Table of Contents

A Little Background 4

Technology Stack 4

Deploy the App 5

What’s Going On? 5

Configure Credentials in Eclipse 6

Get Access Key and Secret Key from QwikLab 6

Configure the AWS Toolkit for Eclipse 6

Provide AWS Credentials as JVM Args 7

Achievement Unlocked! 9

Download Source and Create Project 10

Import Application to Eclipse 10

Configure Project in Eclipse 11

Import EB Environment Into Eclipse 15

Open the Elastic Beanstalk Console 18

OK, Now What? 20

Import App Config for Local Dev 21

Modify and Deploy Application 23

Roll Back to Initial Version 27

Project Anatomy 28

Example of a Challenge 28

Package Layout 29

Templates and Static Assets 29

Challenge: Get App Config From S3 31

How Did Config Get to S3, Anyways? 31

The Challenge 31

Super Powers 31

Resources 31

If You Get Stuck 31

Detail Detour 31

Challenge: DynamoDB and Users 33

About The DynamoDB Users Table 33

The Challenge 33

Super Powers 33

Resources 33

If You Get Stuck 33

Challenge: RDS, Read Replicas, and Connection Strings 34

The Challenge 34

Super Powers 34

Resources 34

If You Get Stuck 34

Challenge: S3 for Profile Pictures 35

The Challenge 35

Important Considerations 35

Super Powers 35

Resources 35

If You Get Stuck 35

Challenge: Elastic Transcoder Service 36

The Challenge 36

Important Considerations 36

Super Powers 36

Resources 36

If You Get Stuck 37

Challenge: Custom CloudWatch Metrics 38

Basic cw api stuff 38

Coding a Java Web Application

A Little Background

# A Little Background

Our sample application is a scalable Java web app that allows users to upload, convert, and share videos via a web browser. The app uses several AWS services that allow it to grow to very large scale: Amazon S3 for uploaded videos and app logs; Amazon RDS for storing searchable video metadata; Amazon DynamoDB for storing user profile information; and Amazon ElastiCache to cache database query results and improve application performance. We develop the application locally in Eclipse and deploy it to AWS Elastic Beanstalk, which gives us Tomcat 7 instances that scale automatically behind an Elastic Load Balancer.

# Technology Stack

The application utilizes the following technologies:

* Spring Framework 3.2
* Spring Security 3.1
* Spring Web MVC
* ThymeLeaf Template Engine
* AWS SDK for Java
* Apache commons-fileupload, commons-io, common-dbcp, and commons-lang

Deploy the App

# Deploy the App

We’re going to start by launching a fully functional and implemented version of our application. Throughout the workshop we’ll visit individual components of the application and implement them ourselves.

Open QwikLab in your browser, create an account, sign in, and click ‘Start Lab’. Behind the scenes, QwikLab is using AWS CloudFormation to create your Elastic Beanstalk application ad supporting components (DynamodDB tables, RDS databases, etc). We’ll talk about this is a group in more detail.

## What’s Going On?

CloudFormation is creating an RDS database, DynamoDB table, S3 bucket, and ElastiCache cluster. Once those things are created, CloudFormation will deploy a functional version of the sample application to Elastic Beanstalk and will pass in the IDs of the database, table, bucket, and cluster it created previously.

Configure Credentials In Eclipse

# Configure Credentials in Eclipse

Before we import the source for our sample application and begin coding, let’s configure our IDE with the AWS API credentials (i.e., Access Key and Secret Key) we’ll need to interact with various AWS services.

## Get Access Key and Secret Key from QwikLab

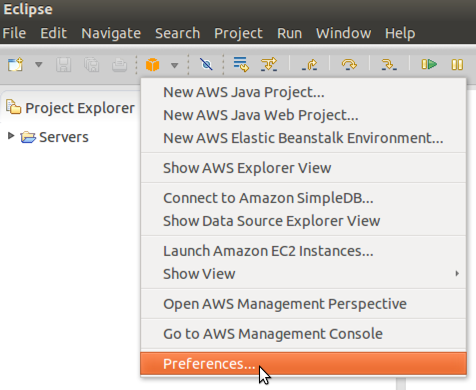
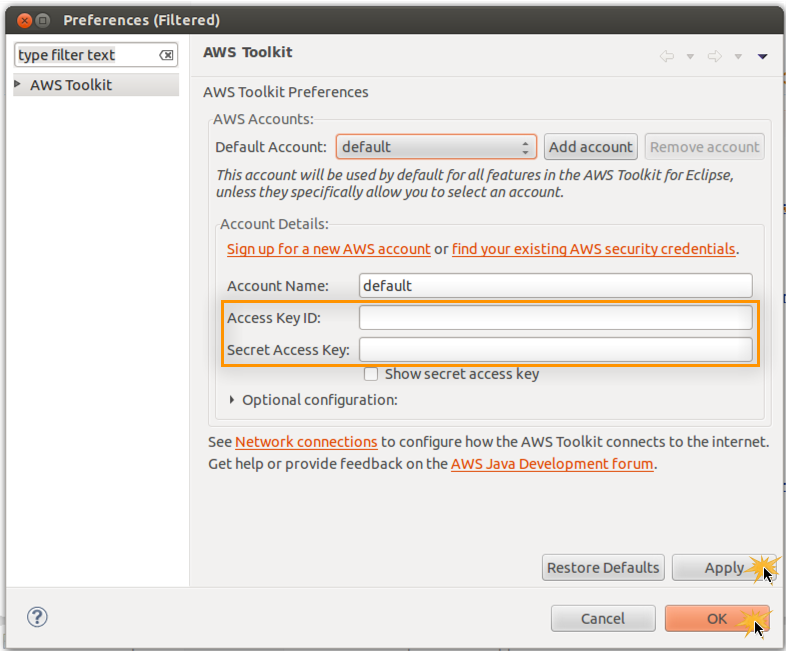
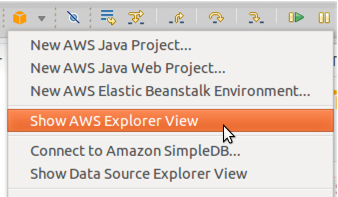
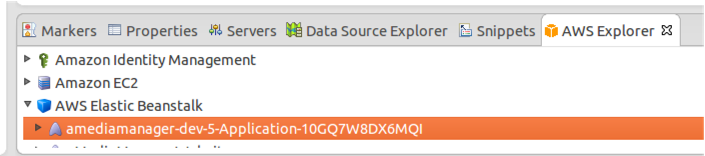
An Access Key (AK) and Secret Key (SK) are used to sign API requests that you make to AWS. QwikLab has helped us out by automatically generating an AK and SK for you. Open the QwikLab tab in your browser and locate them:

**TODO: Provide screenshots of QwikLab UI for getting keys.**

If we weren’t using QwikLab in this workshop, you would manage your keys by using the Identity and Access Management service in the AWS Management Console at <https://console.aws.amazon.com/iam>.

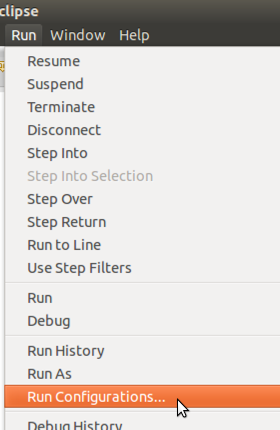
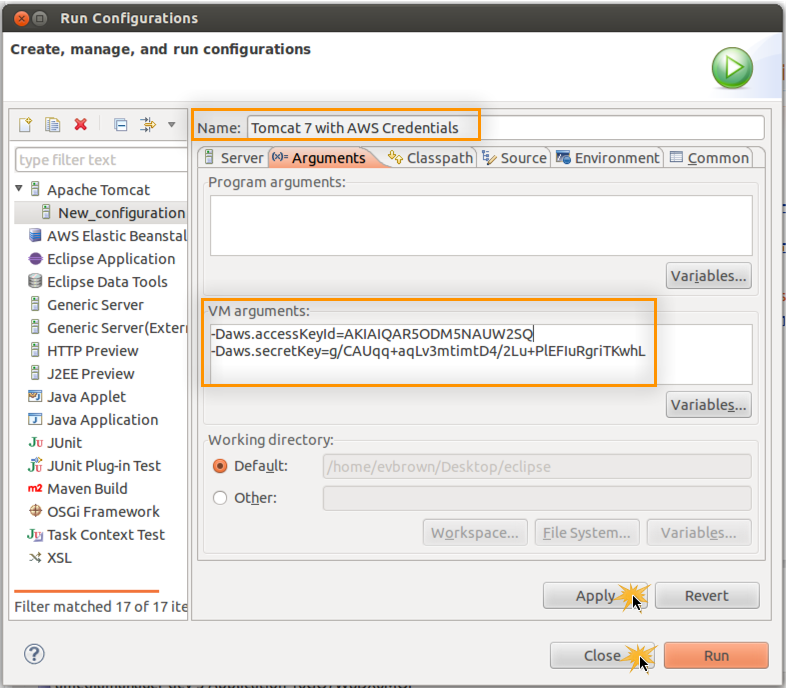
## Configure the AWS Toolkit for Eclipse

The toolkit will help you deploy your application to AWS Elastic Beanstalk and needs an Access Key and Secret Key configured to be able to call the EB APIs.

1. Select the Preferences option in the AWS Toolkit dropdown menu:  
   
2. Enter the Access Key ID and Secret Access Key you retrieved from QwikLab, then click Apply and OK:  
   
3. Confirm that your credentials are correctly configured by opening the AWS Explorer view and listing your AWS Elastic Beanstalk Environments:  
     
     
   

## 

right?

1. **
2. *VM*

## Achievement Unlocked!

Good work. You’ve configured Eclipse to allow the AWS Toolkit to work, and your local Tomcat 7 installation will make your AK and SK available to any apps running on it via System.getProperty() calls (and we don’t have to worry about hard-coding those credentials in code or properties files).

In the next section, you’ll import the application source code into a new project in Eclipse.

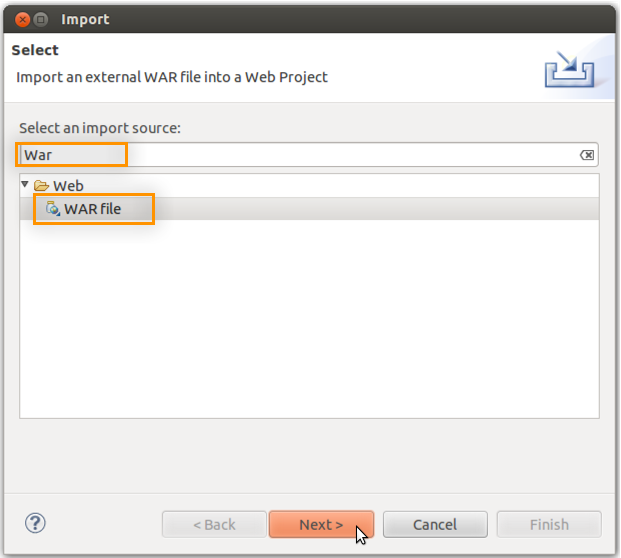
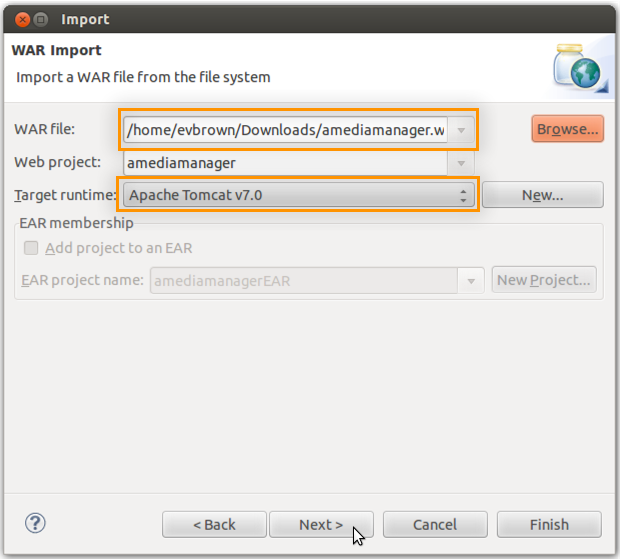
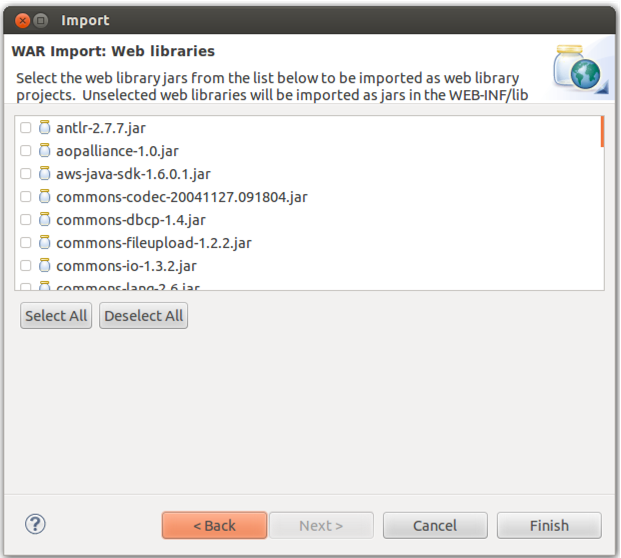
Download Source and Create Project

# Download Source and Create Project

Now that you’ve configured your local IDE, import the application into Eclipse and deploy the first version of the code to Elastic Beanstalk.

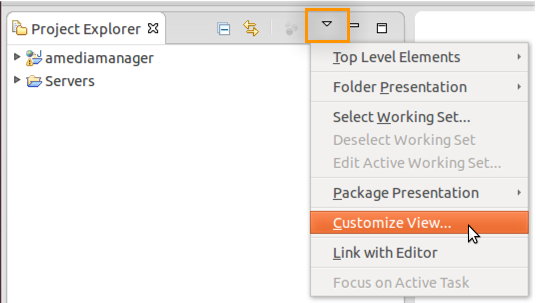
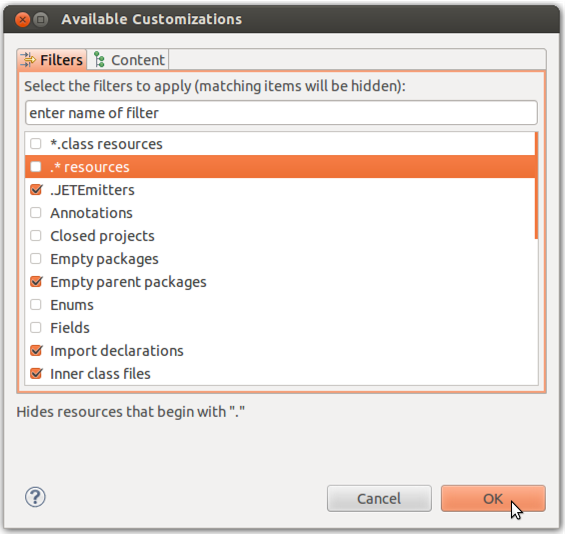
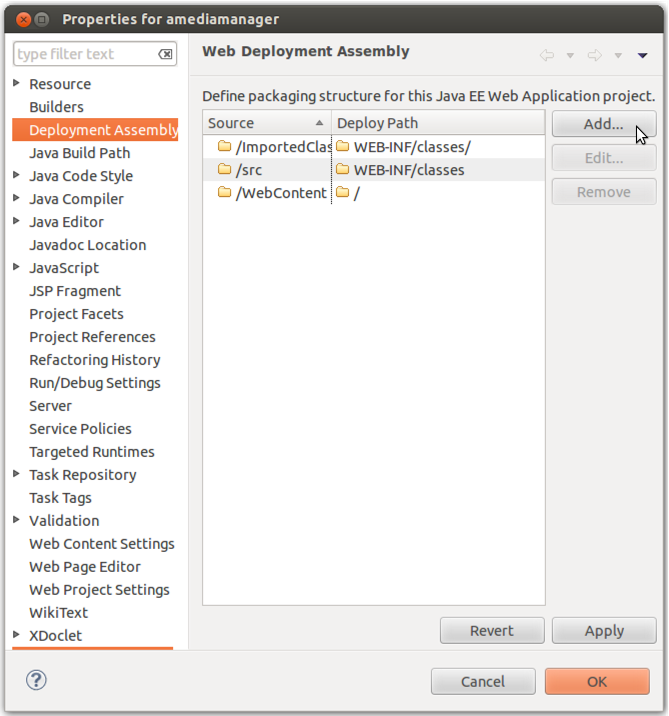
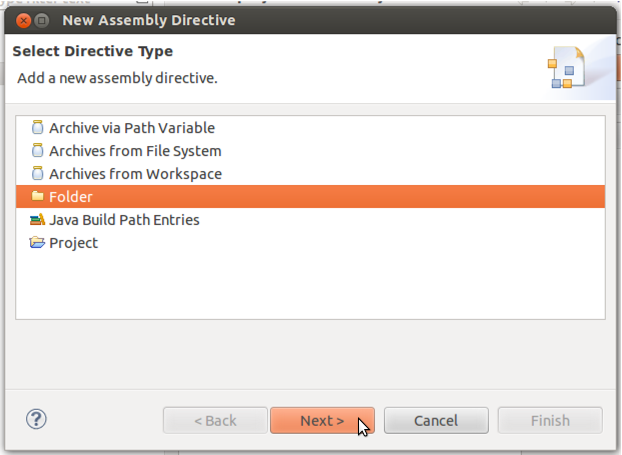
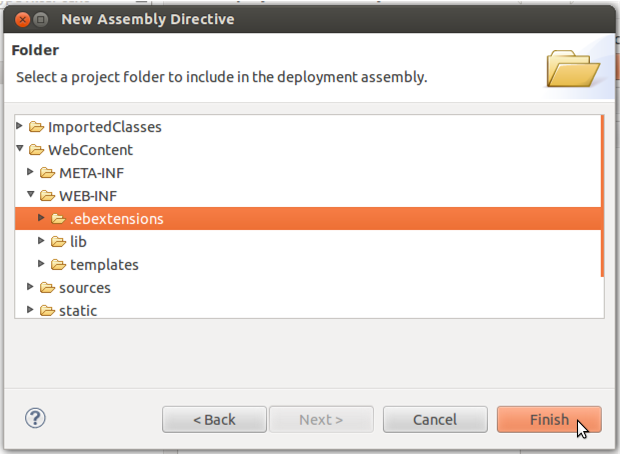
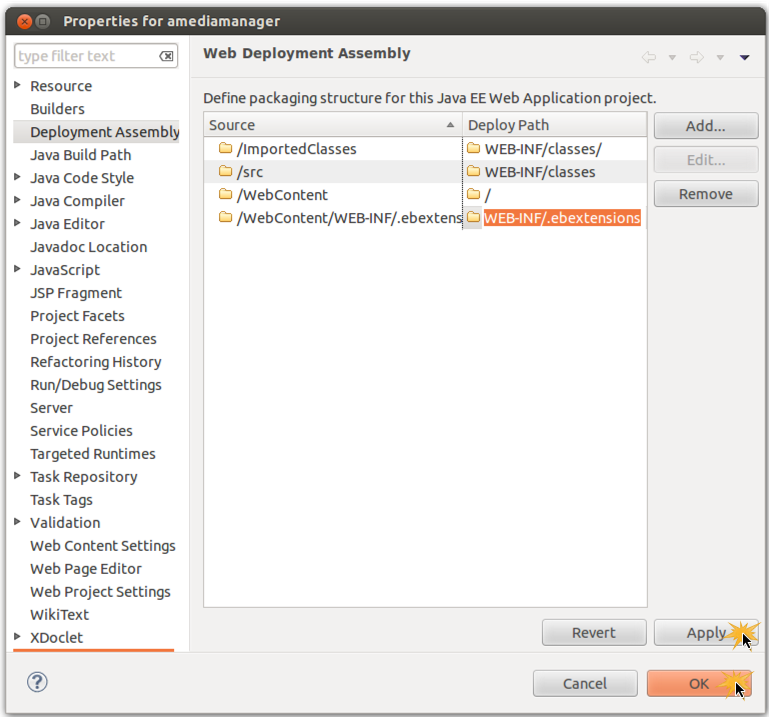
**Developer’s Note:** The source is distributed as a WAR file for this workshop and will be made available on GitHub as a Maven project afterwards.

## Import Application to Eclipse

1. Download the application from <http://amm-us-west-2.s3.amazonaws.com/public/amediamanager.war>
2. In Eclipse, choose File > Import
3. Type *WAR* in the filer box, choose WAR file, and click Next  
   
4. Choose the WAR file you downloaded previously, ensure that Apache Tomcat 7.0 is is selected for *Target runtime*, and click Next:  
   
5. Do not select any WAR files to import. Click Finish:  
   

## Configure Project in Eclipse

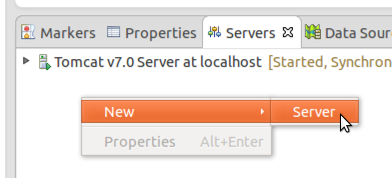
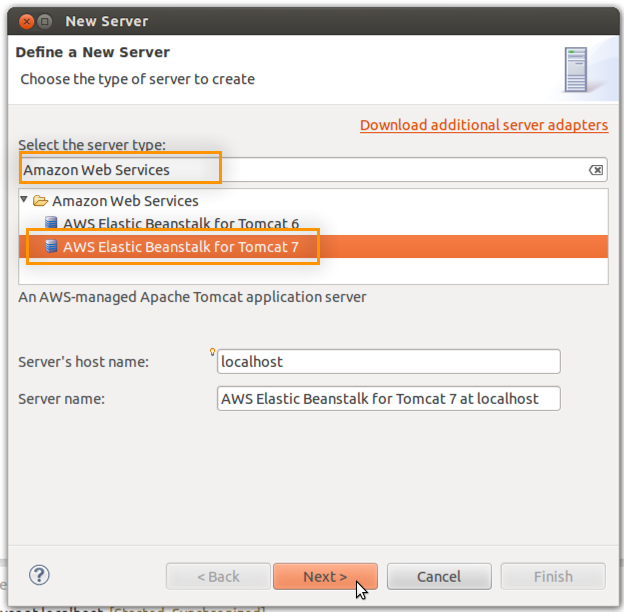
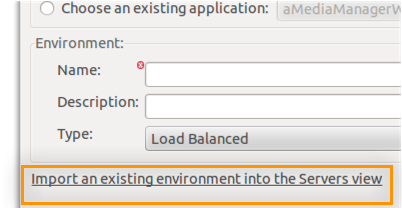
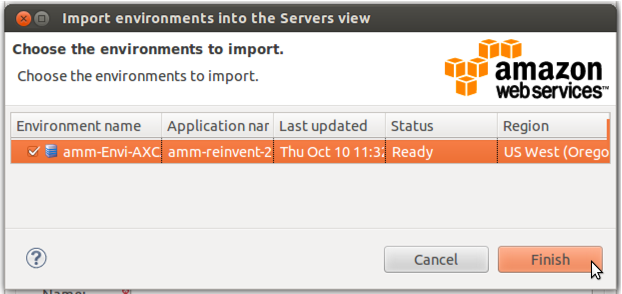
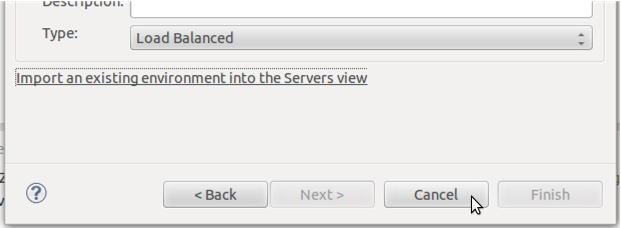
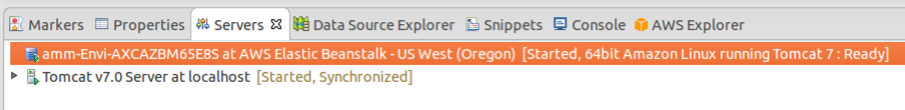
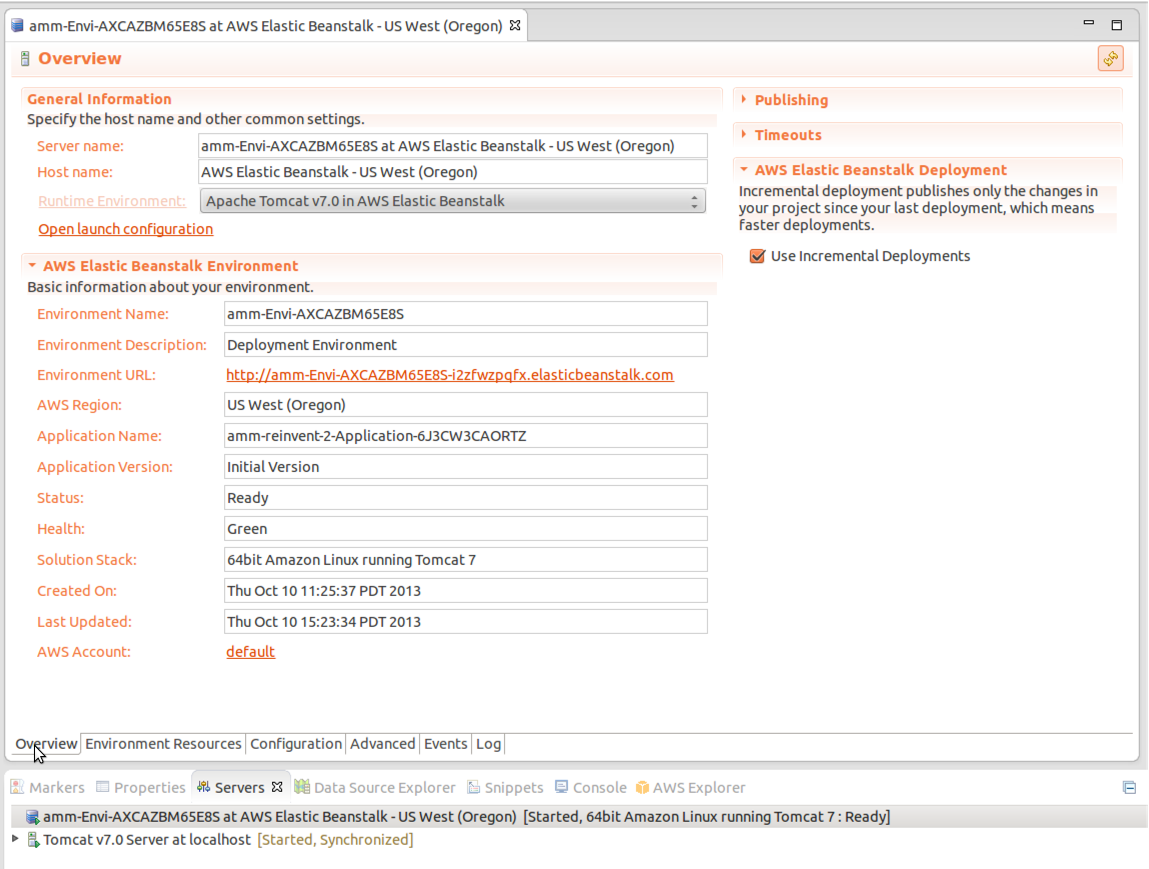
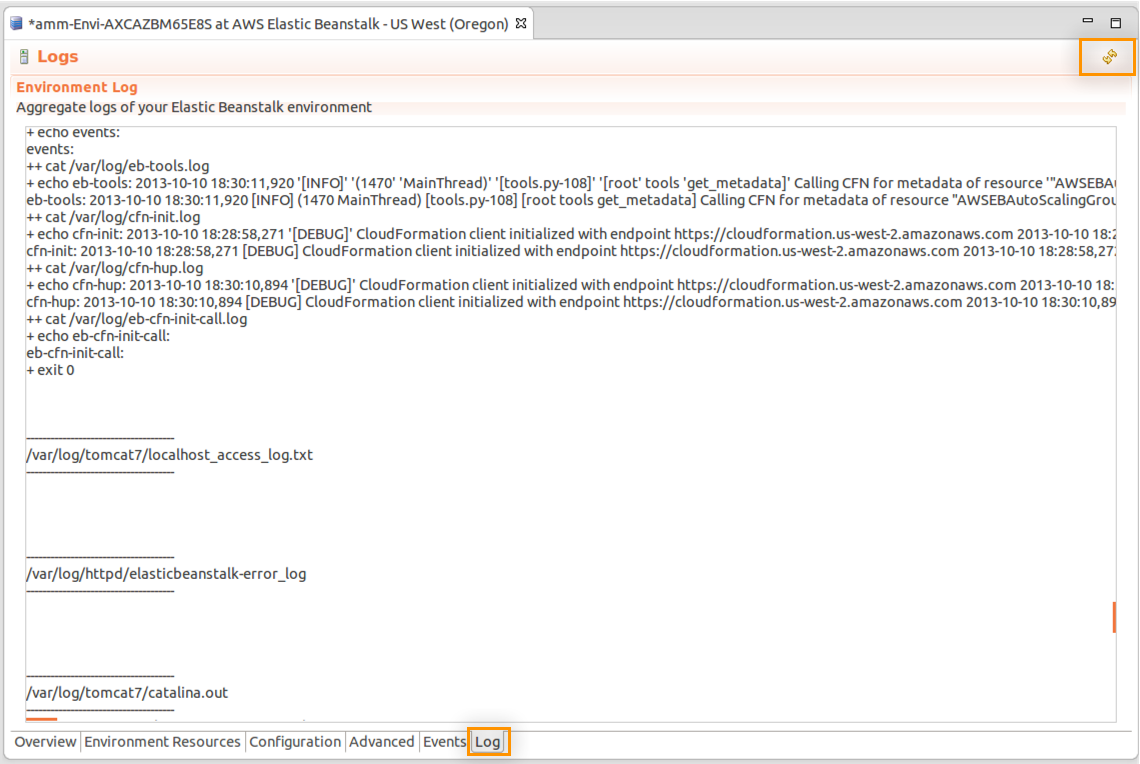
Configure Eclipse to show files and folders that begin with a ‘.’ and copy the *.ebextensions* folder to the WAR file it builds and deploys.

1. Click the down arrow in Project Explorer and choose Customize View:  
   
2. Uncheck *.\* resources* and click OK:  
   
3. Right-click the amediamanger project and choose Properties
4. Select Deployment Assembly and click Add:  
   
5. Select Folder and click Next:  
   
6. Select the WebContent > WEB-INF > .ebextensions directory and click Finish:  
   
7. Double-click the Deploy Path field for the .ebextensions Source, enter *WEB-INF/.ebextensions*, click Apply then OK:  
   

Import EB Environment Into Eclipse

# Import EB Environment Into Eclipse

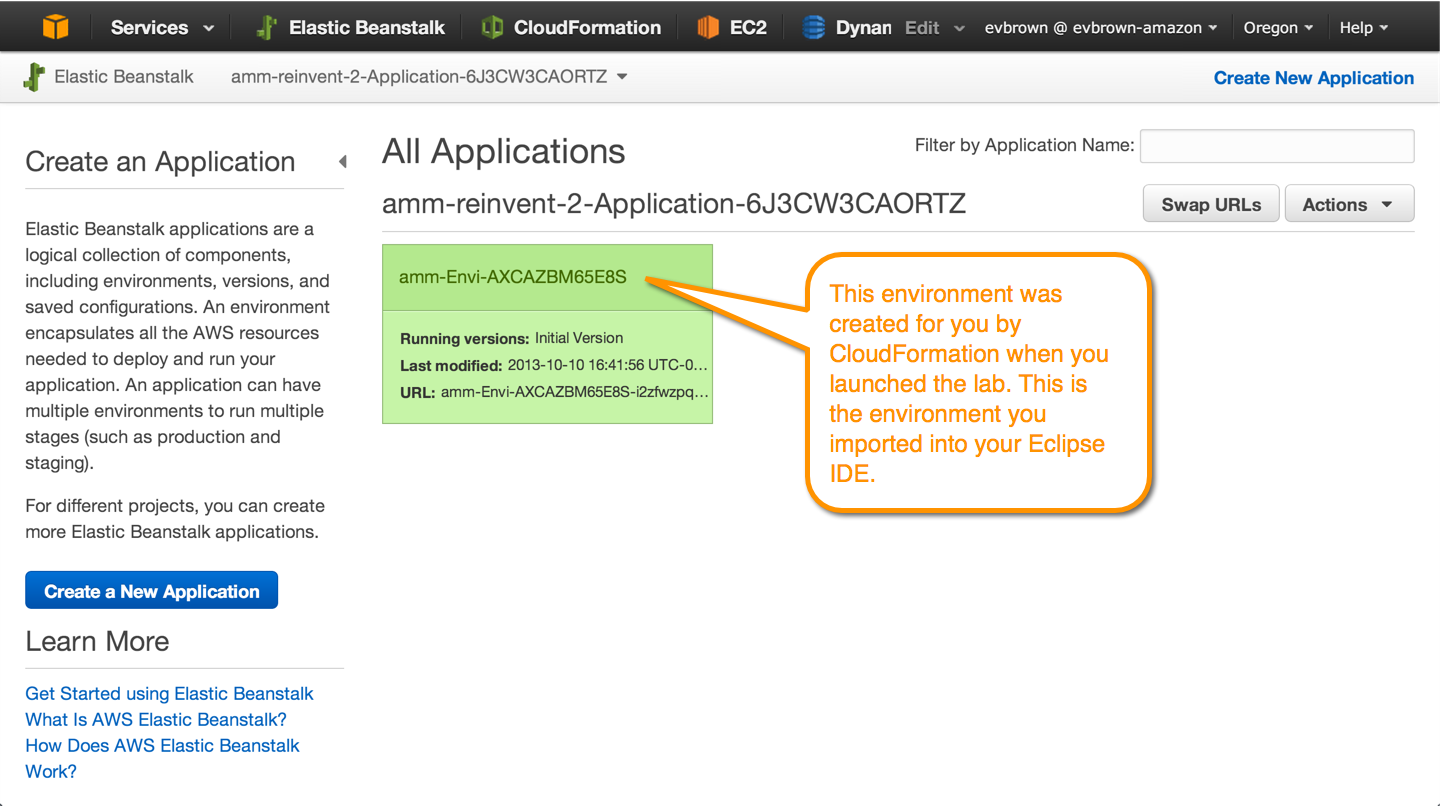
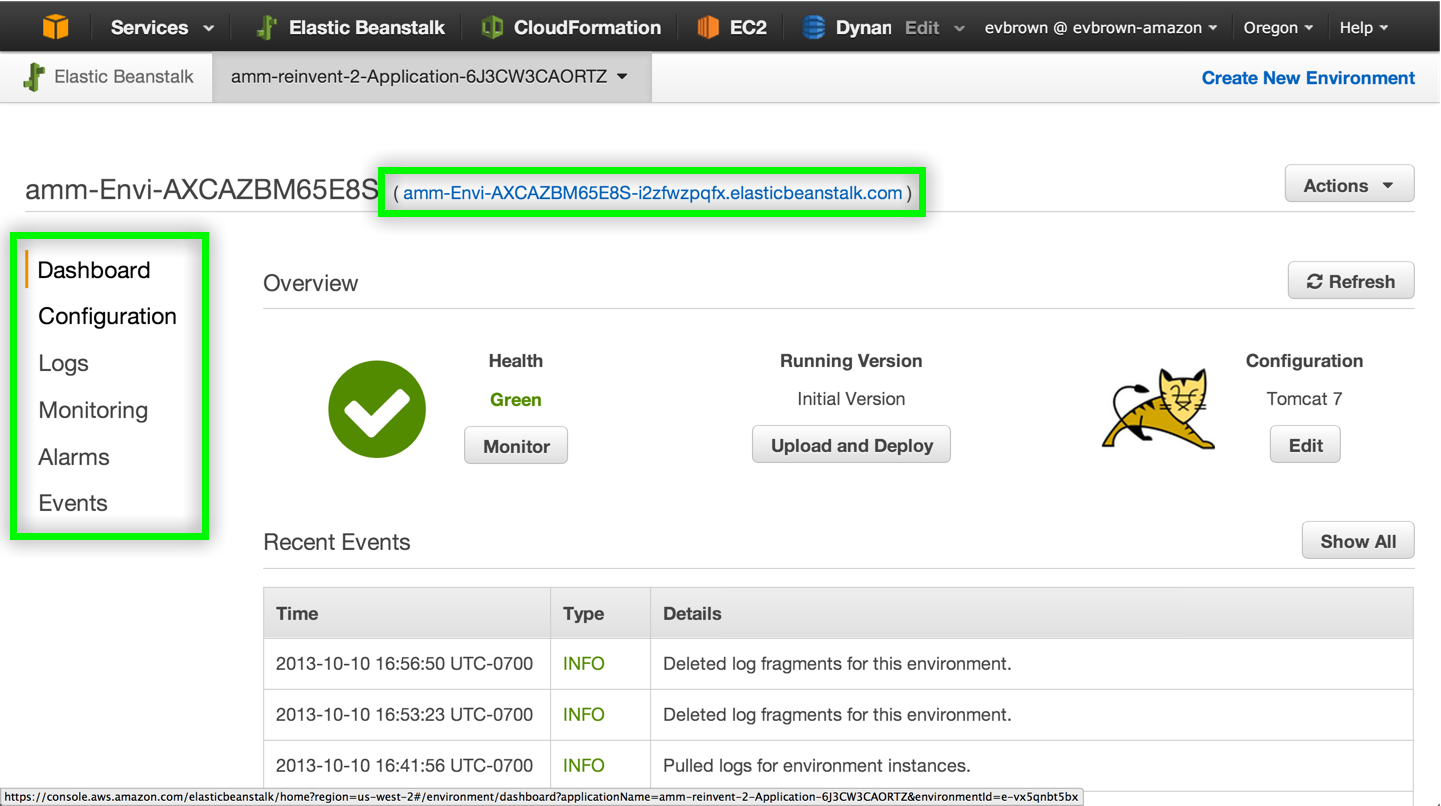
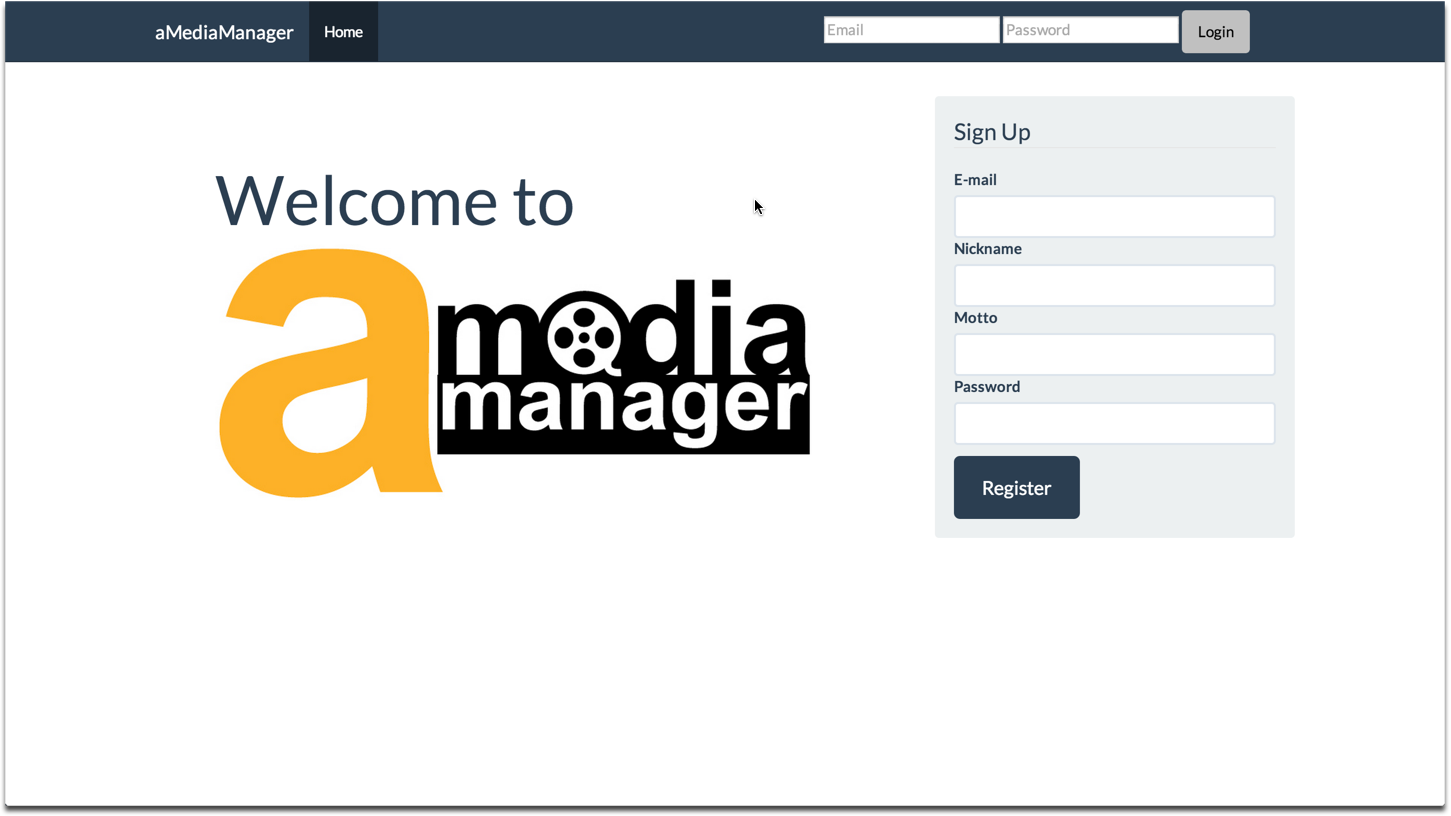
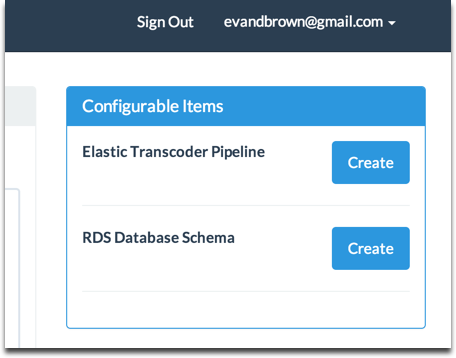
In the previous *Deploy the App* section of this lab, you used CloudFormation to launch your EB application and all of its dependencies (i.e., database, S3 bucket, etc). You should now have an Elastic Beanstalk environment available that is hosting an initial version of the sample application. In the following steps, you will import that environment into your IDE using the AWS Toolkit for Eclipse. This will allow you to deploy your project to Elastic Beanstalk directly from Eclipse.

1. Right-click the Servers view area in Eclipse and choose New > Server:  
   
2. Locate and select the AWS Elastic Beanstalk for Tomcat 7 server type and click Next:  
   
3. Click the link to Import an existing environment into the Servers view:  
   
4. Select the Elastic Beanstalk environment to import (there should be only 1) and click Finish:  
   
5. The environment will be imported and you may click Cancel to exit the New Server wizard:  
   
6. You should now see your Elastic Beanstalk environment as an available server in the Eclipse servers view:  
   
7. Double-click the Elastic Beanstalk server to view its complete configuration within your IDE:  
   
8. Click the Logs tab followed by the refresh button in the top-right of the screen to view a snapshot of the logs from the servers running in your Elastic Beanstalk environment:  
   

Open the Elastic Beanstalk Console

# Open the Elastic Beanstalk Console

In the previous section you viewed the configuration of your Elastic Beanstalk environment using the AWS Toolkit. In this section, use the Elastic Beanstalk Management Console to view and manage your environment from a web browser.

1. Navigate to <https://console.aws.amazon.com/elasticbeanstalk> in a web browser
2. If this is your first time signing into the console, see Appendix Q for instructions on authenticating using QwikLab credentials.
3. The Elastic Beanstalk dashboard shows a view of your applications and environments. Your environment should be in the GREEN state:  
   
4. Click the environment’s name to drill down. The Environment dashboard allows you to view and control the environment’s Configuration, Logs, Monitoring, Alarms, and Events, as well as view the running application via the URL at the top:  
   
5. Click your environment’s URL to view the version of the application that was deployed automatically when you started the lab:  
   
6. Sign up for a new account to access the app.
7. Click the App Config button to view an admin page:  
   
8. Click to create any Configurable Items:  
   

## OK, Now What?

We’re so close to writing code! There’s just one short section to complete before we start developing and shipping. Quickly, let’s recap what you’ve done so far:

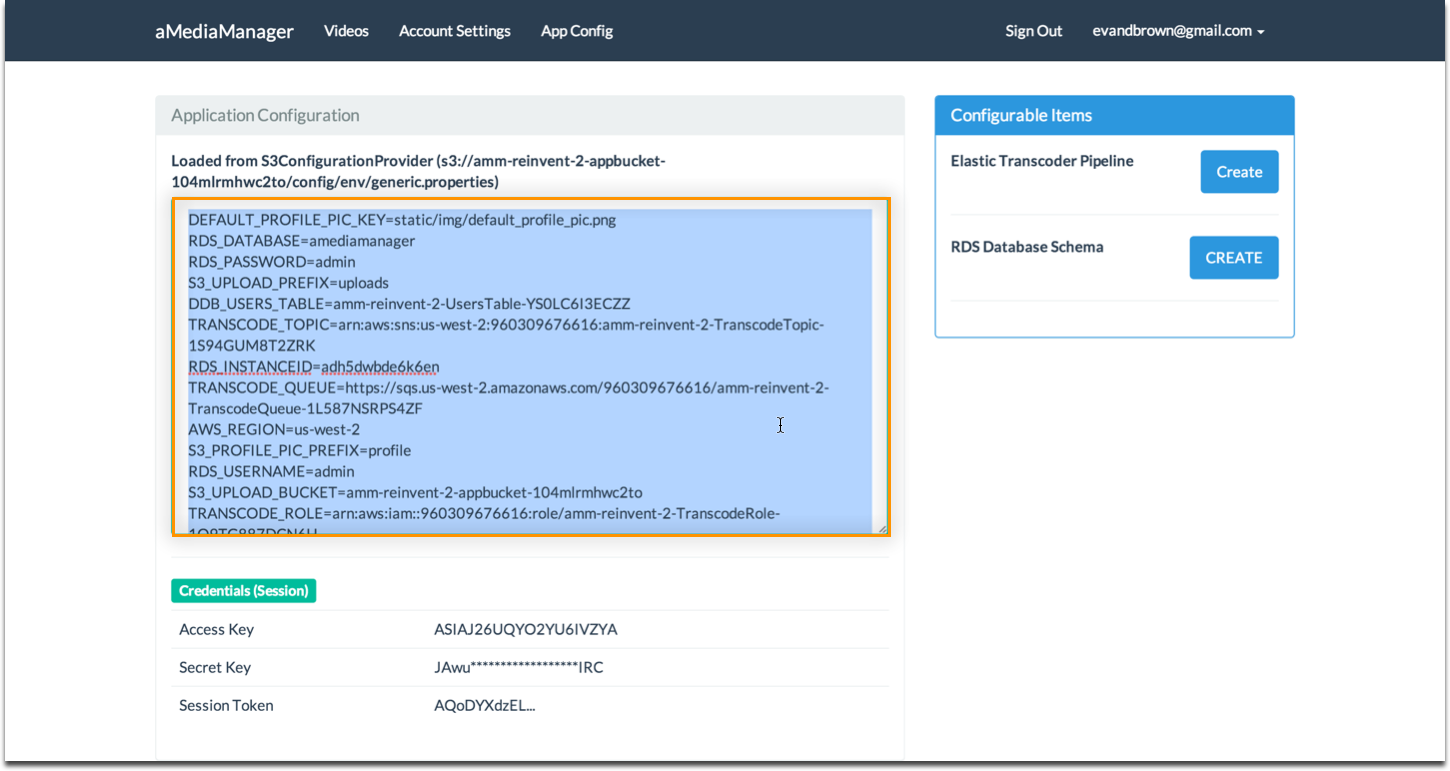
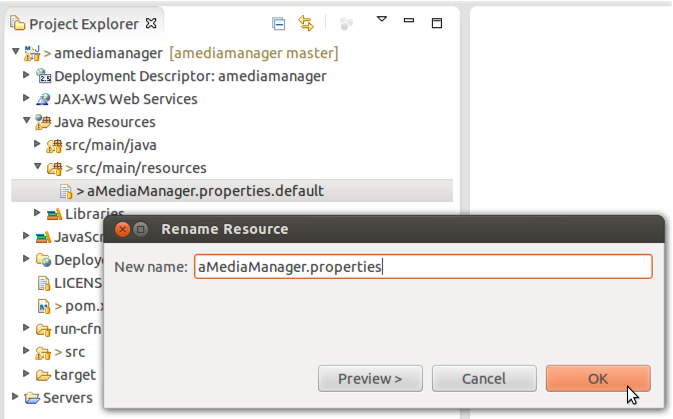
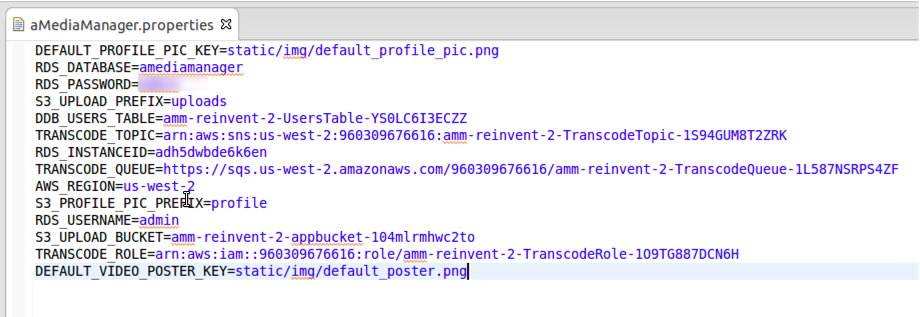
* Used QwikLab to deploy the sample application to Elastic Beanstalk
* Configured Eclipse with your AWS API keys (and remember, you didn’t store them in code or a version-controlled file. Good work!)
* Downloaded and imported the app source into a new project in Eclipse
* Imported the Elastic Beanstalk into Eclipse so you can deploy directly to it from the IDE

Import App Config for Local Dev

# Import App Config for Local Dev

When you started the lab in the very first section, CloudFormation provisioned your Elastic Beanstalk environment and all of your app’s dependencies, things like an RDS database, S3 bucket, DynamoDB table, etc. CloudFormation told Elastic Beanstalk about the names of those resources, and Elastic Beanstalk is making them available to your application via environment variables. The application version running in EB (which you just looked at in a browser in the last section) is good-to-go with its configuration.

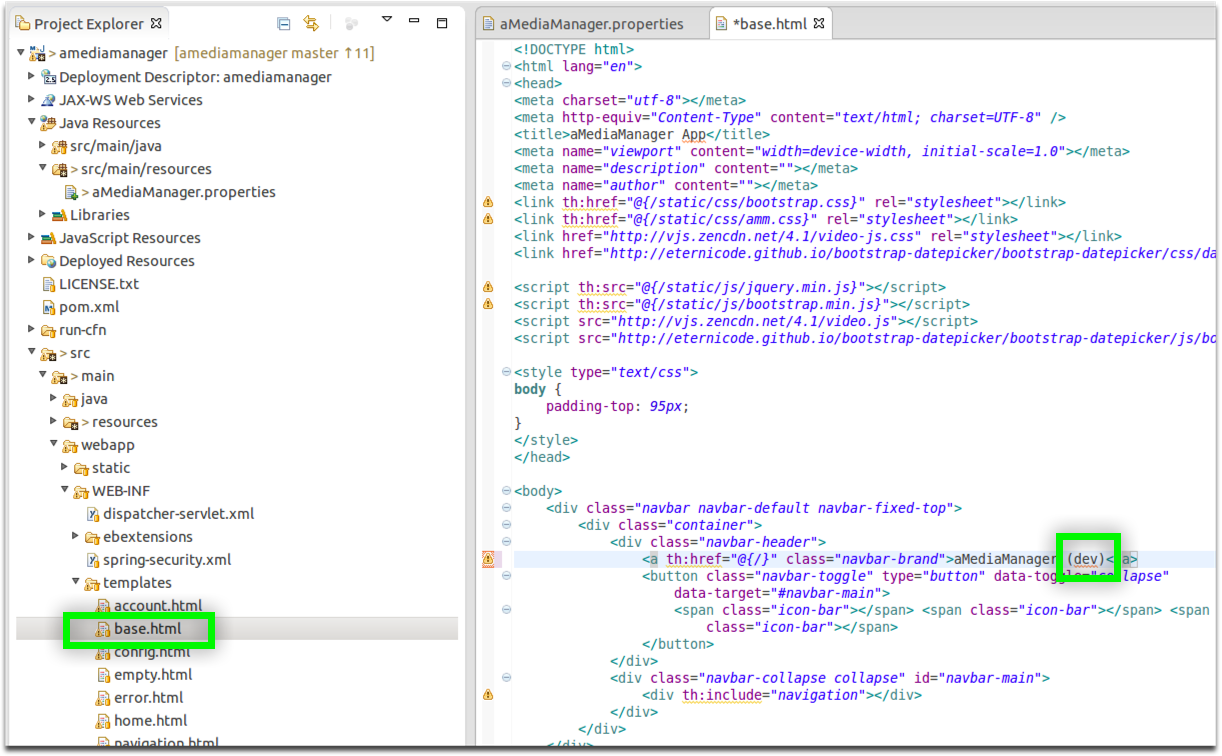
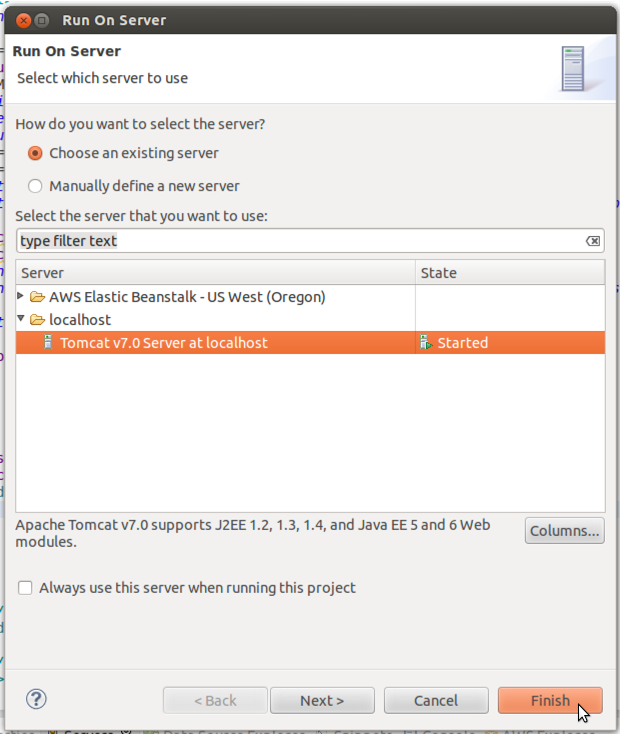
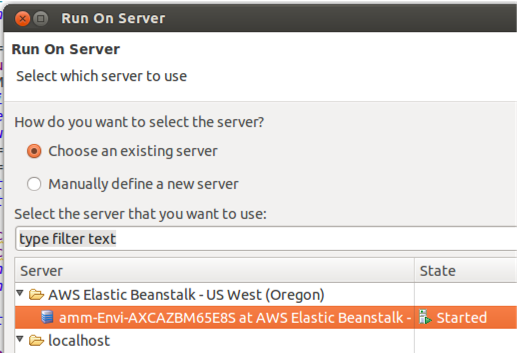
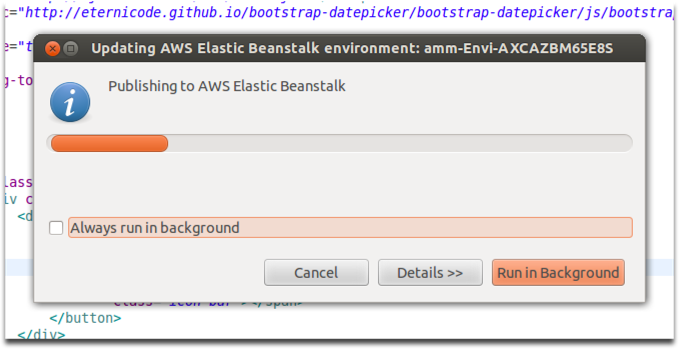
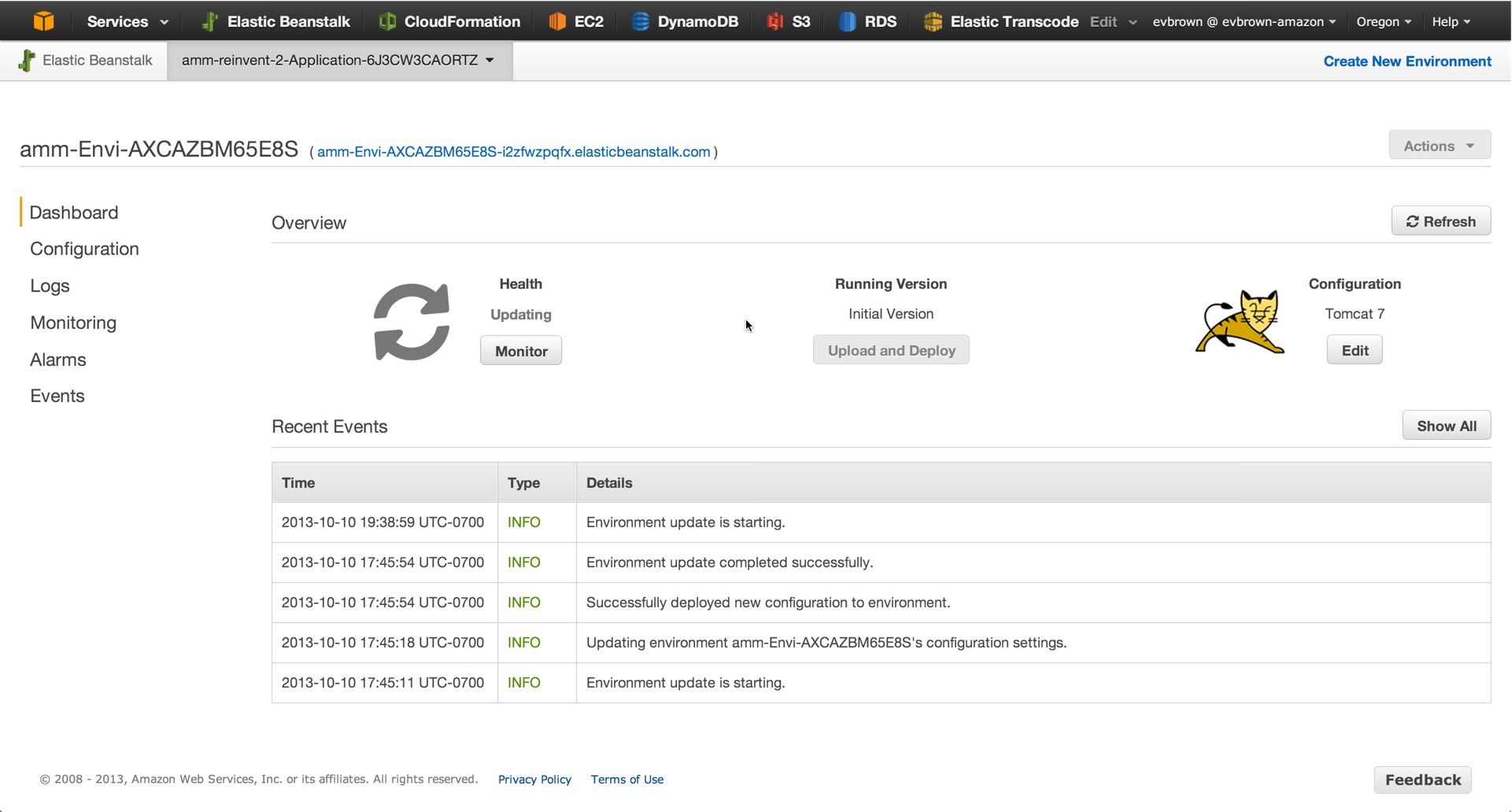
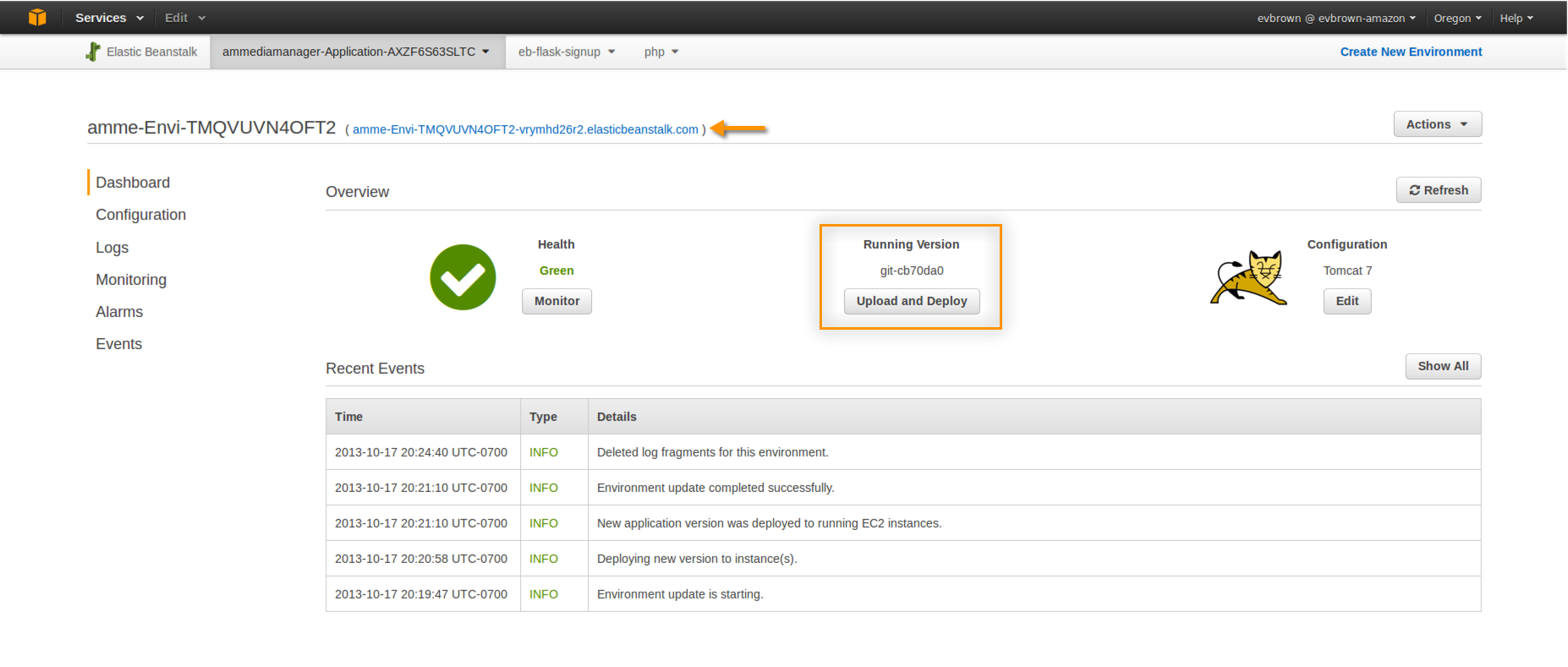
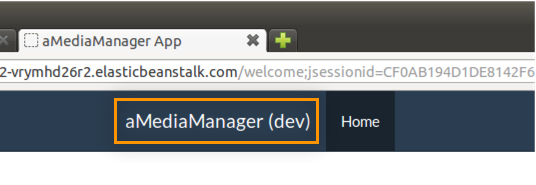
For this lab we want our local development environment to have the same configuration as the application deployed in EB. We’ve added a simple way to copy configuration from the app in EB to your local development environment:

1. From the App Config page, copy the configuration in the textarea to your clipboard:  
   
2. In Eclipse, navigate to amediamanager > Java Resources > src/main/resources and rename aMediaManager.properties.default to aMediaManager.properties:  
   
3. Open the new aMediaManager.properties file and paste in the configuration you copied previously:  
   

Modify and Deploy Application

# Modify and Deploy Application

In this section you will make a small change to the application, then deploy it to both your local Tomcat 7 server as well as your Elastic Beanstalk environment.

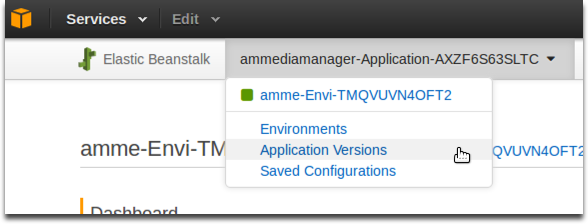
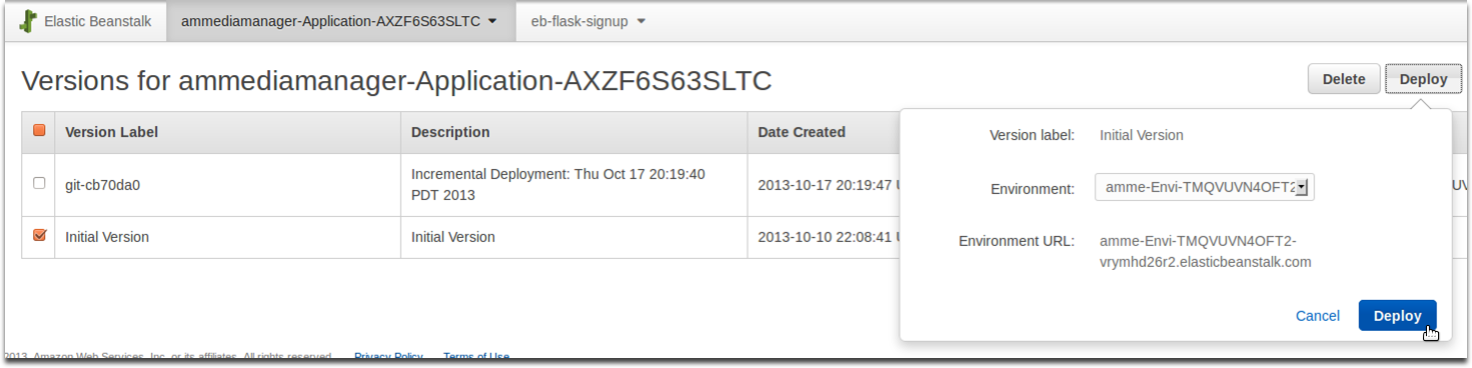
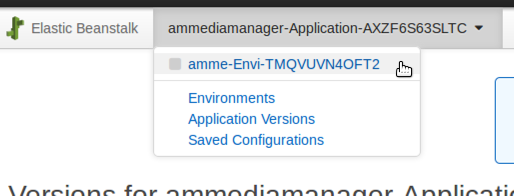
1. Open src > main > webapp > WEB-INF > templates > base.html and append “(dev)” to the navbar header:  
   
2. Right-click the amediamanager project and choose Run As > Run on Server
3. First, deploy the app to Tomcat v7.0 Server at localhost:  
   
4. After the application deploys it should open either in your default web browser or in a browser tab in Eclipse.
5. Deploy the application to Elastic Beanstalk by again right-clicking the project, choosing Run As > Run on Server, and choosing your Elastic Beanstalk environment from the server list:  
   
6. Eclipse will indicate the progress of your deployment in the IDE:  
   
7. Open your Elastic Beanstalk environment in a web browser to view its status, which should be Updating while the new version is deployed from Eclipse:  
   
8. Once your environment is Green, notice the new application version that is running, and click the environment URL to view it:  
   
9. Note the change with the updated application version:  
   

Roll Back to Initial Version

# Roll Back to Initial Version

When you deploy your application to Elastic Beanstalk – whether from Eclipse, the Management Console, or directly via the API – a new Application Version is created and stored in an S3 bucket in your account. Application Versions are retained until you delete them, and Elastic Beanstalk allows you to deploy specific versions with just a few clicks.

In this exercise, rollback to the previous (i.e. initial) Application Version to undo the changes you just made:

1. In the Elastic Beanstalk Management Console, click your application in the navigation bar and select Application Versions:  
   
2. Choose the Initial Version, click Deploy, choose your Environment and click Deploy again:  
   
3. Navigate back to your environment, wait for the deployment to complete, then open the application and confirm that it reverted:  
   

Project Anatomy

# Project Anatomy

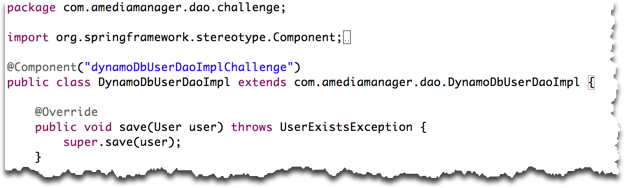
As you’ve already noticed, the application you deployed already works. Nothing’s broken because all of the functionality is already implemented, but you’ll still be writing a lot of code. Here’s how:

* We’ve identified a number of pieces of core application functionality (e.g., storing User data in DynamoDB)
* Defined an Interface for each piece of functionality
* Built a working implementation of that interface
* Subclassed the implementation with a *“challenge”* class that @Overrides superclass methods and calls super.method().
* We use Spring to inject the “challenge” implementation.

The result is that everything just works…until you decide to delete the *super.method()* line of a challenge implementation and DIY.

## Example of a Challenge

Later in the lab we’ll ask you to store a user’s account data in DynamoDB. In the **com.amediamanager.dao.challenge** package you’ll find the **DynamoDbUserDaoImpl** class that extends **com.amediamanager.dao.DynamoDbUserDaoImpl**, overrides the **save** method, and calls **super.save** from the super class:



You’ll start by commenting out **super.save(user);** and writing your own code. If you get stuck, simply open **com.amediamanager.dao.DynamoDbUserDaoImpl** and look at our implementation for hints. If you get really stuck and want to move onto the next challenge, uncomment the call to **super** and move right along.

## Package Layout

**com.amediamanager.config** – Classes for loading and accessing application configuration information (e.g., database connection strings, resource identifiers, etc); classes for provisioning/initializing certain resources (e.g., database schemas)

**com.amediamanager.config** – Coding challenges for config classes

**com.amediamanager.controller** – Controllers (Spring Web MVC) that define application routes. Any URL you access in the application is defined here, and it’s a good place to begin if you want to identify the code behind a feature in the UI

**com.amediamanager.dao** – Data Access Objects that handle storing data in various providers. An interface is defined for each data type (e.g., User), and an implementation is built specific to the datastore (e.g., DynamoDB)

**com.amediamanager.dao.challenge** – Coding challenges for DAO classes

**com.amediamanager.domain** – Domain or entity classes (e.g., User, Video)

**com.amediamanager.exceptions**– Exactly what it sounds like

**com.amediamanager.scheduled** – Classes with methods that need to run periodically (usually via Spring’s @Scheduled annotation)

**com.amediamanager.scheduled** – Coding challenges for scheduled tasks

**com.amediamanager.service** – Classes that implement business logic and broker communication with Data Access Objects (e.g., UserService)

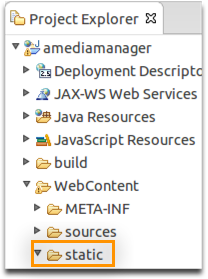
**com.amediamanager.service.challenge** – Coding challenges for the service layer

**com.amediamanager.service.springconfig**– Annotation-based configuration for Spring. Nothing to see here

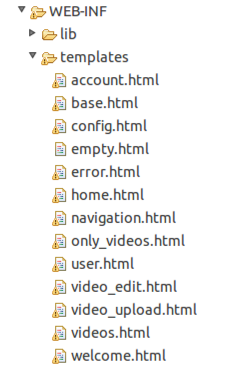
**com.amediamanager.util** – You gotta have a util package, right? These helper classes provide general helper functionality and won’t be a focus of any labs

## Templates and Static Assets

ThymeLeaf is the template engine we use, and Bootstrap is used for layout. You can find static assets for the latter in WebContent > Static:

****

ThymeLeaf templates are a little deeper, located at WebContent > WEB-INF > templates:



Challenge: Get App Config From S3

# Challenge: Get App Config From S3

There a number of really good reasons to store your application’s configuration in S3: it’s durable, centralized, and allows you to easily apply the same configuration to an arbitrarily large number of application servers. In this challenge, you are charged with using the S3 API to retrieve a key=val config file from a bucket and loading it into a Properties object. Your application will use these Properties for its configuration.

## How Did Config Get to S3, Anyways?

When you started this lab, we used a CloudFormation template to create all of your application’s dependencies: an empty S3 bucket, an RDS database, a DynamoDB table, and a few other things. CloudFormation then provided the values of those new resources to the EC2 instances in your Elastic Beanstalk application via environment variables with the *AMM\_* prefix. Your EC2 instance read those values when it booted and uploaded them to S3.

If you finish this challenge with time to spare, there is a **Detail Detour** at the end that will uncover more about how your app config got to S3.

## The Challenge

Implement the **loadProperties()** method of **com.amediamanager.config.challenge.S3ConfigurationProvider.**

## Super Powers

**super.getBucket()**

**super.getKey()**

**super.setProperties()**

### Resources

[The AWS SDK for Java](http://aws.amazon.com/sdkforjava/)

[JavaDoc](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html)

[The AmazonS3 Interface](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/s3/AmazonS3.html)

### If You Get Stuck

Take a peek at the solution in **com.amediamanager.config.S3ConfigurationProvider**

If you get really stuck, simply **return super.loadProperties();**

## Detail Detour

If you’ve finished the challenge with time to spare and are interested in how our app config was pushed to S3, you’re in the right place.

TODO: FINISH THIS PART

Challenge: DynamoDB and Users

# Challenge: DynamoDB and Users

For our application we’ve chosen to store user profile information in DynamoDB. In this challenge, you will persist a **com.amediamanager.domain.User** object to a DynamoDB table, and load the same object given a user’s e-mail address.

## About The DynamoDB Users Table

The table name is in the config (check out your super powers below for info on getting at the config). The table’s primary key name is also hinted at in the super powers section below.

## The Challenge

Implement the **save()** and **find()** methods of **com.amediamanager.dao.challenge.DynamoDbUserDaoImpl**

## Super Powers

**super.config**

**super.HASH\_KEY\_NAME**

**super.dynamoClient**

**super.getUserFromMap**

**super.getMapFromUser**

**super.LOG**

### Resources

[The AWS SDK for Java](http://aws.amazon.com/sdkforjava/)

[JavaDoc](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html)

The AmazonDynamoDB Interface

### If You Get Stuck

Take a peek at the solution in **com.amediamanager.dao.DynamoDbUserDaoImpl**

If you get really stuck, **super.save()** and **super.find()** will get you out of any bind.

Challenge: RDS, Read Replicas, and Connection Strings

# Challenge: RDS, Read Replicas, and Connection Strings

Our application stores metadata about videos (their location in S3, tags, ownership, etc) in a MySQL database. We use the Amazon Relational Database Service (RDS) to offload the heavy lifting of managing a database. When you launched the lab, CloudFormation created an RDS database for you.

In this challenge you will use the RDS API to discover the connection string of your master database and any Read Replicas. This connection string discovery code you write will automatically be invoked every 60 seconds, which means you can dynamically add or remove Read Replicas to scale the application’s database read performance without a code or config push. Pretty cool, and you can check out **com.amediamanager.dao.challenge.RdsDriverManagerDataSource** to see how your implementation is injected and used.

## The Challenge

Implement the **getMasterDbEndpoint()** and **getReadreplicaEndpoints()** methods of **com.amediamanager.dao.challenge.RdsDbEndpointRetriever**

## Super Powers

**super.config**

**super.rds**

**super.LOG**

### Resources

[The AWS SDK for Java](http://aws.amazon.com/sdkforjava/)

[JavaDoc](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html)

The AmazonRDS Interface

### If You Get Stuck

Take a peek at the solution in **com.amediamanager.dao.RdsDbEndpointRetriever**

If you get really stuck, **super.getMasterDbEndpoint()** and **super.getReadreplicaEndpoints()** will free you up to work on the next challenge.

Challenge: S3 for Profile Pictures

# Challenge: S3 for Profile Pictures

In the first challenge you used DynamoDB to save and retrieve user account/profile information. Our application also allows users to customize their profile picture. It doesn’t make a lot of sense to store things images in DynamoDB when we S3 at our disposal. When a user uploads a custom profile pic, the **com.amediamanager.dao.DynamoDbUserDaoImpl.update()** method puts that image in S3 and updates the item in DynamoDB with a reference to the object.

## The Challenge

Implement the **uploadFileToS3()** method of **com.amediamanager.dao.challenge.DynamoDbUserDaoImpl**

## Important Considerations

* Watch out for profile pics from different users with the same name (define a good naming scheme for the photos)
* See **config.getProperty(ConfigProps.S3\_PROFILE\_PIC\_PREFIX)** for a pre-defined key prefix that isolates profile pics in their own keyspace within your S3 bucket
* The profile photo should have a public read ACL
* Return the URL (i.e. http://…) of the uploaded photo.

## Super Powers

**super.config**

**super.s3Client**

**super.LOG**

### Resources

[The AWS SDK for Java](http://aws.amazon.com/sdkforjava/)

[JavaDoc](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html)

The AmazonS3 Interface

### If You Get Stuck

Take a peek at the solution in **com.amediamanager.dao.uploadFileToS3**

If you get really stuck, **super.uploadFileToS3()** will set you free.

Challenge: Elastic Transcoder Service

# Challenge: Elastic Transcoder Service

After a video is uploaded, **com.amediamanager.controller.VideoController.videoIngest** calls **com.amediamanager.service.VideoServiceImpl.createVideoPreview()**, which uses the Elastic Transcoder Service APIs to create a conversion job in an Elastic Transcoder pipeline that will convert the uploaded original video into a format suitable for streaming. You’re certainly welcome to dig into the code that schedules the job, but the focus of this challenge is determining when a transcode job is done.

It’s often tempting to call the *Describe\** APIs for a particular service when you want to know the status of an asynchronous job. *Describe\** APIs, however, aren’t designed for this type of repeated polling. Services that do asynch work offer a more efficient notification pattern that usually involves SNS, SQS, or both. In this case, the Elastic Transcoder pipeline that we provisioned for our application publishes messages status messages to a Simple Notification Service Topic, which in turn places those messages durably in an SQS queue.

In this challenge you will write code to poll a queue for status messages, pass them off to a handler (if you’re feeling super ambitious, you can also implement the handler), and delete them when done.

## The Challenge

Implement the **checkStatus() and deleteMessage()** methods of **com.amediamanager.scheduled.challenge.ElasticTranscoderTasks.**

## Important Considerations

* Your **checkStatus()** implementation must call **super.handleMessage()** for each message it receives.

## Super Powers

**super.config**

**super.sqsClient**

**super.LOG**

### Resources

[The AWS SDK for Java](http://aws.amazon.com/sdkforjava/)

[JavaDoc](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html)

The AmazonSQS Interface

### If You Get Stuck

Take a peek at the solution in **com.amediamanager.scheduled.challenge.ElasticTranscoderTasks**

If you get really stuck, you know the drill by now.

Challenge: Emit Custom CloudWatch Metrics

# Challenge: Emit Custom CloudWatch Metrics

Our application uses a number of AWS APIs: DynamoDB, RDS, S3, SQS, and Elastic Transcoder to name a few. These API calls are over the network, which introduces a latency component. It’s also possible that our application could make requests that file (e.g., trying to upload a user profile pic to a bucket that doesn’t exist). There are certainly a number of scenarios worthy of monitoring; in this lab, we’ll focus on measuring API request latency and success status with Amazon CloudWatch.

Monitoring request latency and success is a crosscutting concern: we want to monitor every AWS API call made from our application, but we shouldn’t have to implement monitoring in every class we use the AWS SDK. In this application we use AspectJ and define an aspect with a Pointcut of “execution(public \* com.amazonaws.services..\*.\*(..))” to intercept all API calls from every AWS client. Don’t worry, AspectJ knowledge isn’t required for this lab; you’ll focus on implementing code to create and upload CloudWatch metrics for API latency and success.

## The Challenge

Implement the **emitMetrics()** method of **com.amediamanager.metrics.challenge.MetricAspect.** The method should use **super.metricBatcher.addDatum()** to add a metric for latency and success status. The method signature provides **long startTime** and **Throwable exception** parameters which will allow you to calculate latency and determine if the call was successful (i.e., did not throw an exception).

## Important Considerations

* Use **super.newDatum()** to create a basic **MetricDatum** object.
* Call the **withMetricName(), withUnit(),** and **withValue()** methods of the returned object to configure the datum.
* Use “Latency” for the latency metric name and “Success” for the success metric name.
* Use **StandardUnit.Milliseconds** for the latency unit and **StandardUnit.Count** for the success metric.
* Use **super.metricBatcher.addDatum()** to submit the datum you calculated. Use “AMM” as the value for the namespace argument to that method.

## Super Powers

**super.metricBatcher**

### Resources

[The AWS SDK for Java](http://aws.amazon.com/sdkforjava/)

[JavaDoc](http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html)

The com.amazonaws.services.cloudwatch.model.MetricDatum documentation

### If You Get Stuck

Take a peek at the solution in **com.amediamanager.metrics.MetricAspect**

If you get really stuck, fall back to **super.emitMetrics().**Nice job making it this far!