learn some programing related terms

**public** **interface** FunctionHandler<T, R> **extends** HierarchyMatch {

**public** R execute(ExecutionContext eCtx, Param<T> actionParameter);

}

**public** **interface** ExpressionEvaluator {

**public** Object getValue(String exprValue, Object rootObject);

**public** <T> T getValue(String exprValue, Object rootObject, Class<T> returnType);

}

**public** **class** SpelExpressionEvaluator **implements** ExpressionEvaluator {

@Override

**public** Object getValue(String exprValue, Object rootObject) {

**return** getValue(exprValue, rootObject, (expr, ctx)->expr.getValue(ctx));

}

@Override

**public** <T> T getValue(String exprValue, Object rootObject, Class<T> returnType) {

**return** getValue(exprValue, rootObject, (expr, ctx)->expr.getValue(ctx, returnType));

}

**private** <R> R getValue(String exprValue, Object rootObject, BiFunction<Expression, StandardEvaluationContext, R> fn) {

StandardEvaluationContext context = **new** StandardEvaluationContext(rootObject);

//==SpelParserConfiguration config = new SpelParserConfiguration(true, true);

ExpressionParser expressionParser = **new** SpelExpressionParser();

Expression expression = expressionParser.parseExpression(exprValue);

R response = fn.apply(expression, context);

**return** response;

}

}

@Data

**public** **class** EntityAssociation **implements** Serializable {

**private** **static** **final** **long** serialVersionUID = 1L;

String domainAlias;

List<Criteria> criteria;

List<EntityAssociation> associatedEntities;

String associationFrom;

String associationTo;

String associationStartWith;

String projectionFields;

String associationAlias;

**boolean** unwind;

@Data

**public** **static** **class** Criteria **implements** Serializable {

**private** **static** **final** **long** serialVersionUID = 1L;

**private** String key;

**private** String value;

}

}

Bvn:

\* Base class for all domain model.

\* Implements {@link Persistable} for the audit fields {@link CreatedBy} and {@link CreatedDate} since the {@link Id} field is manually assigned.

\*

--

@Model

@Getter

**public** **abstract** **class** AbstractEntity<ID **extends** Serializable> **implements** Serializable, Persistable<ID> {

**private** **static** **final** **long** serialVersionUID = 1L;

@Ignore

**private** **final** String \_class = **this**.getClass().getName();

**public** **static** **abstract** **class** IdLong **extends** AbstractEntity<Long> {

**private** **static** **final** **long** serialVersionUID = 1L;

@Id @Getter @Setter

**private** Long id;

}

**public** **static** **abstract** **class** IdString **extends** AbstractEntity<String> {

**private** **static** **final** **long** serialVersionUID = 1L;

@Id @Getter @Setter @StartsWith

**private** String id;

}

**public** **abstract** ID getId();

@Setter @CreatedBy

**private** String createdBy;

@Setter @CreatedDate

**private** ZonedDateTime createdDate;

@Setter @LastModifiedBy

**private** String lastModifiedBy;

@Setter @LastModifiedDate

**private** ZonedDateTime lastModifiedDate;

//@Version

@Setter **private** **long** version;

@JsonIgnore

**public** <T **extends** AbstractEntityBehavior<M, ID>, M **extends** AbstractEntity<ID>> T newBehaviorInstance(Class<T> clazz) {

**try** {

T t = ConstructorUtils.invokeExactConstructor(clazz, **this**);

**return** t;

}

**catch** (Exception ex) {

**throw** **new** FrameworkRuntimeException("Failed to instantiate class of type: " + clazz, ex);

}

}

@JsonIgnore

@Override

**public** **boolean** isNew() {

**return** **this**.lastModifiedDate == **null**;

}

}

function handlers:

<https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html>

A handler is a routine/function/method which is specialized in a certain type of data or focused on certain special tasks.

Examples:

* Event handler - Receives and digests events and signals from the surrounding system (e.g. OS or GUI).
* Memory handler - Performs certain special tasks on memory.
* File input handler - A function receiving file input and performing special tasks on the data, all depending on context of course.

--

# Lambda Function Handler (Node.js)

At the time you create a Lambda function you specify a handler, a function in your code, that AWS Lambda can invoke when the service executes your code. Use the following general syntax when creating a handler function in Node.js.

exports.*myHandler* = function(event, context) {

...

}

The callback parameter is optional, depending on whether you want to return information to the caller.

exports.*myHandler* = function(event, context, callback) {

...

// Use callback() and return information to the caller.

}

In the syntax, note the following:

* event – AWS Lambda uses this parameter to pass in event data to the handler.
* context – AWS Lambda uses this parameter to provide your handler the runtime information of the Lambda function that is executing. For more information, see [The Context Object (Node.js)](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-context.html).
* callback – You can use the optional callback to return information to the caller, otherwise return value is null. For more information, see [Using the Callback Parameter](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html#nodejs-prog-model-handler-callback).

**Note**

The callback is supported only in the Node.js runtimes v6.10 and v4.3. If you are using runtime v0.10.42, you need to use the context methods (done, succeed, and fail) to properly terminate the Lambda function. For information, see [Using the Earlier Node.js Runtime v0.10.42](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-using-old-runtime.html).

* *myHandler* – This is the name of the function AWS Lambda invokes. You export this so it is visible to AWS Lambda. Suppose you save this code as helloworld.js. Then,helloworld.myHandler is the handler. For more information, see handler in [CreateFunction](https://docs.aws.amazon.com/lambda/latest/dg/API_CreateFunction.html).
  + If you used the RequestResponse invocation type (synchronous execution), AWS Lambda returns the result of the Node.js function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the RequestResponse invocation type, so when you test invoke the function using the console, the console will display the return value.

If the handler does not return anything, AWS Lambda returns null.

* + If you used the Event invocation type (asynchronous execution), the value is discarded.

## Example

Consider the following Node.js example code.

exports.*myHandler* = function(event, context, callback) {

console.log("value1 = " + event.key1);

console.log("value2 = " + event.key2);

callback(null, "some success message");

// or

// callback("some error type");

}

This example has one function, which is also the handler. In the function, the console.log()statements log some of the incoming event data to CloudWatch Logs. When the callback is called, the Lambda function exits only after the Node.js event loop is empty (the Node.js event loop is not the same as the event that was passed as a parameter).

**Note**

If you are using runtime v0.10.42, you need to use the context methods (done, succeed, and fail) to properly terminate the Lambda function. For more information, see [Using the Earlier Node.js Runtime v0.10.42](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-using-old-runtime.html).

**To upload and test this code as a Lambda function (console)**

1. In the console, create a Lambda function using the following information:
   * Use the hello-world blueprint.
   * We recommend specifying **nodejs6.10** as the **runtime** but you can also select **nodejs4.3**. The code samples provided will work for either version.

For instructions to create a Lambda function using the console, see [Create a Simple Lambda Function](https://docs.aws.amazon.com/lambda/latest/dg/get-started-create-function.html).

1. Replace the template code with the code provided in this section and create the function.
2. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console.

## Using the Callback Parameter

The Node.js runtimes v4.3 and v6.10 support the optional callback parameter. You can use it to explicitly return information back to the caller. The general syntax is:

callback(Error error, Object result);

Where:

* error – is an optional parameter that you can use to provide results of the failed Lambda function execution. When a Lambda function succeeds, you can pass null as the first parameter.
* result – is an optional parameter that you can use to provide the result of a successful function execution. The result provided must be JSON.stringify compatible. If an error is provided, this parameter is ignored.

**Note**

Using the callback parameter is optional. If you don't use the optional callbackparameter, the behavior is same as if you called the callback() without any parameters. You can specify the callback in your code to return information to the caller.

If you don't use callback in your code, AWS Lambda will call it implicitly and the return value is null.

When the callback is called (explicitly or implicitly), AWS Lambda continues the Lambda function invocation until the Node.js event loop is empty.

The following are example callbacks:

callback(); // Indicates success but no information returned to the caller.

callback(null); // Indicates success but no information returned to the caller.

callback(null, "success"); // Indicates success with information returned to the caller.

callback(error); // Indicates error with error information returned to the caller.

AWS Lambda treats any non-null value for the error parameter as a handled exception.

Note the following:

* Regardless of the invocation type specified at the time of the Lambda function invocation (see [Invoke](https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html)), the callback method automatically logs the string representation of non-null values of error to the Amazon CloudWatch Logs stream associated with the Lambda function.
* If the Lambda function was invoked synchronously (using the RequestResponse invocation type), the callback returns a response body as follows:
  + If error is null, the response body is set to the string representation of result.
  + If the error is not null, the error value will be populated in the response body.

**Note**

When the callback(error, null) (and callback(error)) is called, Lambda will log the first 256 KB of the error object. For a larger error object, AWS Lambda truncates the log and displays the text Truncated by Lambda next to the error object.

--

# Lambda Function Handler (Java)

At the time you create a Lambda function you specify a handler that AWS Lambda can invoke when the service executes the Lambda function on your behalf.

Lambda supports two approaches for creating a handler:

* Loading the handler method directly without having to implement an interface. This section describes this approach.
* Implementing standard interfaces provided as part of aws-lambda-java-core library (interface approach). For more information, see [Leveraging Predefined Interfaces for Creating Handler (Java)](https://docs.aws.amazon.com/lambda/latest/dg/java-handler-using-predefined-interfaces.html).

The general syntax for the handler is as follows:

*outputType* *handler-name*(*inputType* input, Context context) {

...

}

In order for AWS Lambda to successfully invoke a handler it must be invoked with input data that can be serialized into the data type of the input parameter.

In the syntax, note the following:

* *inputType* – The first handler parameter is the input to the handler, which can be event data (published by an event source) or custom input that you provide such as a string or any custom data object. In order for AWS Lambda to successfully invoke this handler, the function must be invoked with input data that can be serialized into the data type of the input parameter.
* *outputType* – If you plan to invoke the Lambda function synchronously (using the RequestResponse invocation type), you can return the output of your function using any of the supported data types. For example, if you use a Lambda function as a mobile application backend, you are invoking it synchronously. Your output data type will be serialized into JSON.

If you plan to invoke the Lambda function asynchronously (using the Event invocation type), the outputType should be void. For example, if you use AWS Lambda with event sources such as Amazon S3 or Amazon SNS, these event sources invoke the Lambda function using the Event invocation type.

* The *inputType* and *outputType* can be one of the following:
  + Primitive Java types (such as String or int).
  + Predefined AWS event types defined in the aws-lambda-java-events library.

For example S3Event is one of the POJOs predefined in the library that provides methods for you to easily read information from the incoming Amazon S3 event.

* + You can also write your own POJO class. AWS Lambda will automatically serialize and deserialize input and output JSON based on the POJO type.

For more information, see [Handler Input/Output Types (Java)](https://docs.aws.amazon.com/lambda/latest/dg/java-programming-model-req-resp.html).

* You can omit the Context object from the handler method signature if it isn't needed. For more information, see [The Context Object (Java)](https://docs.aws.amazon.com/lambda/latest/dg/java-context-object.html).

For example, consider the following Java example code.

package example;

import com.amazonaws.services.lambda.runtime.Context;

import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Hello implements RequestHandler<Integer, String>{

public String myHandler(int myCount, Context context) {

return String.valueOf(myCount);

}

}

In this example input is of type Integer and output is of type String. If you package this code and dependencies, and create your Lambda function, you specify example.Hello::myHandler(*package*.*class*::*method-reference*) as the handler.

In the example Java code, the first handler parameter is the input to the handler (myHandler), which can be event data (published by an event source such as Amazon S3) or custom input you provide such as an Integer object (as in this example) or any custom data object.

For instructions to create a Lambda function using this Java code, see [(Optional) Create a Lambda Function Authored in Java](https://docs.aws.amazon.com/lambda/latest/dg/get-started-step4-optional.html).

## Handler Overload Resolution

If your Java code contains multiple methods with same name as the handler name, then AWS Lambda uses the following rules to pick a method to invoke:

1. Select the method with the largest number of parameters.
2. If two or more methods have the same number of parameters, AWS Lambda selects the method that has the Context as the last parameter.

If none or all of these methods have the Context parameter, then the behavior is undefined.

## Additional Information

The following topics provide more information about the handler.

* For more information about the handler input and output types, see [Handler Input/Output Types (Java)](https://docs.aws.amazon.com/lambda/latest/dg/java-programming-model-req-resp.html).
* For information about using predefined interfaces to create a handler, see [Leveraging Predefined Interfaces for Creating Handler (Java)](https://docs.aws.amazon.com/lambda/latest/dg/java-handler-using-predefined-interfaces.html).

If you implement these interfaces, you can validate your handler method signature at compile time.

**aws-lambda-java-core** – This library provides the Context object, RequestStreamHandler, and the RequestHandler interfaces. The Context object ([The Context Object (Java)](https://docs.aws.amazon.com/lambda/latest/dg/java-context-object.html)) provides runtime information about your Lambda function. The predefined interfaces provide one way of defining your Lambda function handler.

**aws-lambda-java-events** – This library provides predefined types that you can use when writing Lambda functions to process events published by Amazon.. These classes help you process the event without having to write your own custom serialization logic.

**Custom Appender for Log4j2.8** – You can use the custom Log4j (see [Apache Log4j 2](http://logging.apache.org/log4j/2.x)) appender provided by AWS Lambda for logging from your lambda functions. Every call to Log4j methods, such as log.debug() or log.error(), will result in a CloudWatch Logs event. The custom appender is called LambdaAppender and must be used in the log4j2.xml file. You must include the aws-lambda-java-log4j2 artifact (artifactId:aws-lambda-java-log4j2) in the deployment package (.jar file).

# Lambda Function Handler (Node.js)

At the time you create a Lambda function you specify a handler, a function in your code, that AWS Lambda can invoke when the service executes your code. Use the following general syntax when creating a handler function in Node.js.

exports.*myHandler* = function(event, context) {

...

}

The callback parameter is optional, depending on whether you want to return information to the caller.

exports.*myHandler* = function(event, context, callback) {

...

// Use callback() and return information to the caller.

}

In the syntax, note the following:

* event – AWS Lambda uses this parameter to pass in event data to the handler.
* context – AWS Lambda uses this parameter to provide your handler the runtime information of the Lambda function that is executing. For more information, see [The Context Object (Node.js)](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-context.html).
* callback – You can use the optional callback to return information to the caller, otherwise return value is null. For more information, see [Using the Callback Parameter](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html#nodejs-prog-model-handler-callback).

**Note**

The callback is supported only in the Node.js runtimes v6.10 and v4.3. If you are using runtime v0.10.42, you need to use the context methods (done, succeed, and fail) to properly terminate the Lambda function. For information, see [Using the Earlier Node.js Runtime v0.10.42](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-using-old-runtime.html).

* *myHandler* – This is the name of the function AWS Lambda invokes. You export this so it is visible to AWS Lambda. Suppose you save this code as helloworld.js. Then,helloworld.myHandler is the handler. For more information, see handler in [CreateFunction](https://docs.aws.amazon.com/lambda/latest/dg/API_CreateFunction.html).
  + If you used the RequestResponse invocation type (synchronous execution), AWS Lambda returns the result of the Node.js function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the RequestResponse invocation type, so when you test invoke the function using the console, the console will display the return value.

If the handler does not return anything, AWS Lambda returns null.

* + If you used the Event invocation type (asynchronous execution), the value is discarded.

## Example

Consider the following Node.js example code.

exports.*myHandler* = function(event, context, callback) {

console.log("value1 = " + event.key1);

console.log("value2 = " + event.key2);

callback(null, "some success message");

// or

// callback("some error type");

}

This example has one function, which is also the handler. In the function, the console.log()statements log some of the incoming event data to CloudWatch Logs. When the callback is called, the Lambda function exits only after the Node.js event loop is empty (the Node.js event loop is not the same as the event that was passed as a parameter).

**Note**

If you are using runtime v0.10.42, you need to use the context methods (done, succeed, and fail) to properly terminate the Lambda function. For more information, see [Using the Earlier Node.js Runtime v0.10.42](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-using-old-runtime.html).

**To upload and test this code as a Lambda function (console)**

1. In the console, create a Lambda function using the following information:
   * Use the hello-world blueprint.
   * We recommend specifying **nodejs6.10** as the **runtime** but you can also select **nodejs4.3**. The code samples provided will work for either version.

For instructions to create a Lambda function using the console, see [Create a Simple Lambda Function](https://docs.aws.amazon.com/lambda/latest/dg/get-started-create-function.html).

1. Replace the template code with the code provided in this section and create the function.
2. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console.

## Using the Callback Parameter

The Node.js runtimes v4.3 and v6.10 support the optional callback parameter. You can use it to explicitly return information back to the caller. The general syntax is:

callback(Error error, Object result);

Where:

* error – is an optional parameter that you can use to provide results of the failed Lambda function execution. When a Lambda function succeeds, you can pass null as the first parameter.
* result – is an optional parameter that you can use to provide the result of a successful function execution. The result provided must be JSON.stringify compatible. If an error is provided, this parameter is ignored.

**Note**

Using the callback parameter is optional. If you don't use the optional callbackparameter, the behavior is same as if you called the callback() without any parameters. You can specify the callback in your code to return information to the caller.

If you don't use callback in your code, AWS Lambda will call it implicitly and the return value is null.

When the callback is called (explicitly or implicitly), AWS Lambda continues the Lambda function invocation until the Node.js event loop is empty.

The following are example callbacks:

callback(); // Indicates success but no information returned to the caller.

callback(null); // Indicates success but no information returned to the caller.

callback(null, "success"); // Indicates success with information returned to the caller.

callback(error); // Indicates error with error information returned to the caller.

AWS Lambda treats any non-null value for the error parameter as a handled exception.

Note the following:

* Regardless of the invocation type specified at the time of the Lambda function invocation (see [Invoke](https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html)), the callback method automatically logs the string representation of non-null values of error to the Amazon CloudWatch Logs stream associated with the Lambda function.
* If the Lambda function was invoked synchronously (using the RequestResponse invocation type), the callback returns a response body as follows:
  + If error is null, the response body is set to the string representation of result.
  + If the error is not null, the error value will be populated in the response body.

**Note**

When the callback(error, null) (and callback(error)) is called, Lambda will log the first 256 KB of the error object. For a larger error object, AWS Lambda truncates the log and displays the text Truncated by Lambda next to the error object.

--

@Configuration

**public** **class** DefaultProcessConfig {

@Bean

**public** ActivitiExpressionManager activitiExpressionManager(){

**return** **new** ActivitiExpressionManager();

}

@Bean

**public** BPMGateway bpmGateway(BeanResolverStrategy beanResolver){

**return** **new** ActivitiBPMGateway(beanResolver);

}

@Bean(name="default.\_new$execute?fn=\_initEntity")

**public** FunctionHandler<?, ?> defaultActionNewInitFunctionHandler(BeanResolverStrategy beanResolver)

{

**return** **new** DefaultActionNewInitEntityFunctionHandler<>(beanResolver);

}

@Bean(name="default.\_get$execute?fn=param")

**public** FunctionHandler<?, ?> defaultParamFunctionHandler(BeanResolverStrategy beanResolver){

**return** **new** DefaultParamFunctionHandler<>(beanResolver);

}

@Bean(name="default.\_nav$execute?fn=default")

**public** PageIdEchoNavHandler<?> pageIdEchoNavHandler(){

**return** **new** PageIdEchoNavHandler<>();

}

@Bean(name="default.\_process$execute?fn=\_set")

**public** SetFunctionHandler<?,?> setFunctionHandler(){

**return** **new** SetFunctionHandler<>();

}

@Bean(name="default.\_process$execute?fn=\_update")

**public** UpdateFunctionHandler<?,?> updateFunctionHandler(){

**return** **new** UpdateFunctionHandler<>();

}

@Bean(name="default.\_process$execute?fn=\_setByRule")

**public** FunctionHandler<?,?> setByRuleFunctionHandler(){

**return** **new** SetByRuleFunctionalHandler<>();

}

@Bean(name="default.\_process$execute?fn=\_add")

**public** AddFunctionHandler<?,?> addFunctionHandler(){

**return** **new** AddFunctionHandler<>();

}

@Bean(name="default.\_process$execute?fn=\_bpm")

**public** StatelessBPMFunctionHanlder<?,?> statelessBPMFunctionHanlder(BeanResolverStrategy beanResolver){

**return** **new** StatelessBPMFunctionHanlder<>(beanResolver);

}

@Bean(name="expressionEvaluator")

**public** ExpressionEvaluator expressionEvaluator(BeanResolverStrategy beanResolver){

**return** **new** SpelExpressionEvaluator();

}

@Bean(name="commandExecutorTaskDelegate")

**public** CommandExecutorTaskDelegate commandExecutorTaskDelegate(BeanResolverStrategy beanResolver){

**return** **new** CommandExecutorTaskDelegate(beanResolver);

}

@Bean(name="default.\_search$execute?fn=lookup")

**public** FunctionHandler<?, ?> lookupFunctionHandler(){

**return** **new** DefaultSearchFunctionHandlerLookup<>();

}

@Bean(name="default.\_search$execute?fn=example")

**public** FunctionHandler<?, ?> exampleFunctionHandler(){

**return** **new** DefaultSearchFunctionHandlerExample<>();

}

@Bean(name="default.\_search$execute?fn=query")

**public** FunctionHandler<?, ?> queryFunctionHandler(){

**return** **new** DefaultSearchFunctionHandlerQuery<>();

}

@Bean(name="default.\_process$execute?fn=\_eval")

**public** EvalFunctionHandler<?,?> evalFunctionHandler(ExpressionManager expressionManager){

**return** **new** EvalFunctionHandler(expressionManager);

}

}

default.\_process$execute?fn=\_eval"