Table of Contents

Coding a Java Web Application

Start the Lab (Deploy the App)

Start the Lab (Deploy the App)

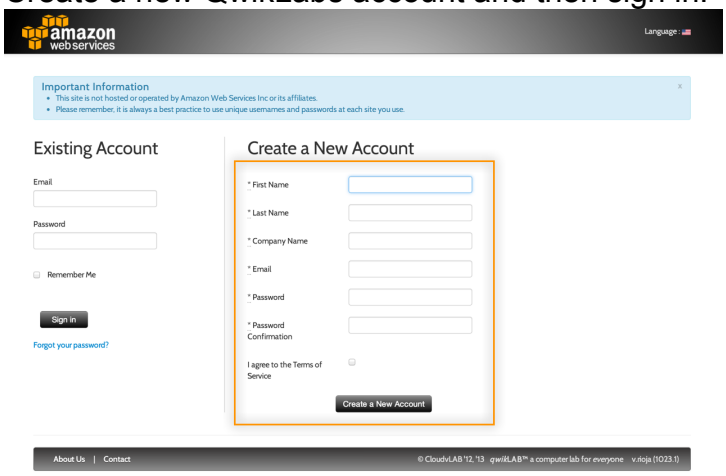
We're going to start by launching a fully functional and implemented version of our application. Throughout the workshop we'll visit individual components of the application and implement them ourselves.

You won't be using your own AWS Account for this workshop. Instead you will create an account with QwikLab which will provide you with a clean AWS account for the duration of the class, and will take care of cleaning it up when we finish at the end of the day.

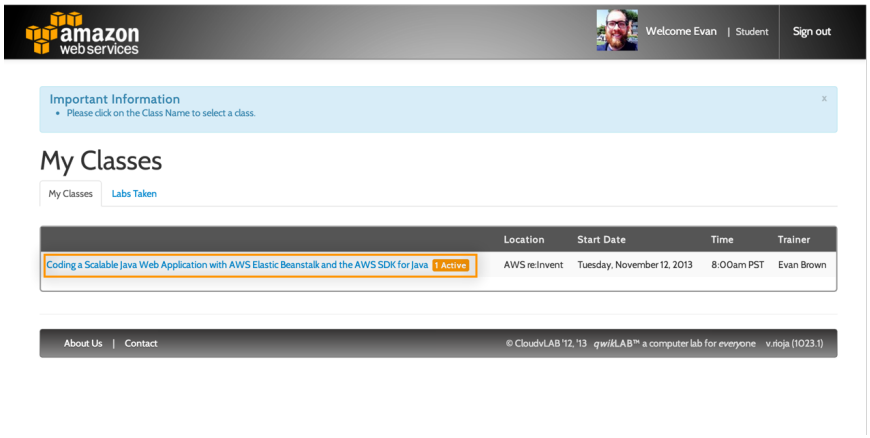
Start the Lab (Deploy the App)

1. Navigate to https://events-aws.qwiklab.com in your web browser

2. Create a new QwikLabs account and then sign in:

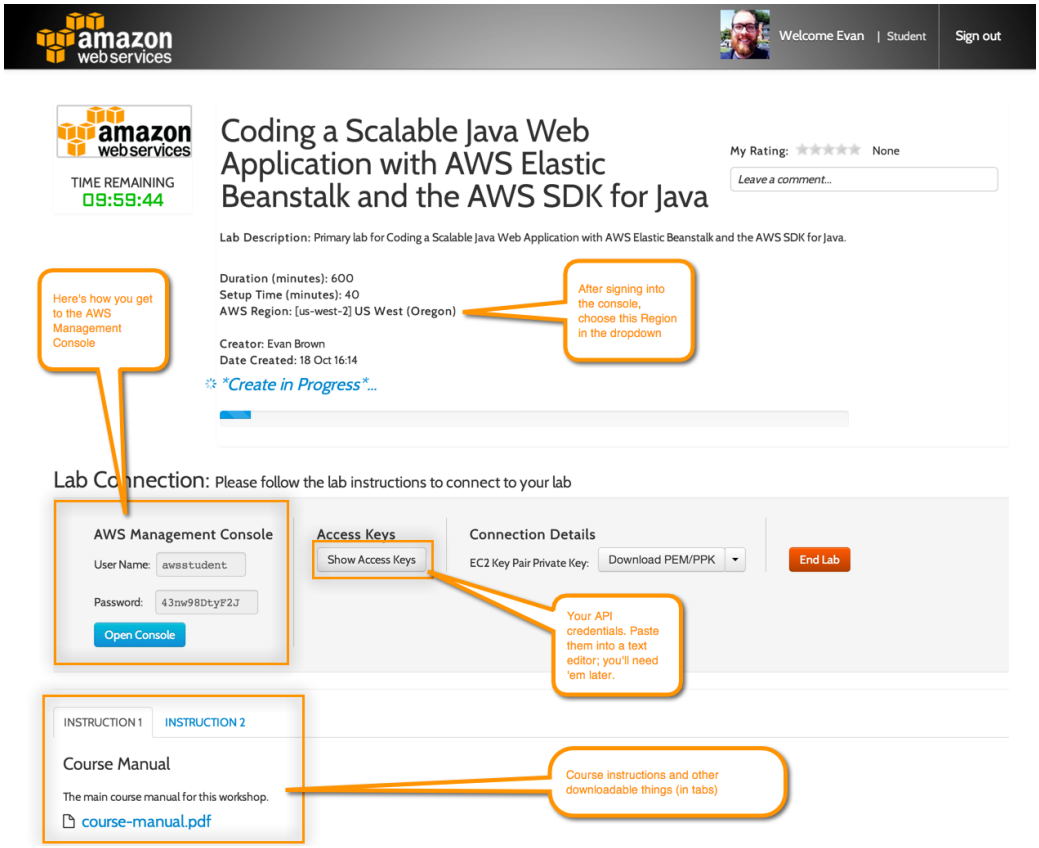


3. In My Classes, click on the 'Coding a Scalable…' class:



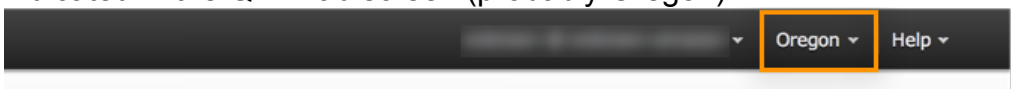
4. Click Start Lab:

5. The lab will take ~30 minutes to completely start. In the meantime, use the lab landing page to access the AWS Management Console and retrieve your Access Key and Secret Key:



6. Click the Open Console button and sign into the Console with the User Name and Password provided by QwikLab

7. Navigate to the CloudFormation Console and choose the AWS Region indicated in the QwikLab screen (probably Oregon):



8. Confirm that you have a CloudFormation stack being created, and read on for details on what the CloudFormation stack is doing.

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 as environment variables. When your application launches in Elastic Beanstalk, it will read those environment variables and upload them to S3 as a configuration file that other servers can use, and that you will configure your local dev environment to use.

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; and Amazon DynamoDB for storing user profile information. We develop the application locally in Eclipse and deploy it to AWS Elastic Beanstalk (EB), 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

Configure Credentials In Eclipse

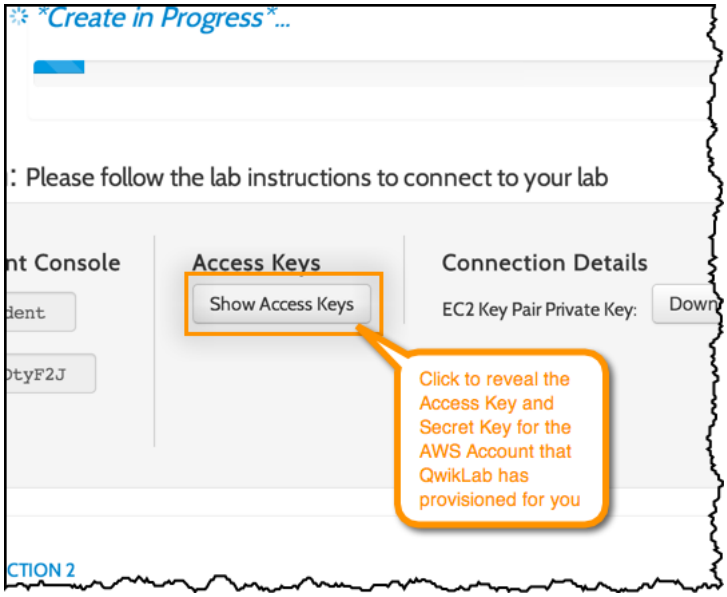
Configure Credentials in Eclipse

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.

9. Open the QwikLab tab in your browser, locate them, and copy them to a text editor:

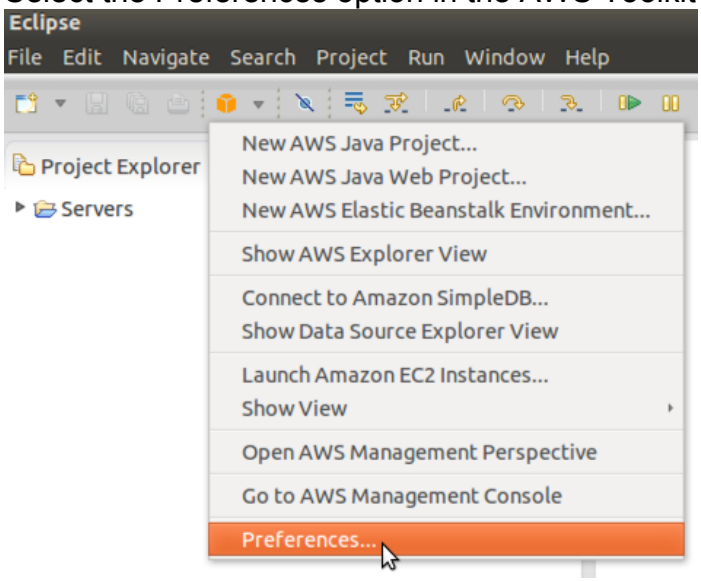


Note: If we weren't using QwikLab in this workshop, you would manage your keys by using the Identity and Access Management service in the AWS Management Console at https://console.aws.amazon.com/iam.

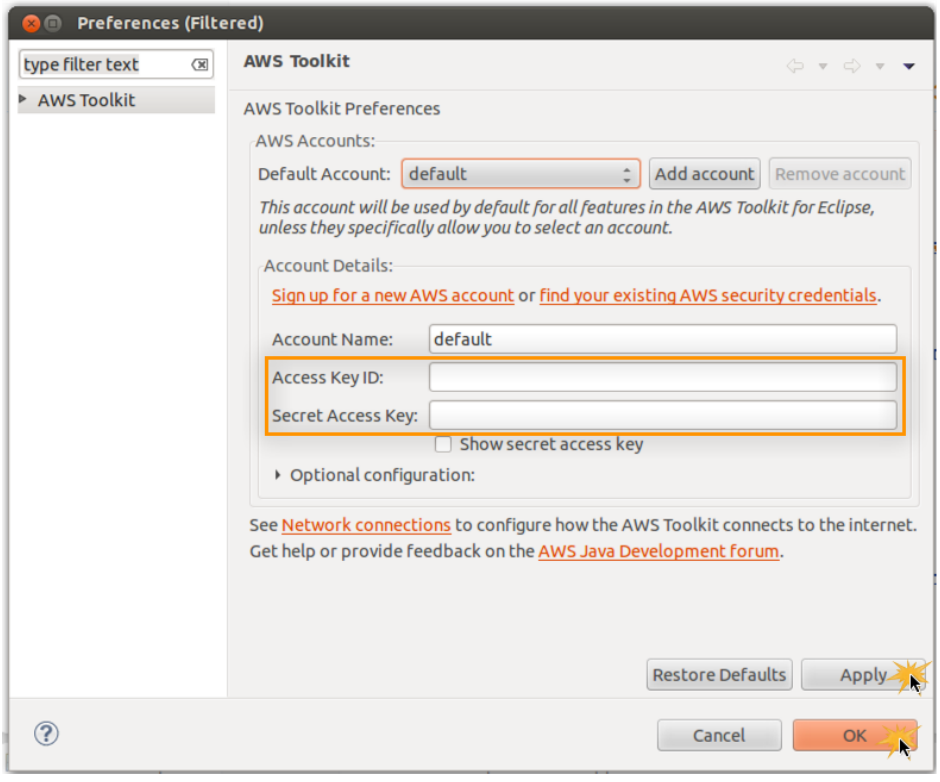
Configure the AWS Toolkit for Eclipse

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.

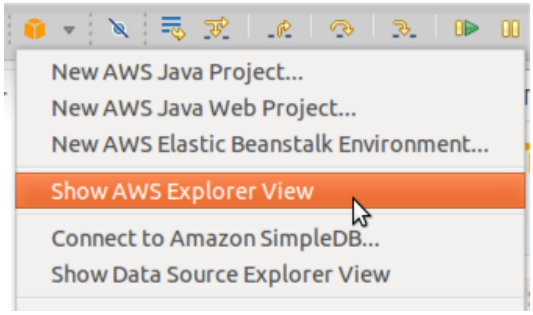
10. Select the Preferences option in the AWS Toolkit dropdown menu:

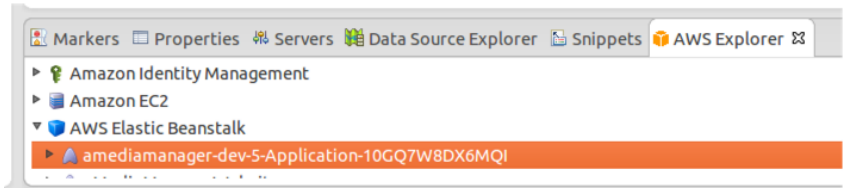


11. Enter the Access Key and Secret Key you retrieved from QwikLab, then click Apply and OK:



12. Confirm that your credentials are correctly configured by opening the AWS Explorer view and listing your AWS Elastic Beanstalk Environments:

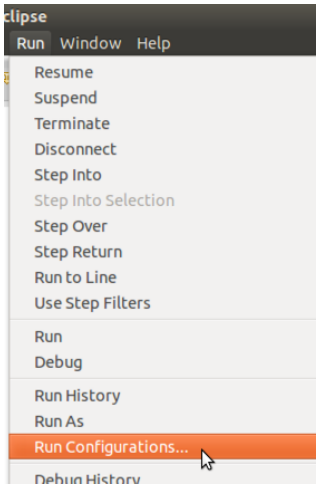




Provide AWS Credentials as JVM Args

The app you'll be building uses the AWS SDK for Java to make requests to services like DynamoDB, S3, RDS, etc. Those requests must be authenticated, and the SDK needs an Access Key and Secret Key to sign them. Of course, you would never embed those credentials in source code (right? right?!), and we would prefer to avoid entering them into a config file (where there's a risk of committing that file to source control). Instead, for local dev/test on your laptop (i.e., Eclipse + Tomcat 7, not Elastic Beanstalk), you will provide the Access Key and Secret Key to your app via JVM arguments:

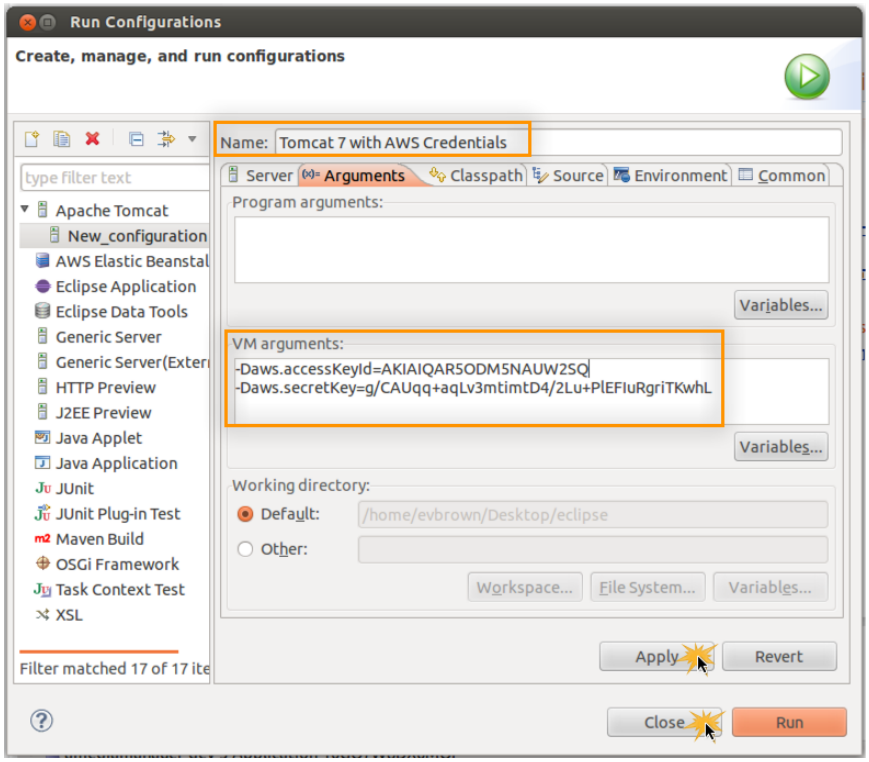
13. Click *Run* > *Run Configurations…*



14. Right-click *Apache Tomcat* and click *New*

15. Name the configuration *Tomcat 7 with AWS Credentials*, select the Arguments tab, and enter the following in the *VM arguments* text box, then click *Apply* and *Close:*

```
-Daws.accessKeyId=REPLACE_WITH_YOUR_ACCESS_KEY
-Daws.secretKey=REPLACE_WITH_YOUR_SECRET_KEY
```



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.

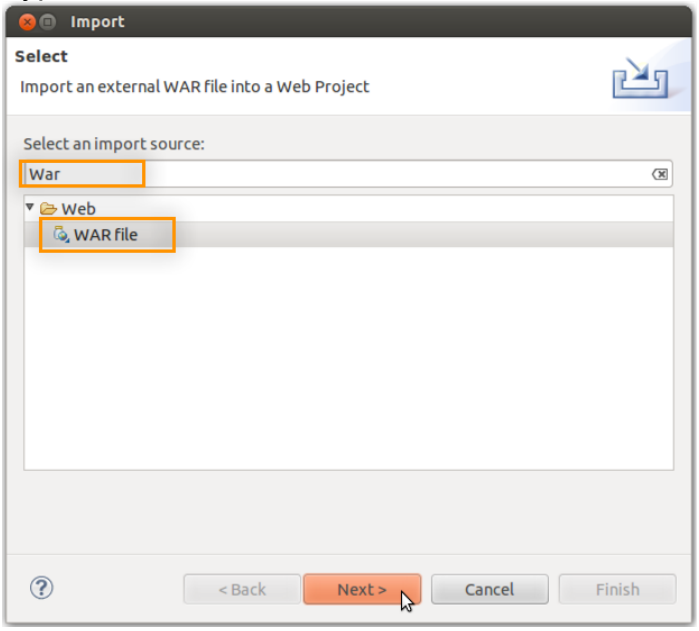
Developer's Note: The source is distributed as a WAR file for this workshop and will be made available on GitHub as a Maven project afterwards.

Import Application to Eclipse

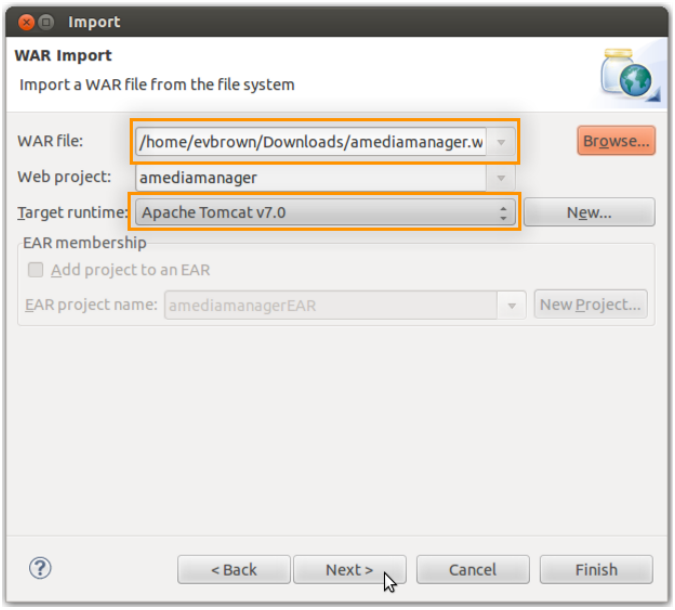
16. Download the application from http://amm-us-west-2.s3.amazonaws.com/public/amediamanager.war

17. In Eclipse, choose File > Import

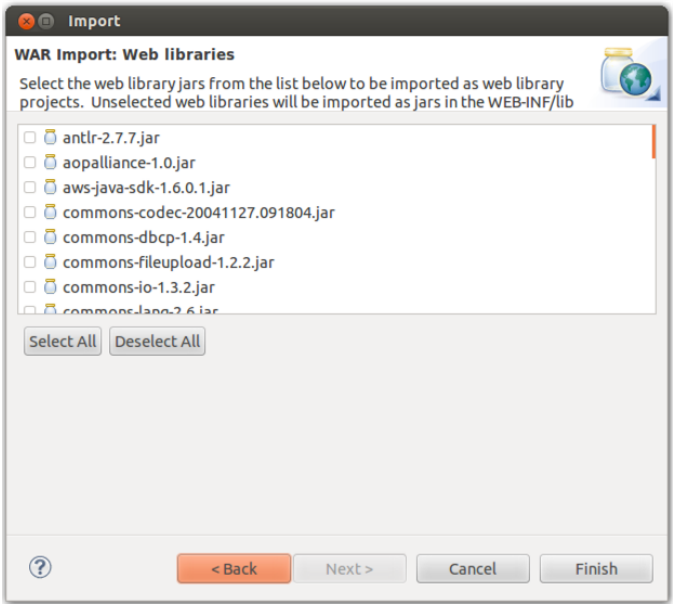
18. Type *WAR* in the filer box, choose WAR file, and click Next



19. Choose the WAR file you downloaded previously, ensure that Apache Tomcat 7.0 is is selected for *Target runtime*, and click Next. If you have no *Target runtime* options available, please see the pre-work document for information on installing Tomcat 7 on your local workstation:



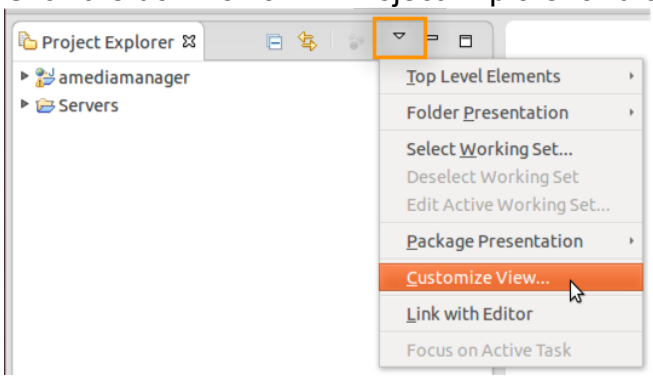
20. Do not select any WAR files to import. Click Finish:



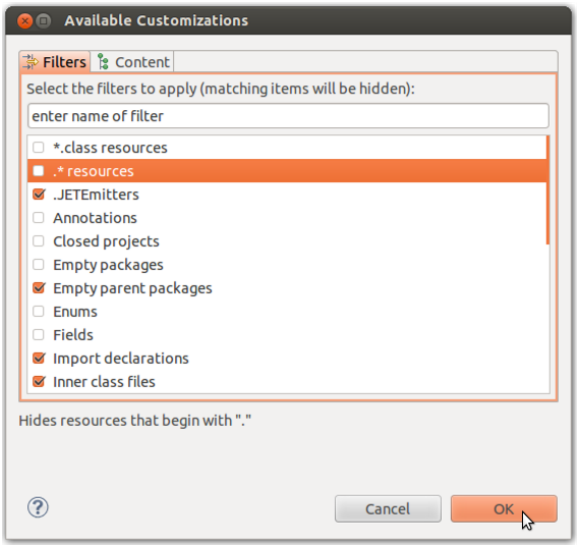
Configure Project in Eclipse

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

21. Click the down arrow in Project Explorer and choose Customize View:

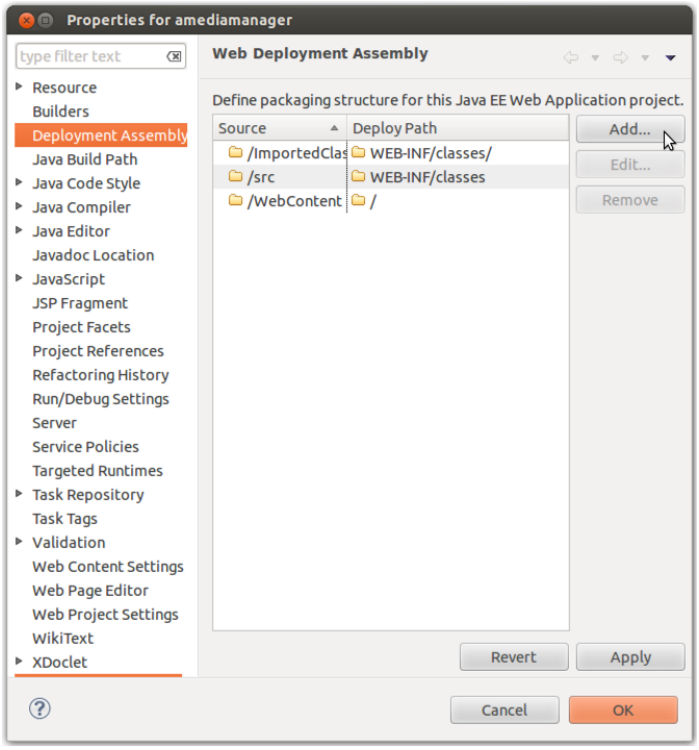


22. Uncheck *.* resources* and click OK:

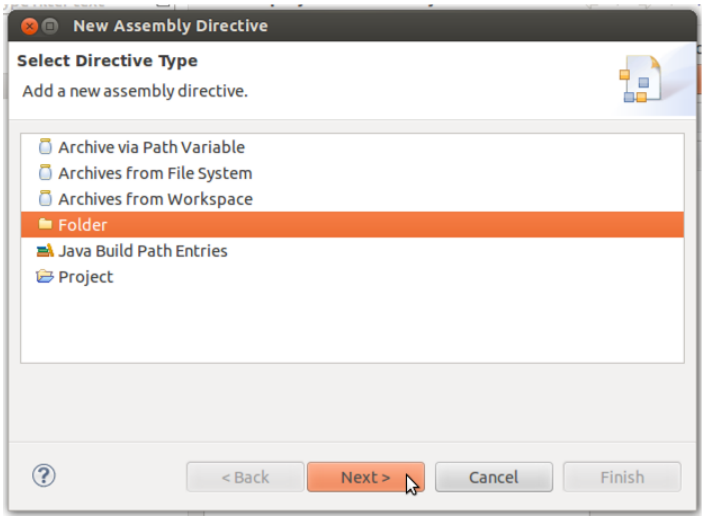


23. Right-click the amediamanger project and choose Properties

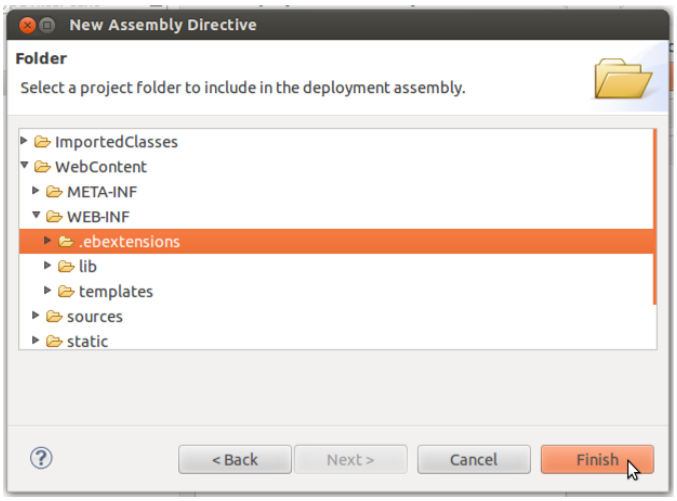
24. Select Deployment Assembly and click Add:



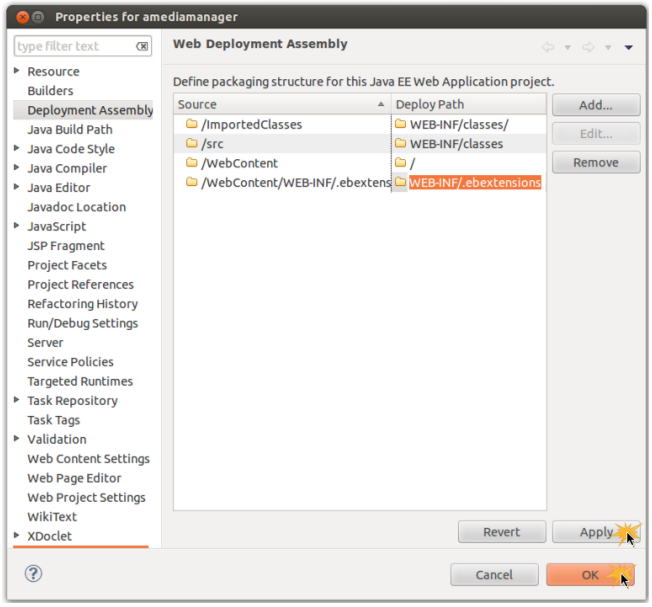
25. Select Folder and click Next:



26. Select the WebContent > WEB-INF > .ebextensions directory and click Finish:



27. Double-click the Deploy Path field for the .ebextensions Source, enter *WEB-INF/.ebextensions*, click Apply then OK:

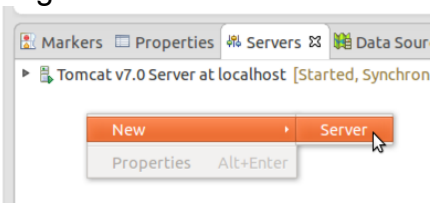


Import EB Environment Into Eclipse

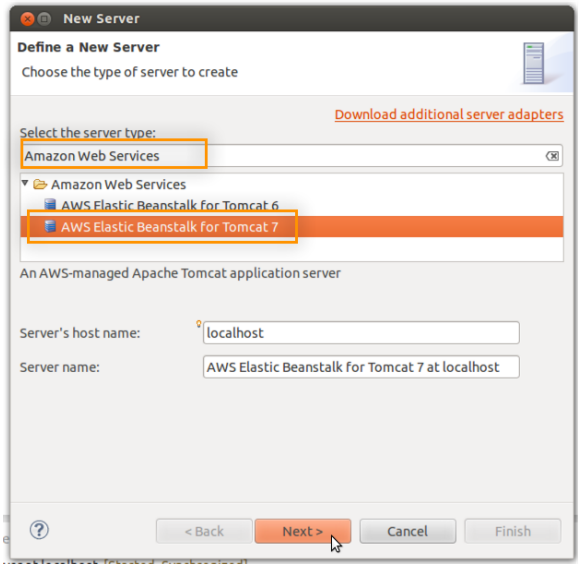
Import EB Environment Into Eclipse

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.

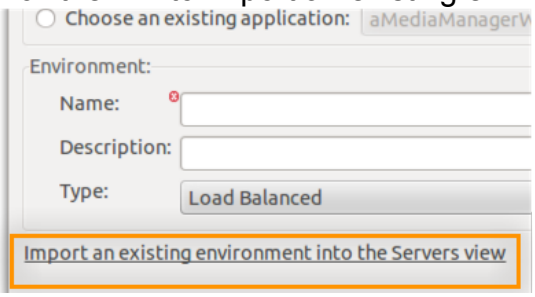
28. Right-click the Servers view area in Eclipse and choose New > Server:



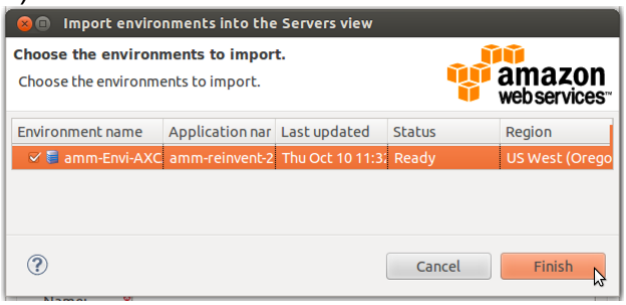
29. Locate and select the AWS Elastic Beanstalk for Tomcat 7 server type and click Next:



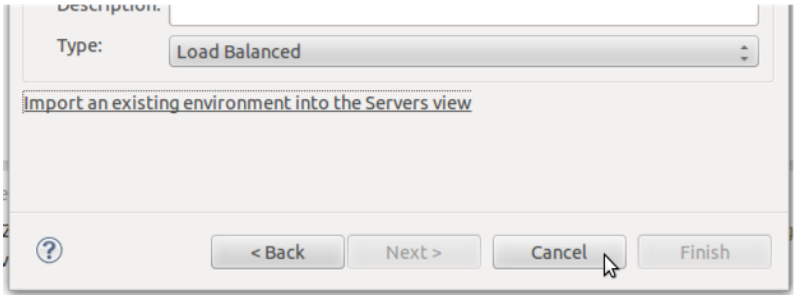
30. Click the link to Import an existing environment into the Servers view:



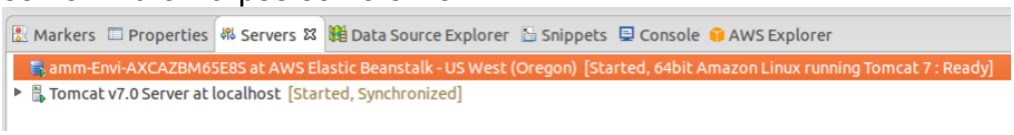
31. Select the Elastic Beanstalk environment to import (there should be only 1) and click Finish:



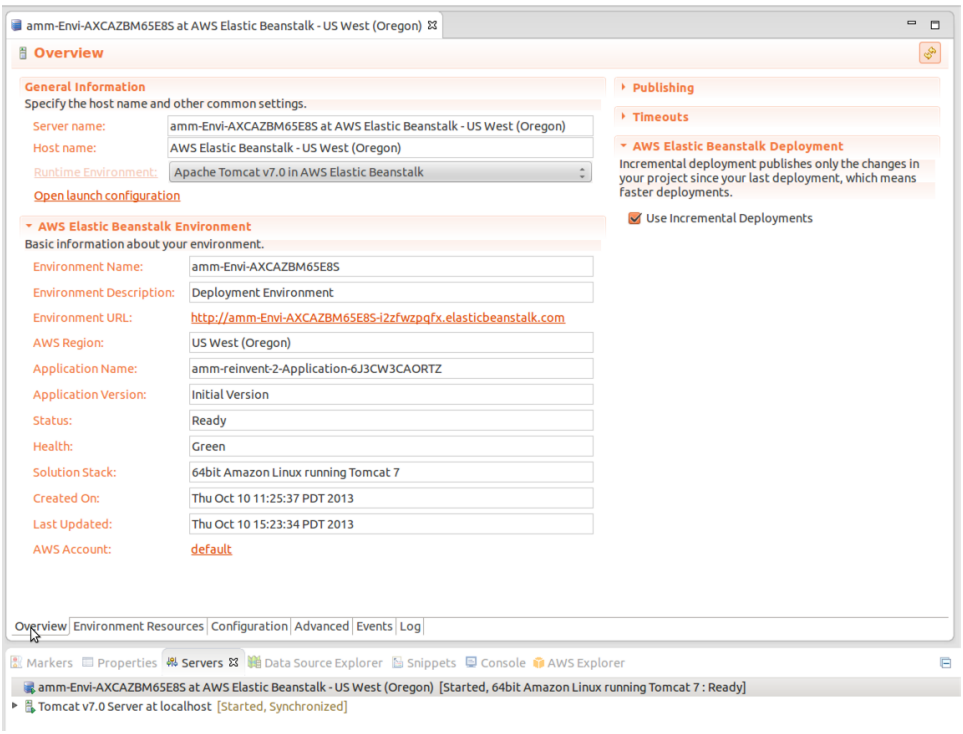
32. The environment will be imported and you may click Cancel to exit the New Server wizard:



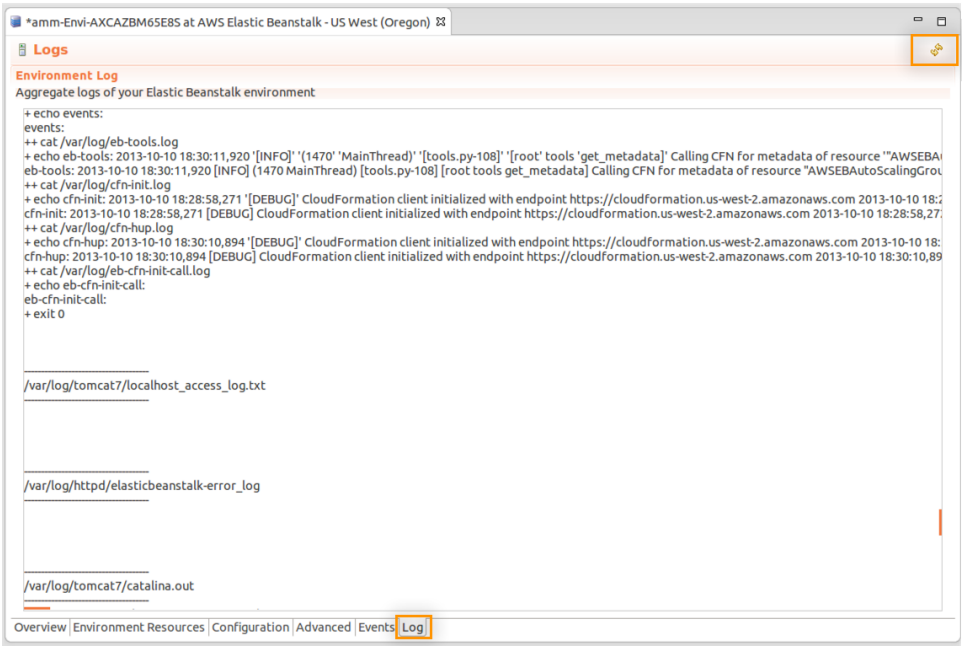
33. You should now see your Elastic Beanstalk environment as an available server in the Eclipse servers view:



34. Double-click the Elastic Beanstalk server to view its complete configuration within your IDE:



35. 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

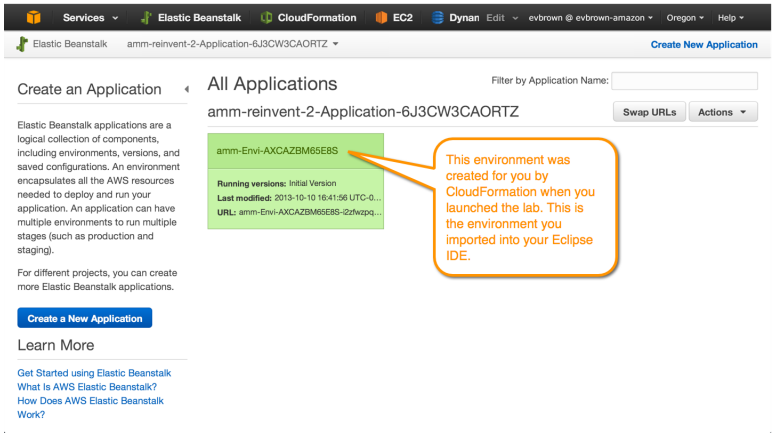
Open the Elastic Beanstalk Console

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.

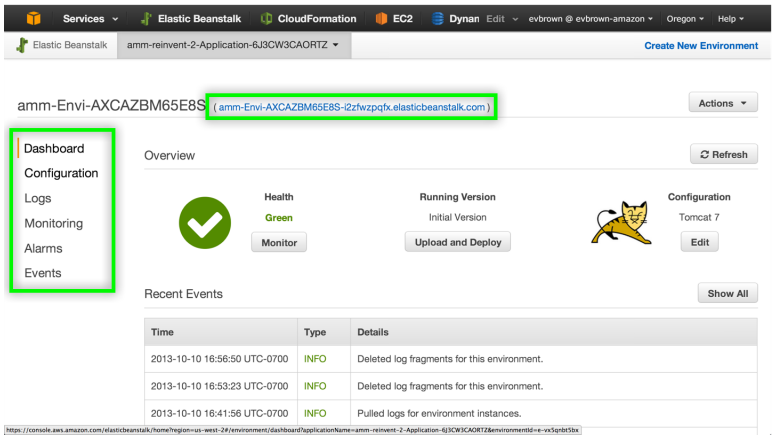
36. Navigate to https://console.aws.amazon.com/elasticbeanstalk in a web browser

37. If this is your first time signing into the console, see Appendix Q for instructions on authenticating using QwikLab credentials.

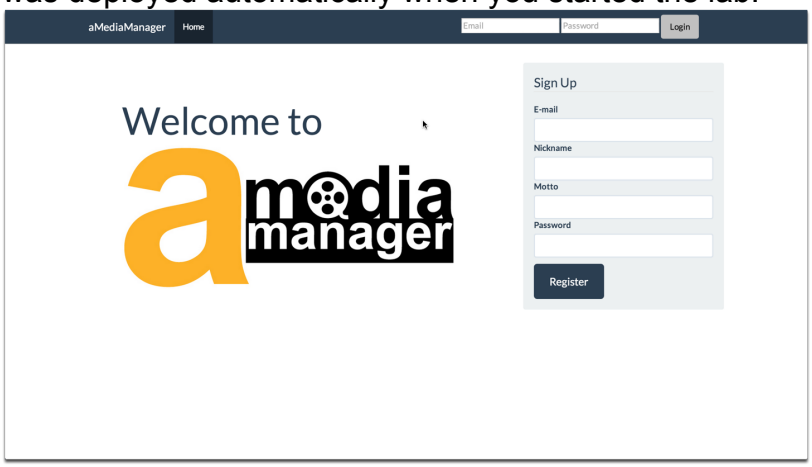
38. The Elastic Beanstalk dashboard shows a view of your applications and environments. Your environment should be in the GREEN state:



39. 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:



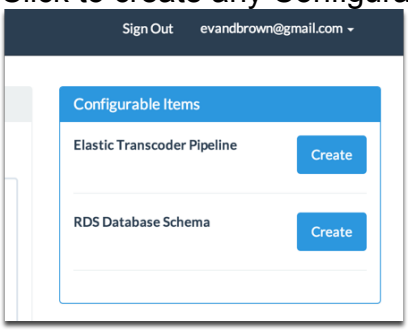
40. Click your environment's URL to view the version of the application that was deployed automatically when you started the lab:



41. Sign up for a new account to access the app.

42. Click the App Config button to view an admin page:

43. 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:

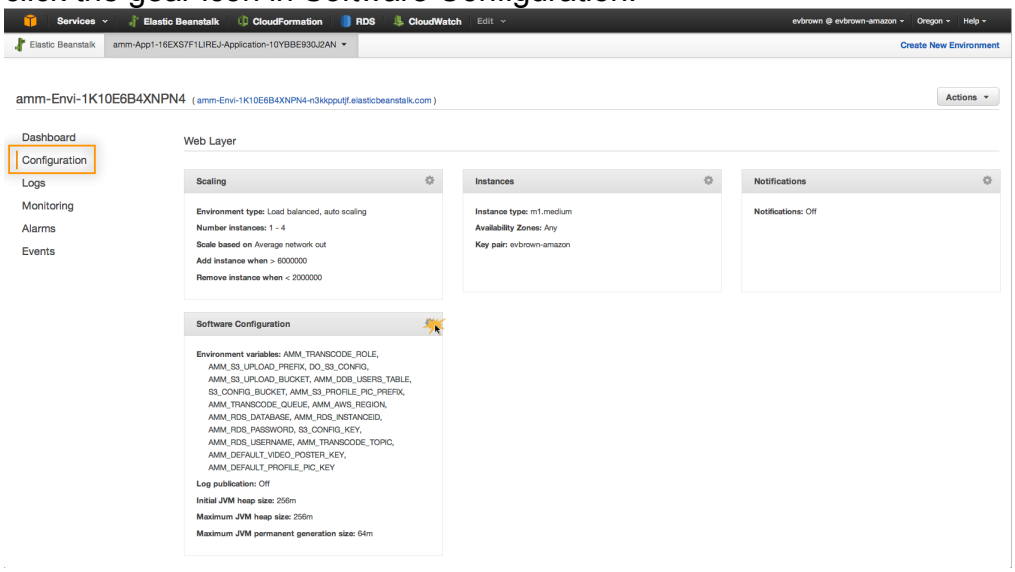
- Used QwikLab to deploy the sample application to Elastic Beanstalk
- Configured Eclipse with your AWS API keys (and remember, you didn't store them in code or a version-controlled file. Good work!)
- Downloaded and imported the app source into a new project in Eclipse
- Imported the Elastic Beanstalk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

Import App Config for Local Dev

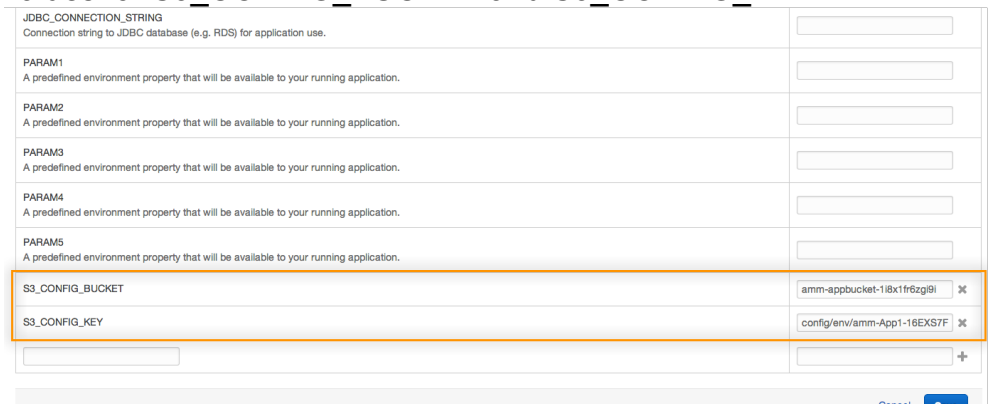
Import App Config for Local Dev

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 provided the names of each of those resources to your Elastic Beanstalk environment, and we had EB run a small Python script that uploaded the values to an S3 bucket. Our configuration is centralized in S3, the app deployed in EB is using it, and now we'll configure your local dev environment to use it as well.

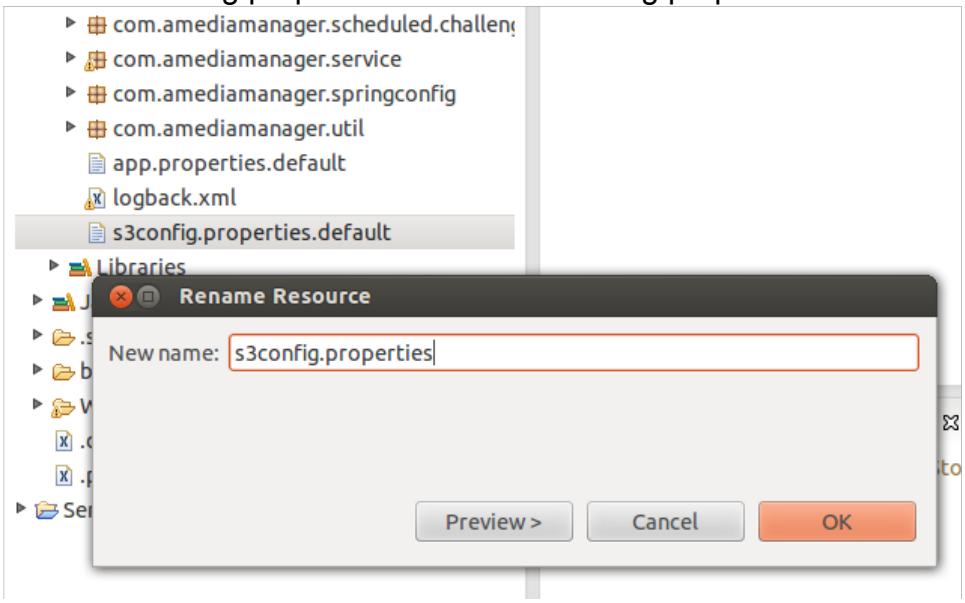
44. Click Configuration in the Elastic Beanstalk Management Console, then click the gear icon in Software Configuration:



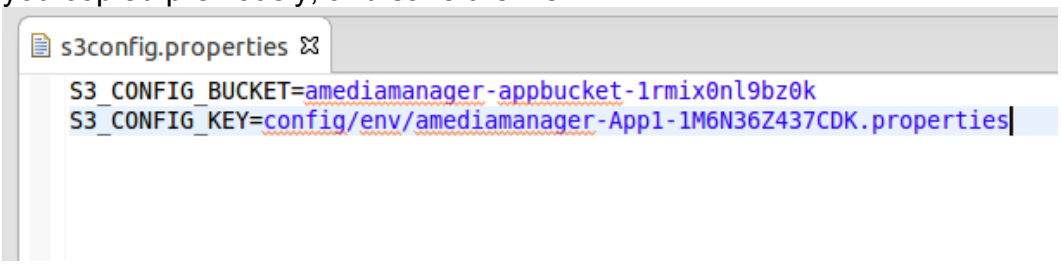
45. Scroll through the Environment Properties section and copy/paste the values for S3_CONFIG_BUCKET and S3_CONFIG_KEY:



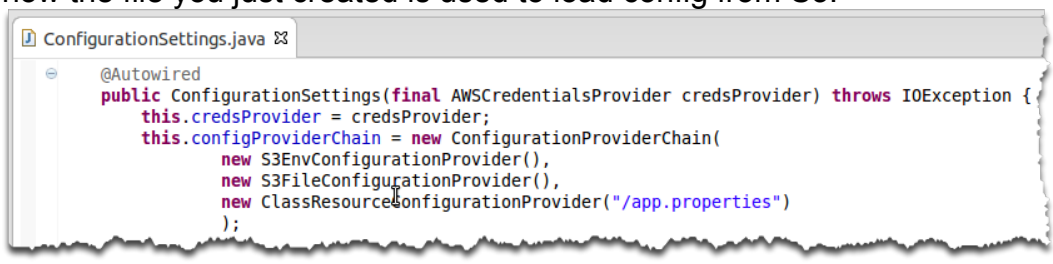
46. In Eclipse, navigate to amediamanager > Java Resources > src and rename s3config.properties.default to s3config.properties:



47. Open the new s3config.properties file, paste the S3 bucket and key values you copied previously, and save the file:



48. In **`com.amediamanager.config.ConfigurationSettings`** observe how **`this.configProviderChain`** is set, then take a look at **`com.amediamanager.config.S3FileConfigurationProvider`** to see how the file you just created is used to load config from S3:

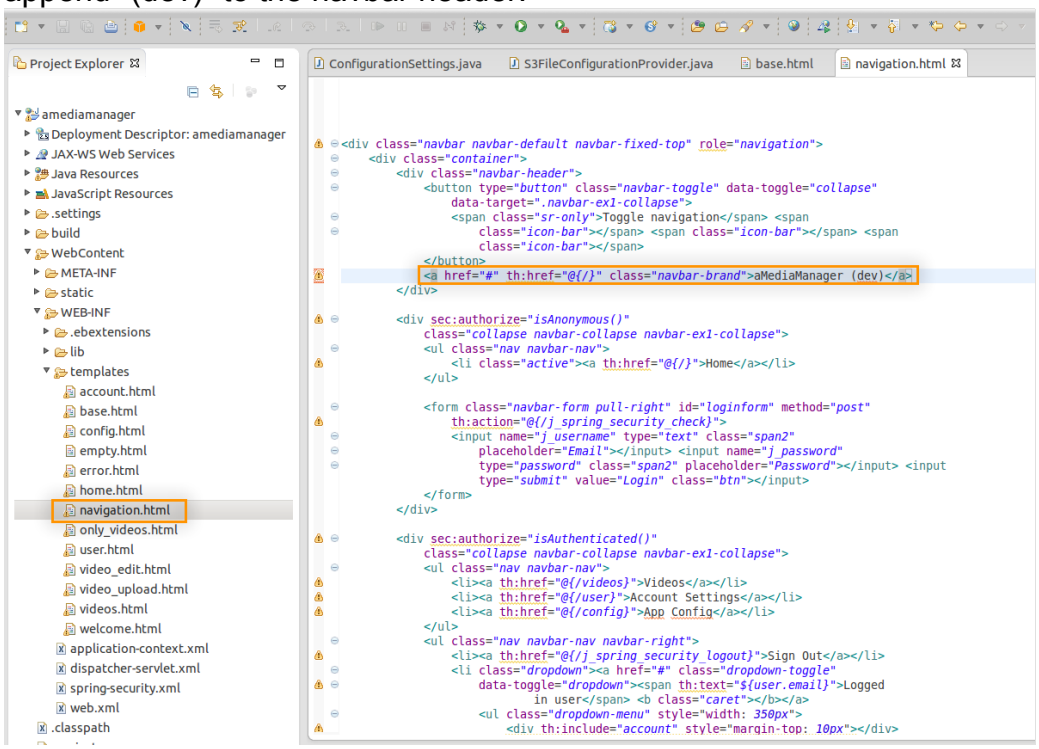


Modify and Deploy Application

Modify and Deploy Application

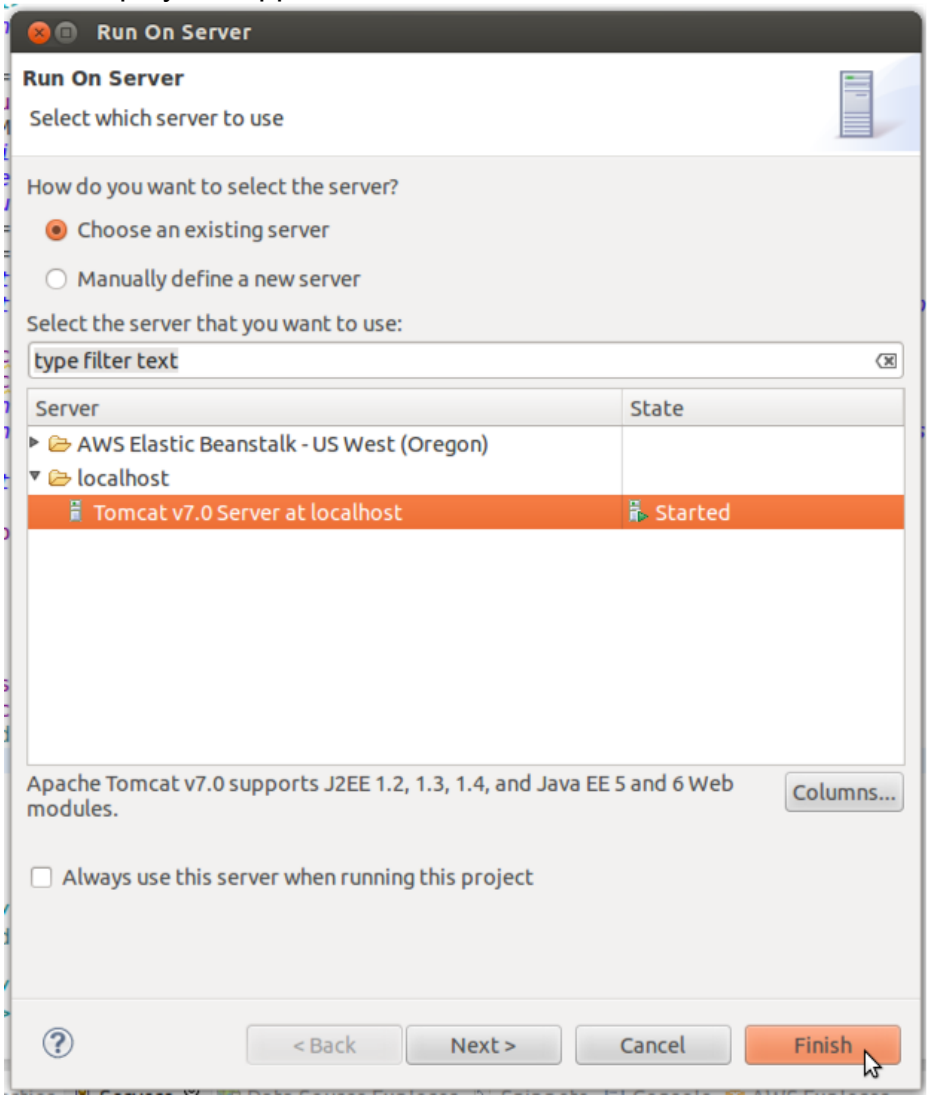
In this section you will make a small change to the application, then deploy it to both your local Tomcat 7 server as well as your Elastic Beanstalk environment.

49. Open **WebContent > WEB-INF > templates > navigation.html** and append “(dev)” to the navbar header:



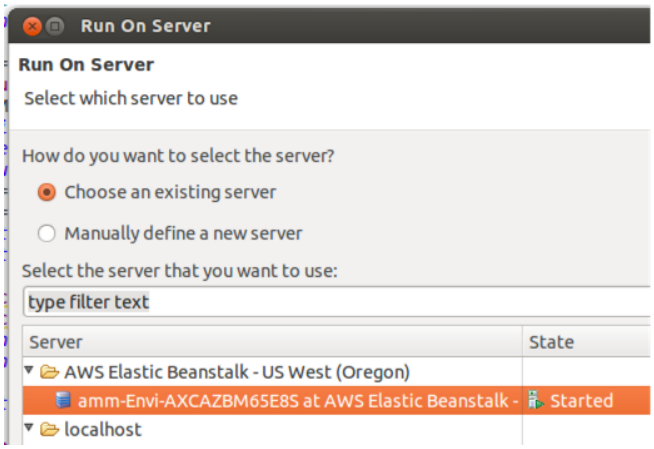
50. Right-click the amediamanager project and choose Run As > Run on Server

51. First, deploy the app to Tomcat v7.0 Server at localhost:

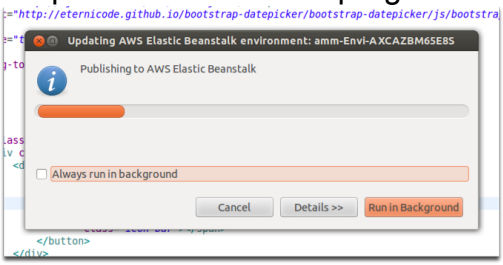


52. After the application deploys it should open either in your default web browser or in a browser tab in Eclipse.

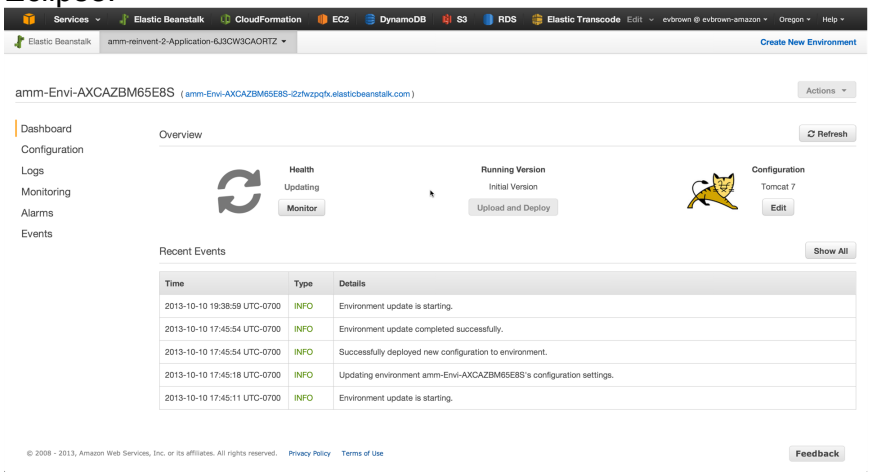
53. Deploy the application to Elastic Beanstalk by again right-clicking the project, choosing Run As > Run on Server, and choosing your Elastic Beanstalk environment from the server list:



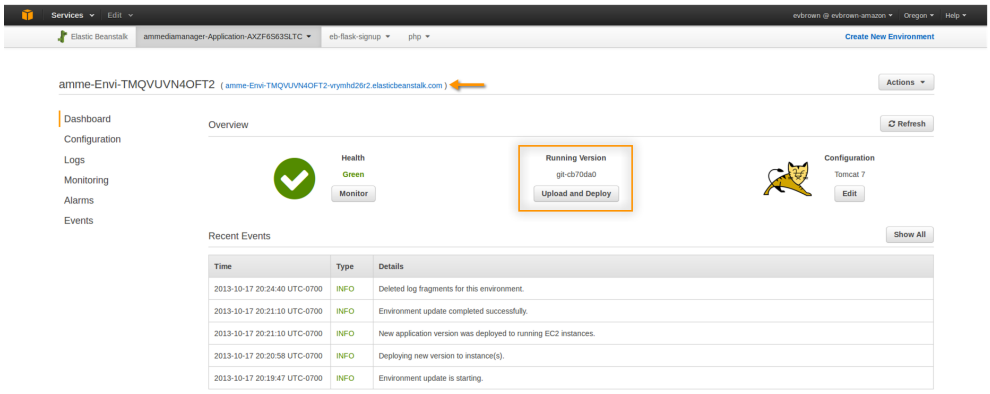
54. Eclipse will indicate the progress of your deployment in the IDE:



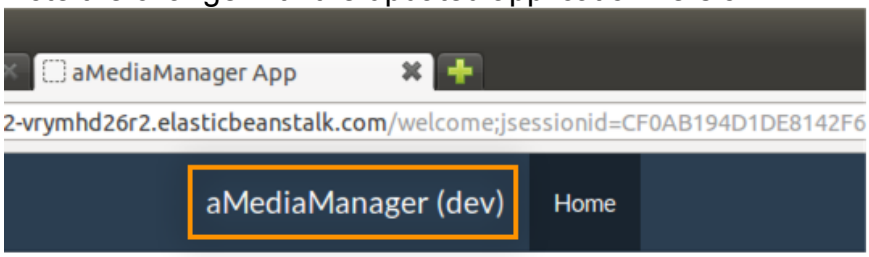
55. Open your Elastic Beanstalk environment in a web browser to view its status, which should be Updating while the new version is deployed from Eclipse:



56. Once your environment is Green, notice the new application version that is running, and click the environment URL to view it:



57. Note the change with the updated application version:



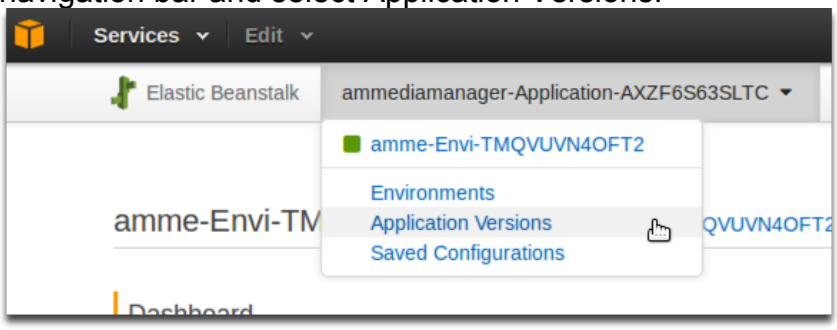
Roll Back to Initial Version

Roll Back to Initial Version

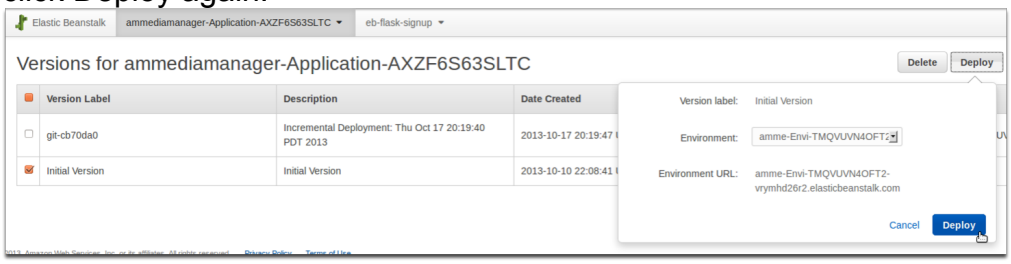
When you deploy your application to Elastic Beanstalk – whether from Eclipse, the Management Console, or directly via the API – a new Application Version is created and stored in an S3 bucket in your account. Application Versions are retained until you delete them, and Elastic Beanstalk allows you to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

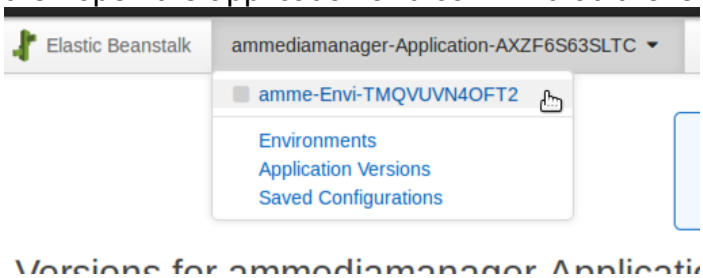
58. In the Elastic Beanstalk Management Console, click your application in the navigation bar and select Application Versions:



59. Choose the Initial Version, click Deploy, choose your Environment and click Deploy again:



60. 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:

```
package com.amediamanager.dao.challenge;

import org.springframework.stereotype.Component;

@Component("dynamoDbUserDaoImplChallenge")
public class DynamoDbUserDaoImpl extends com.amediamanager.dao.DynamoDbUserDaoImpl {

    @Override
    public void save(User user) throws UserExistsException {
        super.save(user);
    }
```

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.challenge` – 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)

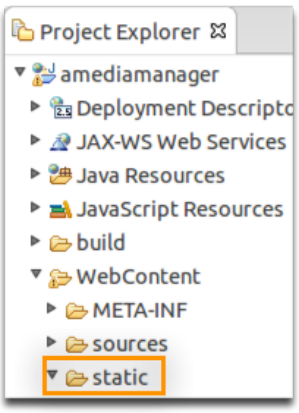
`com.amediamanager.service.challenge` – Coding challenges for the service layer

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

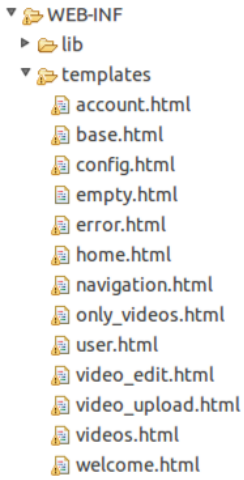
`com.amediamanager.util` – You gotta have a util package, right? These helper classes provide general helper functionality and won't be a focus of any labs

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 are 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

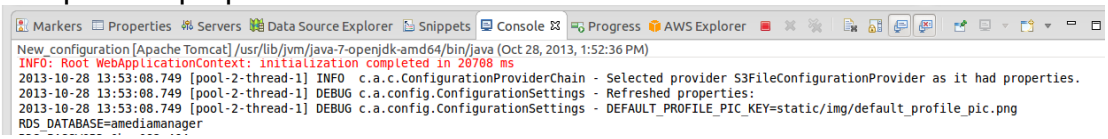
`super.getBucket()`
`super.getKey()`
`super.setProperties()`

Resources

The AWS SDK for Java JavaDoc
The `AmazonS3` interface
The `AmazonS3Client` class

Is It Working?

A working implementation won't throw exceptions, and will log a message to the Console indicating that a S3FileConfigurationProvider was selected, along with a dump of the properties loaded to S3:



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 your app config was pushed to S3, you're in the right place.

First, you may recall that CloudFormation provisioned many of the resources our app needed (e.g., S3 bucket, DynamoDB table, etc). CloudFormation also launched our Elastic Beanstalk environment. Take a look at the CloudFormation documentation – focusing on the OptionSettings piece – to see how CloudFormation provided the names of those resources as environment variables to your Elastic Beanstalk environment.

Next, learn how you can use configuration files to execute commands on your EB instances by reading this section of the documentation.

Finally, in Eclipse open **WebContent > WEB-INF > .ebextensions > 01_app_config.config** and **deploy_config.py** to see how we used the **`container_commands`** convention to push the environment vars to S3.

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.dynamoClient`
`super.config`
`super.HASH_KEY_NAME`
`super.getUserFromMap`
`super.getMapFromUser`
`super.LOG`

Resources

The AWS SDK for Java
JavaDoc
The AmazonDynamoDB Interface

Is It Working?

If you're able to register a new user and sign in, it's working. If not, the console log should have some information about what went wrong.

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 be invoked when Tomcat starts or restarts, which means you can dynamically add or remove Read Replicas to scale the application's database read performance without a code or config push. Check out **`com.amediamanager.dao.challenge.RdsDriverManagerDataSource`** to see how your implementation is injected and used.

If you finish the challenge with time to spare, the Detail Detour at the end of the section will provide some guidance on testing RDS Read Replicas with your app.

The Challenge

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

Important Considerations

fewf

Super Powers

`super.rds`
`super.config`
`super.LOG`

Resources

The AWS SDK for Java
JavaDoc
The AmazonRDS Interface

Is It Working?

Upload a new video with some tags. Since video metadata and tags are stored in RDS, if this works you know you're golden. Also, if your code is successfully locating and returning endpoints for RDS, you should see console output

indicating as much:

```
RDS_USERNAME=admin
S3_UPLOAD_BUCKET=amediamanager-appbucket-1rmix0nl9bz0k
TRANSCODE_ROLE=arn:aws:iam::960309676616:role/amedia...
DEFAULT_VIDEO_POSTER_KEY=static/img/default_poster.png
TRANSCODE_PIPELINE=1382334838368-kbash7
2013-10-28 14:32:31.192 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.config.ConfigurationSettings - ---------------------
2013-10-28 14:32:31.192 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.config.ConfigurationSettings - Effective AWS credential config:
2013-10-28 14:32:31.192 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.config.ConfigurationSettings - Access Key=AKIAIHQPROGYKOFGFNVA
2013-10-28 14:32:32.651 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.dao.RdsDriverManagerDataSource - Detected RDS Master database
Oct 28, 2013 2:32:32 PM org.springframework.jdbc.datasource.DriverManagerDataSource setDriverClassName
INFO: Loaded JDBC driver: com.mysql.jdbc.Driver
2013-10-28 14:32:32.654 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.dao.RdsDriverManagerDataSource - No Read Replicas detected
2013-10-28 14:32:32.679 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] DEBUG org.jboss.logging - Logging Provider: org.jboss.logging.Slf4jLoggerProvider
2013-10-28 14:32:34.488 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.dao.RdsDriverManagerDataSource - Detected RDS Master database
Oct 28, 2013 2:32:34 PM org.springframework.jdbc.datasource.DriverManagerDataSource setDriverClassName
INFO: Loaded JDBC driver: com.mysql.jdbc.Driver
2013-10-28 14:32:34.488 [ContainerBackgroundProcessor[StandardEngine[Catalina]]] INFO  c.a.dao.RdsDriverManagerDataSource - No Read Replicas detected
Oct 28, 2013 2:32:34 PM org.springframework.web.servlet.handler.AbstractUrlHandlerMapping registerHandler
INFO: Mapped URL path [/static/**] onto handler 'org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0'
Oct 28, 2013 2:32:34 PM org.springframework.web.servlet.handler.AbstractUrlHandlerMapping registerHandler
```

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.

Detail Detour

The code you've written will detect RDS Read Replicas if they exist. Open the RDS Management Console, locate your database, and provision a Read Replica. When the RR is ready, restart Tomcat and confirm that the Read Replica was detected.

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 images in DynamoDB when we have 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.
- Don't forget to set the content type of the object!

Super Powers

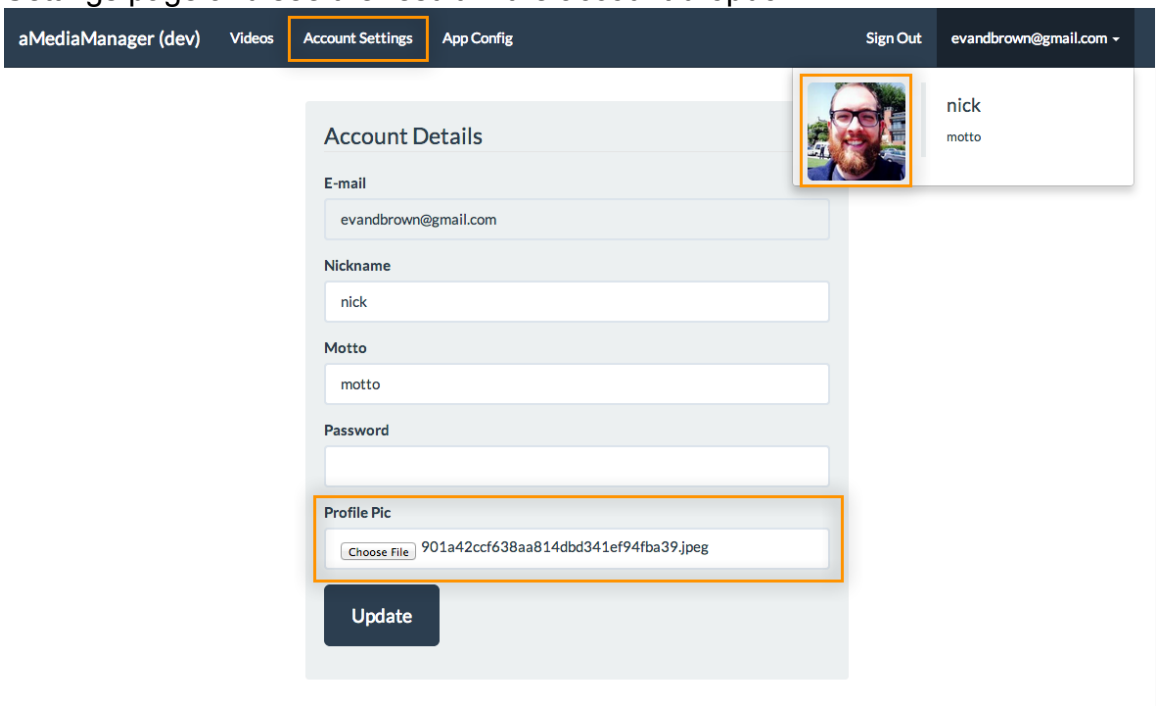
`super.config`
`super.s3Client`
`super.LOG`

Resources

The AWS SDK for Java JavaDoc
The **`AmazonS3`** Interface
`CommonsMultipartFile` documentation

Is It Working?

If your code is working, you should be able to upload a profile pic in the Account Settings page and see the result in the account dropdown:



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 async 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 status messages to a Simple Notification Service Topic, which in turn places those messages 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

- Queue URL is at **`config.getProperty(ConfigProps.TRANSCODE_QUEUE)`**
- Your **`checkStatus()`** implementation must call **`super.handleMessage()`** for each message it receives.

Super Powers

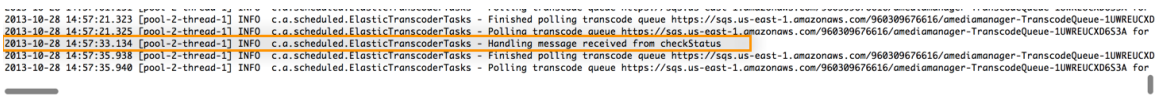
`super.config`
`super.sqsClient`
`super.LOG`

Resources

The AWS SDK for Java JavaDoc

Is It Working?

The handler function will log output when a message is received. Consider using super.LOG to output when you've connected to and received messages from SQS in your implementation.



Finally, upload a few videos to your application and confirm that they are converted.

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.

Detail Detour

Pay special attention to the HTML form that uploads the video from the user's browser. Where does the form POST to? Find the service documentation that describes how to implement this feature, then look at the **`video_upload.html`** template and inspect **`com.amediamanager.controller.VideoController.videoUpload()`** to see how the form is prepared.

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 fail (e.g., trying to upload a user profile pic to a bucket that doesn't exist). There are 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 everywhere 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 that 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

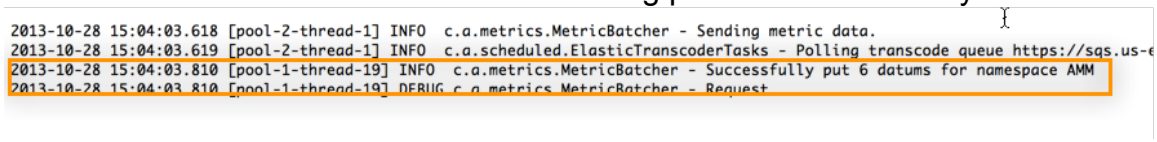
```
super.newDatum()
super.metricBatcher()
```

Resources

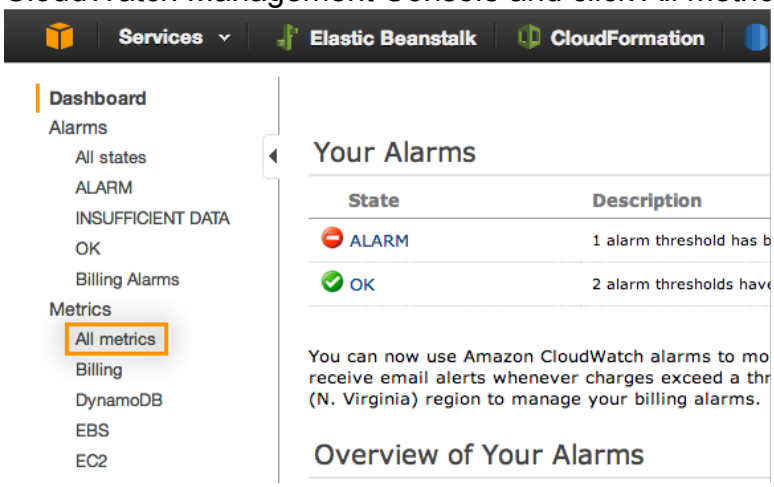
[The AWS SDK for Java JavaDoc](#)
The com.amazonaws.services.cloudwatch.model.MetricDatum documentation

Is It Working?

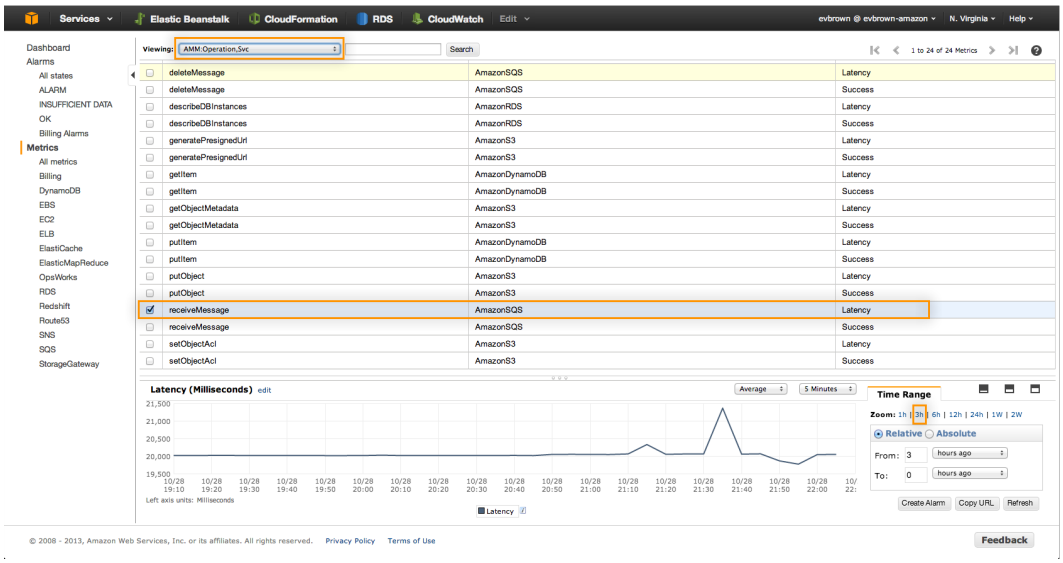
The batcher you emitted metrics to (**`com.amediamanager.metrics.MetricBatcher`**) pushes those metrics to CloudWatch every 60 seconds and logs out how many metrics were put. Check the console and confirm that metrics are being put to CloudWatch by the batcher:



Finally, let's look at the average request latency for calls to S3. Open the CloudWatch Management Console and click All metrics:



Choose AMM:Operation,Svc in the dropdown, then select the receiveMessage metric for AmazonSQS latency. Adjust the graph to view the 5-minute Average for the last 3 hours:



20,000ms seems like a lot, but this is actually a good thing. SQS allows us to "long poll" for up to 20 seconds per HTTP request (i.e., if we make a request and there is no message there, the connection can stay open for up to 20 seconds). This graph verifies we're keeping connections open and taking advantage of this efficiency.

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!