make interact with the database, using either the hard-coded database endpoint (to an Amazon RDS database or to some outside SQL database), or, if the RDS database is associated with the environment, using RDS information read dynamically from the environment system (AWS, 2015). At this time, my application does not require a database, so I will leave this box unchecked. At this stage, you can also create a VPC here, which would let you define a private, virtual network, which could be used for web service backends, for example. Again, it is possible to add this feature to the environment at a later stage if it is not selected here. For my application, I will leave this box unchecked, and proceed to the next page.

Environment Configuration


On the next page, you can configure some launch settings for the environment. Select the type of instances that you to run your application and supply an EC2 Key-Pair to enable logging into them remotely. Next, provide a URL for health checks to be made to; if no URL is supplied and health checks are enabled, the health check involves a TCP request on port 80 to the instance (AWS, 2015). For my application, I will use “/index.jsp” which is the welcome page for my web application.

Instance type: 
Determines the processing power of the servers in your environment.

EC2 key pair:  [Refresh](#) 
Optional: Enables remote login to your instances.

Email address:
Optional: Get notified about any major changes to your environment.

Application health check URL:
Enter the relative URL that ELB continually monitors to ensure your application is available.

Enable rolling updates: ☒ Lets you control how changes to the environment's instances are propagated. [Learn more.](#)
 
Specifies whether to wait to deploy updates and deployments according to a set period of time or instance health.

At this stage, you can also provide an email address to receive information about changes to the environment, and select the type of health reporting, “Basic” or “Enhanced.” Enhanced reporting uses additional resources to provide a more detailed analysis of the environment’s health. The health reporting level can be switched later on, when the environment is running. You will also be able to further specify settings for application health monitoring (such as the number of health checks required to declare an instance healthy and the interval of time between checks) after the environment has been launched.

When configuration settings are changed after launch, the environment must be updated; for some changes, instances do not need to be changed. However, changes to the launch configuration (instance type, security groups, Key-Pair, etc.) will result in instances being terminated and replaced (AWS, 2015). As with deploying new versions, this can be done in

batches to prevent downtime. The details of this can be similarly specified from the environment dashboard after it is By default, rolling updates progress based on the health of updated instances; if updated batches fail health checks, the update will be not progress and the updated batch will return to its previous configuration (AWS, 2015). Check the box for “Enable rolling updates” to use this feature.

Load Balancer Settings

By default, the load balancer will route visitors to instances within their own availability zone (AWS, 2015). If cross-zone load balancing is enabled, traffic is spread evenly throughout all existing instances, regardless of their location. For my application, which will depend on session stickiness to route users to the same instance during one session (game data is stored in the application instance), I will disable cross-zone load balancing to narrow the range of instances one user might be routed to. Enabling connection draining means that instances that are failing health checks or are being deregistered due to auto scaling, remain connected to the load balancer for some amount of time which can be specified, if there is an in-progress request to that instance URL. The load balancer will not send any new requests to this URL during this “draining” period. The default, without connection draining enabled, is for such instances to be disconnected immediately. For my application, I will leave this box checked to enable connection draining, and accept the default time of 20 seconds.

Roles and Permissions

The remaining configuration steps include choosing a volume type (I’ll choose the default), adding environment tags (I won’t be using any), and providing an AWS Identity and Access Management, or IAM, role. Elastic Beanstalk needs to be assigned a service IAM role so that it can make API calls to other services, like Load Balancing, Auto Scaling, and EC2, to get information about the health of the resources being used by those services for the environment. Elastic Beanstalk also requires an instance profile so that environment instances can be given permission to use other services to do things like write logs to S3. For worker environments, the Instance Profile is even more important, as worker applications’ primarily use is to perform backend tasks with AWS services. When creating an IAM role via the console, as opposed to the CLI or SDK, an instance profile containing that role is also created automatically (AWS, 2015). If you have not previously created an instance profile or service role, you will have to create them at this stage.

Instance profile:

Service role: [Select this option to create a preconfigured role in the IAM console \(opens a new window\).](#)

[Cancel](#) [Previous](#) [Next](#)

Click “Next,” and in the new window that opens, click “Allow” to create the new IAM roles.

Role Summary ?

Role Description AWS Elastic Beanstalk needs permission to use resources in your account.

IAM Role Create a new IAM Role ▼

Role Name aws-elasticbeanstalk-service-role

► View Policy Document

Once the service role and instance profile are created, you will be returned to the environment setup page. At this point, the application and environment are ready to be created. Review the selected settings and click “Launch.”

Managing the Application Environment

Elastic Beanstalk will take several minutes to launch the new environment. The Elastic Beanstalk home lists all of the application environments for your account in the selected region as boxes color-coded according to their health status. While it is launching, the environment is shown as gray. Once it becomes functional, it will turn green. If a major problem occurs, it will turn red. The tabs at the top of the page let you switch between the EB home and the dashboard for a specific application environment.

All Applications

simple-game

Clicking “Application Versions” from the dropdown menu for the application will bring you to a table showing all of the versions of the applications. Any version can be deleted, deployed to the current environment, or used to launch a new environment. New versions can also be uploaded here, as ZIP or WAR files, and existing versions can be downloaded.

Version Label	Description	Date Created	Source	Deployed To
First Release		2015-12-12 02:21:17 UTC-0800	2015346wIT-ROOT.war	simpleGame-env

Clicking the name of the environment (“simpleGame-env”) from the dropdown, or clicking the box for the environment on EB home, will take you to the environment dashboard, which displays the environment’s health status, currently running application version, platform, a list of recent environment events, and a link to the load balancer URL. When the environment has completed its launch process, you can visit the live site by clicking that link.



Environment Dashboard

The environment dashboard includes a menu to sections for viewing and managing different configurations and metrics, including Configurations, Logs, Monitoring, Alarms, Events, and Tags. The Configurations page groups the environment’s settings into sub-sections.

Scaling ⚙️ Environment type: Load balanced, auto scaling Number instances: 1 - 4 Scale based on: Average network out Add instance when: > 6000000 Remove instance when: < 2000000	Instances ⚙️ Instance type: t1.micro Availability Zones: Any Key pair: HW2KeyPair	Notifications ⚙️ Notifications: On Send notifications to: Blank
Software Configuration ⚙️ Log publication: Off Initial JVM heap size: 256m JVM command line options: Blank Maximum JVM heap size: 256m Maximum JVM permanent generation size: 64m	Updates and Deployments ⚙️ Application deployment batch size: 30% Rolling updates: are enabled Rolling update type: Health Max batch size: 1 Minimum instances in service: 1	Health ⚙️ Application health check URL: /index.js Health reporting: Basic

Clicking the gear icon on any group box opens a more detailed configuration page where you can make changes to the settings. For example, my environment initially was failing health checks. When I navigated to this page, I noticed that the application health check URL was set to “/index.js” when it should have been “/index.jsp”. In Health configuration, I was able to edit this and apply the change. Once the environment finished updating, it was reported healthy.

[simple-game](#) ▶ [simpleGame-env](#) (Environment ID: e-pntzkspk34, URL: [simplegame-env.elasticbeanstalk.com](#))

Dashboard	Application Health Check	
Configuration	The following setting lets you configure how Elastic Beanstalk determines whether your application is healthy.	
Logs	Application health check URL:	<input type="text" value="/index.jsp"/> Path to which ELB sends an HTTP GET request

Another change I wanted to make was to enable session stickiness. My application uses data stored in Java classes on an instance-level scope, so it would be ideal for users to be routed consistently to the same instance. To do this, I clicked the gear icon on the Load Balancer box (located further down on the main Configurations page) and found the relevant section.

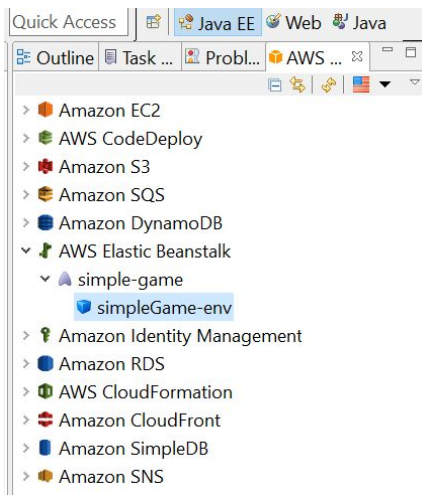
Sessions

The following settings let you control whether the load balancer routes requests for the same session to the Amazon EC2 instance with the smallest load or consistently to the same instance.

Session stickiness:	<input checked="" type="checkbox"/> Enable session stickiness.
Cookie expiration period (seconds):	<input type="text" value="12000"/> Maximum amount of time that a session cookie between an Amazon EC2 instance and the load balancer is valid.

After making my changes, I scrolled down and clicked “Apply.” When you make a configuration change, the environment has to be updated, which take a minute or two to complete. Only one change can be made at a time.

The other sections of the dashboard provide various kinds of information about the environment’s status. On the logs page, you can request a set of logs for every server in your environment, or a log of the more recent 100 lines taken from all of the server logs combined. The health page, if enhanced health is enabled, shows details for every server, including average number of requests per second, latency, load average, and CPU utilization. Monitoring shows similar information but for the environment as a whole, rather than for each server. You can also create alarms for any monitored metrics on this page; any alarms you create will be displayed on the alarms page. Lastly, the events page shows the complete list (of which only the most recent handful appear on the main dashboard) of environment events, such as environment state transitions (e.g. “RED to “GREEN”), the start of a new deployment or update, the successful completion or failure of a deployment or update, or other events.

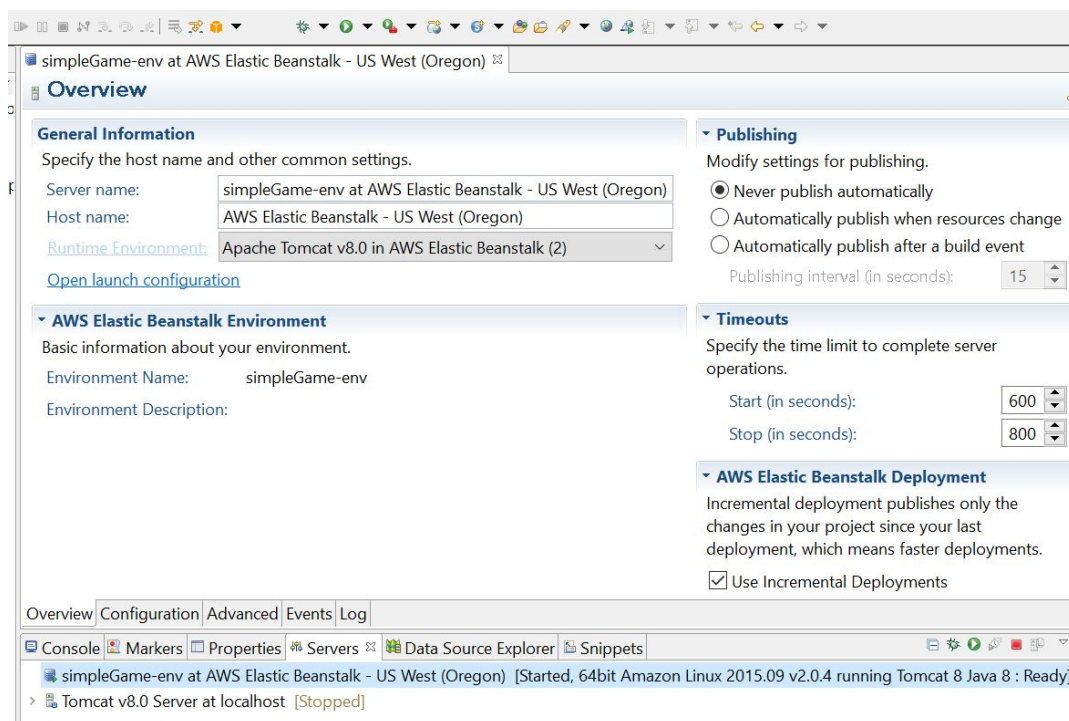


Deploying Via AWS Eclipse Toolkit

Once an environment has been launched, it can be imported into an Eclipse workspace. In Eclipse, view the AWS explorer window. Open the AWS Elastic Beanstalk listing; any environments currently running should be listed there. Double click the environment to open a window for managing it.

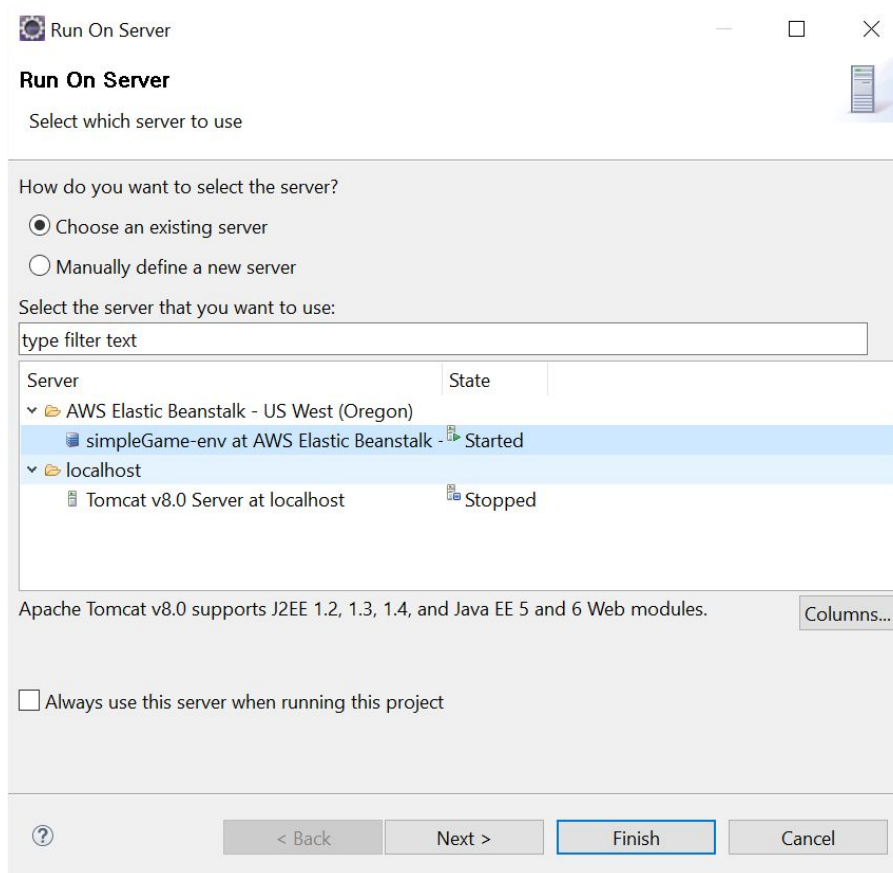
In the window that opens, you can choose some settings to define how publishing and deploying to this environment from Eclipse is handled. You can also manage and view environment configurations here, in addition to in the AWS web console, by navigating to the “Configuration,”

“Advanced,” “Events,” and “Log” tabs in this window.



By opening this window, the environment has been imported into your Eclipse workspace. You can easily deploy applications created in Eclipse directly to the environment server, as it should be automatically added to your list of servers (as shown in the above screenshot). Since bundling my application as a WAR, I have made some changes to my application. To deploy the updated version to my environment, I right-click the project and select “Run as server.” In the dialogue box that appears, with the option to “Choose from existing”

selected, the server for my environment should be listed: “simpleGame-env at AWS Elastic Beanstalk.” I can select this server and click “Finish.”



When Eclipse has finished pushing the new version of the application to Elastic Beanstalk, navigate to the Application Versions section in the Elastic Beanstalk console. Clicking “Application Versions” should show the new version, as well as any previous version.

Delete Deploy Upload Refresh					
<input type="checkbox"/>	Version Label	Description	Date Created	Source	Deployed To
<input type="checkbox"/>	git-438c92d	Incremental Deployment: Sat Dec 12 02:43:19 PST 2015	2015-12-12 02:50:07 UTC-0800	git-438c92d.zip	simpleGame-env
<input type="checkbox"/>	First Release		2015-12-12 02:21:17 UTC-0800	2015346wIT-ROOT.war	

The environment is now running the new version deployed from Eclipse. Any previous version of the application that has been uploaded to AWS can be redeployed to the environment at any time. Note that only one environment can be run on a given environment at a time, however. If you want to run multiple versions simultaneously, an application version can be launched on a new environment. To delete a version, select it and click “Delete.” If you delete the version which is currently running on the environment, the environment will go offline, and all instances running the application will be terminated.

Conclusion

There are many advantages to using Elastic Beanstalk to host a web application. The ability to dynamically scale the number of servers hosting an application in response to traffic means that a web application will not outgrow the service even if it sees a major increase in popularity, and if an application has very variable traffic loads, for example, an online store might spike abruptly during holiday shopping periods, the auto scaling and load balancing provided by Elastic Beanstalk will keep the application's performance from suffering. Another benefit is the option to roll out new versions or configurations in batches on an environment with multiple servers. For some web applications, having any downtime when the site becomes completely unavailable would be a major problem; with Elastic Beanstalk, this can be avoided.

A possible drawback to using Elastic Beanstalk may be that the webmaster has less tight control over the provisioning and infrastructure of an application than if they were to configure all of the application's resources from scratch, programmatically. On the other hand, the way that Elastic Beanstalk automatically manages resources for the application can clearly be seen as an advantage in many situations where such tight control is not needed, and Elastic Beanstalk does allow for a large degree of control. The EB console conveniently displays all of the application's resources and services in one place and allows the developer to manage configurations for all of these, with the web console interface, the EB CLI, or even programmatically through the SDK API, although that was not covered here. The way that Elastic Beanstalk bundles together a set of optional and commonly used services is very convenient and makes the process of getting a web application up and running very efficient. There are services that are not included in Elastic Beanstalk, which a developer might like to use; this could be seen as another drawback. However, other services can be used in conjunction with EB, although integrating them with an EB application would require extra steps.

Another benefit is that with EB, the abstraction of an application environment exists. While you could create an application that runs exactly like an EB application just by combining different AWS services, the final product, the combination of these tools and resources, does not have one clear "home." With Elastic Beanstalk, the product which emerges from combined resources is treated as an object itself; its health is monitored and its resources and information gathered and displayed in one place. The way that the source code is stored and displayed as deployable versions is very convenient as well.

Citations and References

Amazon Web Services. “AWS Elastic Beanstalk Developer Guide.” (API Version 2010-12-01).

<https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html>.

JSP Tutorial. Retrieved from <http://www.jsptut.com/>.

Oracle. The Java Tutorials. “Packaging.” “Scope.” “Getting Started with Web Applications.” <https://docs.oracle.com/javase/7/tutorial/>.