2017

Practical Application of Java 8 and Lambdas within Alfresco

KEITH BATON

DISCOVERTECHNOLOGIES.COM | 13241 WOODLAND PARK ROAD, SUITE 610 | HERNDON, VA 20171 | 800.368.2344
INFO@DISCOVERTECHNOLOGIES.COM

Contents

Behaviors and Observer Pattern 3

Figure 1. An excerpt of the Alfresco documentation version of custom behaviors 3

Figure 2. A simple interface that will be implemented by a custom event 4

Figure 3. An event handler that will serve as the observer for all events 4

Figure 4. An example of instantiating an event handler bean with events 5

Figure 5. An implementation of a dynamic event 5

Anonymous inner classes versus Lambda 6

Table 1. Lambda Advantages and Disadvantages 6

Figure 6. Implementation of creating a folder using anonymous inner classes 7

Figure 7. Implementation of creating a folder using lambda instead of anonymous inner classes 8

Figure 8. An alternative implementation of creating a folder using JavaScript-like lambda nesting 8

Lambda Composition and Streams 9

Figure 9. A library of lambda functions and stream methods 9

Figure 10. Advanced stream-based filtering mechanism 10

Figure 11. Find, filter and enrich nodes with metadata using advanced streams. 10

Conclusion 11

Contact us 11

Coming Soon 11

References 11

Java 8 is the most recent major update to the Java platform. Alfresco supports Java 8 as of version 5.0.1 (Alfresco Software Inc., 2017). Java 8 includes many appealing new language features that could ease development of Alfresco extensions for both now and future development efforts. Java 8 introduces multiple new language features:

- New date and time API
- Lambda expressions
- Optional objects
- Simplified concurrency constructs

These new language features were generally well-received by the Java community (Wolf, 2016). Now that Java 9 is on the horizon; Discover Technologies engineers feel that this is a good time to catch developers up on the Java 8 technology and its use within Alfresco.

Discover Technologies developers have found many Java 8 features to be extremely useful for developing Alfresco extensions. The use of lambdas within the context of the Alfresco API have enabled us to provide some interesting and scalable extensions, especially when dealing with large datasets. The overall readability of our codebase also improves as we refactor certain structures to utilize lambdas. The JDK has bytecode optimizations which allow lambda functions to perform similarly to the non-lambda implementations they are created to mirror. This whitepaper focuses on the practical application of Java 8 lambdas within the scope of Alfresco Platform Extensions and Java API.

Behaviors and Observer Pattern

When developing extensions, the need arises to create a behavior to provide event-based execution of business logic. Often, it is tempting to follow the documentation directly to construct a behavior. Below is an example of the kind of behavior that is generated when using only the documentation as a guide:

```
public class DocumentEventHandler {
    private static Logger logger = LoggerFactory.getLogger(DocumentEventHandler.class);

    private PolicyComponent eventManager;
    private ServiceRegistry serviceRegistry;

    public void setServiceRegistry(ServiceRegistry serviceRegistry) {
        this.serviceRegistry = serviceRegistry;
    }

    public void setPolicyComponent(PolicyComponent policyComponent) {
        this.eventManager = policyComponent;
    }

    public void registerEventHandlers() {
        eventManager.bindClassBehaviour(
                NodeServicePolicies.OnCreateNodePolicy.QNAME,
                ContentModel.TYPE_CONTENT,
                new JavaBehaviour(this, "onAddDocument",
                        Behaviour.NotificationFrequency.TRANSACTION_COMMIT));
    }

    public void onAddDocument(ChildAssociationRef parentChildAssocRef) {
        NodeRef parentFolderRef = parentChildAssocRef.getParentRef();
        NodeRef docRef = parentChildAssocRef.getChildRef();

        // Check if node exists, might be moved, or created and deleted in same transaction.
        if (docRef == null || !serviceRegistry.getNodeService().exists(docRef)) {
            // Does not exist, nothing to do
            logger.warn("onAddDocument: A new document was added but removed in same transaction");
            return;
        } else {
            logger.info("onAddDocument: A new document with ref ({}) was just created in folder ({})",
                    docRef, parentFolderRef);
        }
    }
}
```

Figure 1: An excerpt of the Alfresco documentation version of custom behaviors

The single-class event handler may be refactored into another form to facilitate a separation of concerns architecture. The basic design for a separation of concerns solution should splinter the event handler from the events themselves. Java 8 lambdas may then be used to facilitate the sharing of business logic throughout the application.

The observer design pattern can be used to refactor Figure 1 into a single-bean event handler which accepts multiple different events of varying types. The design of the observer pattern for event handling is simple, first a core event class should be created that will observe the individual events. Because of the way in which Alfresco has engineered the behavior registration, it would make more sense to invert the common control of this task from the observer to the observed objects themselves. Therefore, each event should be responsible for the registration itself. The solution will need a single common interface that all events will implement and, when passed as an object to the event handler, the events may be registered to the observer's internal tracking.

```
public interface Event<T> {
    BehaviourDefinition<ClassBehaviourBinding> register(PolicyComponent eventManager);
    void swapBehavior(T behavior);
}
```

Figure 2: A simple interface that will be implemented by a custom event.

Another consideration is the fact that Alfresco uses different method signatures depending on the type of behavior, so dynamically changing behavior would be difficult to code generically from the aspect of the event handler. Therefore, the event handler will be barren and only serve to register and retrieve custom behavior events.

```
public class DocumentEventHandler {
    private PolicyComponent eventManager;
    private Map<QName, Event<?>> boundEvents;

    @PostConstruct
    public void registerEvents() {
        for(Event<?> event : boundEvents.values()) {
            event.register(eventManager);
        }
    }

    public <T extends Event> Optional<T> getBoundEvent(QName policyQName, Class<T> eventType) {
        return Optional.ofNullable(eventType.cast(boundEvents.get(policyQName)));
    }

    public void setPolicyComponent(PolicyComponent policyComponent) {
        this.eventManager = policyComponent;
    }

    @Resource
    public void setBoundEvents(Map<QName, Event<?>> boundEvents) {
        this.boundEvents = boundEvents;
    }
}
```

Figure 3. An event handler that will serve as the observer for all events

The developer can then instantiate and place the policy component into the bean itself. Therefore, the bean registration in spring will look like this:

```
<bean id="com.dtech.example 2.documentEventHandler" class="com.dtech.example 2.DocumentEventHandler"
      init-method="registerEvents">
    <property name="policyComponent">
        <ref bean="policyComponent"/>
    </property>
    <property name="boundEvents">
        <map>
            <entry>
                <key>
                    <value type="org.alfresco.service.namespace.QName"> org.alfresco.repo.node.NodeServ-
icePolicies..OnCreateNodePolicy.QNAME</value>
                </key>
                <bean class="com.dtech.example 2.CreationEvent"/>
            </entry>
        </map>
    </property>
</bean>
```

Figure 4. an example of instantiating an event handler bean with events

Finally, a developer may define each event independently of the logic which performs the registration; and then, may add the logic necessary to register with the Alfresco subsystem:

```
public class CreationEvent implements Event<Consumer<ChildAssociationRef>> {
    private static Logger logger = LoggerFactory.getLogger(CreationEvent.class);

    private Consumer<ChildAssociationRef> currentBehavior = parentAssociationRef -> {
        NodeRef parentFolderRef = parentAssociationRef.getParentRef();
        NodeRef docRef = parentAssociationRef.getChildRef();

        logger.info("onCreateNode: A new document with ref ({}) was just created in folder ({})",
                docRef, parentFolderRef);
    };

    @Override
    public BehaviourDefinition<ClassBehaviourBinding> register(PolicyComponent eventManager) {
        return eventManager.bindClassBehaviour(
                NodeServicePolicies.OnCreateNodePolicy.QNAME,
                ContentModel.TYPE_CONTENT,
                new JavaBehaviour(this, "onCreateNode",
                        Behaviour.NotificationFrequency.TRANSACTION_COMMIT));
    }

    public void onCreateNode(ChildAssociationRef parentAssociationRef) {
        currentBehavior.accept(parentAssociationRef);
    }

    @Override
    public void swapBehavior(Consumer<ChildAssociationRef> alternateBehavior) {
        currentBehavior = alternateBehavior;
    }
}
```

Figure 5. An implementation of a dynamic event

Note the lambda object being defined in the **currentBehavior** variable. This enables creation of an anonymous class, but with a type that is globally accepted in different parts of the Java language including the new collection and concurrency frameworks (Get started with the Java Collections Framework, n.d.). For this event, the solution calls for modification to the first example's code to enable dynamic execution of business logic. The first task is to perform Alfresco's required registration using the dummy method **onCreateNode**, which has the signature needed to map to the policy object internally. Next, the solution should execute the lambda function assigned to the **currentBehavior** variable. The **swapBehavior** method, if called before the event is executed, will change out the default behavior for a new behavior at runtime, allowing for dynamic business logic.

Anonymous inner classes versus Lambda

In the Behaviors and Observer Pattern section, it was demonstrated that one could use lambda functions to create a clean and dynamic event handler service. One thing that was not mentioned, was the fact that the entire implementation could have built without using Java 8 features at all. When viewed from a high-level, lambda functions are a "**syntactic sugar**" or alternate method to develop anonymous inner classes. When taking a closer look at the JVM level, there are some striking differences between lambda functions and anonymous inner classes. Here are some advantages and disadvantages to using a lambda versus an anonymous inner class:

The Advantages and Disadvantages of using lambda functions in place of anonymous inner classes

Advantages	Disadvantages
Lambdas can be used to easily make operations involving collections concurrent and robust.	For more advanced requirements in business logic, anonymous inner classes can provide more than one method override. (Oracle, 2016)
Lambda implementations are capable of being compiled at the JVM level more efficiently using JDK 7's invokeDynamic. (Warburton, 2014)	Lambda functions are not capable of being compiled into class files. Loading a lambda function manually is difficult as they are given random names by the JVM at runtime. (Warburton, 2014)
Passing business logic to other methods is straightforward, and can generally be done in a single line.	The state passed into a lambda function is immutable (effectively final), while anonymous inner classes can maintain state using instance variables for the life of the anonymous object. (Oracle, 2016)

Table 1. Lambda Advantages and Disadvantages

One may be asked to develop a solution that requires the use of a transaction and a system user account. Alfresco has given developers the **AuthenticationUtil.RunAsWork** interface for running in a different user context; as well as the **RetryingTransactionHelper.RetryingTransactionHelperCallback** interface for wrapping logic in a transaction. The traditional implementation of which may be viewed here:

```
public void classicInit() {
    // wrap folder creation in a transaction
    RetryingTransactionHelper.RetryingTransactionCallback<Integer> transactionCallback = new RetryingTrans-
actionHelper.RetryingTransactionCallback<Integer>() {
        @Override
        public Integer execute() throws Throwable {
            NodeService nodeService = serviceRegistry.getNodeService();
            NodeLocatorService nodeLocatorService = serviceRegistry.getNodeLocatorService();
            FileFolderService folderService = serviceRegistry.getFileFolderService();
            String name = "Retrying Performer";
            NodeRef companyHome = nodeLocatorService.getNode(CompanyHomeNodeLocator.NAME, null, null);
            NodeRef retryingFolder = folderService.searchSimple(companyHome, name);
            if(retryingFolder == null) {
                Map<QName, Serializable> prop = new HashMap<>();
                prop.put(ContentModel.PROP NAME, name);
                nodeService.createNode(companyHome, ContentModel.ASSOC CONTAINS,
                        QName.createQName(NamespaceService.CONTENT MODEL 1 0 URI+name), Content-
Model.TYPE FOLDER,prop);
            }
            return 0;
        }
    };

    // wrap the transaction in a separate work object
    AuthenticationUtil.RunAsWork<Integer> work = new AuthenticationUtil.RunAsWork<Integer>() {
        @Override
        public Integer doWork() throws Exception {
            try {
                return transactionCallback.execute();
            } catch (Throwable throwable) {
                throwable.printStackTrace();
            }
            return -1;
        }
    };

    // perform the transaction with the given user's permissions
    AuthenticationUtil.runAs(work, authorizedUser);
}
```

Figure 6. Implementation of creating a folder using anonymous inner classes

To show how one could optimize the readability of this code, consider the following, which is the same example programmed with lambda functions instead of anonymous inner classes:

```
public void lambdaInit() {
    // wrap folder creation in a transaction
    RetryingTransactionHelper.RetryingTransactionCallback<Integer> transactionCallback = () -> {
        NodeService nodeService = serviceRegistry.getNodeService();
        NodeLocatorService nodeLocatorService = serviceRegistry.getNodeLocatorService();
        FileFolderService folderService = serviceRegistry.getFileFolderService();
        String name = "Retrying Performer";
        NodeRef companyHome = nodeLocatorService.getNode(CompanyHomeNodeLocator.NAME, null, null);
        NodeRef retryingFolder = folderService.searchSimple(companyHome, name);
        if(retryingFolder == null) {
            Map<QName, Serializable> prop = new HashMap<>();
            prop.put(ContentModel.PROP NAME, name);
            nodeService.createNode(companyHome, ContentModel.ASSOC CONTAINS,
                    QName.createQName(NamespaceService.CONTENT MODEL 1 0 URI+name), Content-
Model.TYPE FOLDER,prop);
        }
        return 0;
    };

    // wrap the transaction in a separate work object
    AuthenticationUtil.RunAsWork<Integer> work = () -> serviceRegistry.getRetryingTransac-
tionHelper().doInTransaction(transactionCallback);

    // perform the transaction with the given user's permissions
    AuthenticationUtil.runAs(work , authorizedUser);
}
```

Figure 7. Implementation of creating a folder using lambda instead of anonymous inner classes

Alternatively, one can nest the lambda calls into a JavaScript like functional structure:

```
public void alternativeInit() {
    // wrap folder creation in a transaction
    // wrap the transaction in a separate work object
    // perform the transaction with the given user's permissions
    AuthenticationUtil.runAs( () -> serviceRegistry.getRetryingTransactionHelper().doInTransaction( () -> {
        NodeService nodeService = serviceRegistry.getNodeService();
        NodeLocatorService nodeLocatorService = serviceRegistry.getNodeLocatorService();
        FileFolderService folderService = serviceRegistry.getFileFolderService();
        String name = "Retrying Performer";
        NodeRef companyHome = nodeLocatorService.getNode(CompanyHomeNodeLocator.NAME, null, null);
        NodeRef retryingFolder = folderService.searchSimple(companyHome, name);
        if(retryingFolder == null) {
            Map<QName, Serializable> prop = new HashMap<>();
            prop.put(ContentModel.PROP NAME, name);
            nodeService.createNode(companyHome, ContentModel.ASSOC CONTAINS,
                    QName.createQName(NamespaceService.CONTENT MODEL 1 0 URI+name), Content-
Model.TYPE FOLDER,prop);
        }
        return 0;
    }), authorizedUser);
}
```

Figure 8. An alternative implementation of creating a folder using JavaScript-like lambda nesting

With the introduction of lambda functions, Alfresco Platform Extension points are a bit more flexible in terms of readability. Several API in the Alfresco API utilize single-method anonymous inner classes for implementation of logic. These interfaces are forward-compatible with Java 8 lambda functions and make the code less complicated to read at first glance.

Lambda Composition and Streams

Lambda functions may be stored inside objects in the same way that anonymous inner classes can. This enables lambdas to be passed as parameter values to methods for dynamic execution of business logic; see the first study in the Behaviors and Observer Pattern section of this paper.

Alfresco provides a search service API for performing node searching using FTS_ALFRESCO query language. The syntax and API are different to what is provided using Java 8. Consider the example from the Alfresco documentation for searching using the search service (Alfresco Software Inc., n.d.). This service can be emulated using the Java 8 stream interface with the lambda functions.

Streams are **lazily evaluated** and can be used to provide an easy way to create complex solutions with a simplistic syntax. The Java language API for streams provides an example of laziness:

"…executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed. " (Oracle, 2016)

Developers using Alfresco Java API with Java 8 Streams can quickly piece together solutions which work well for both small and large datasets thanks to the lazy evaluation of Streams. For instance, take the common need to search a tree for nodes with any content, type, property or aspect. One could accomplish this using several search-service-specific queries. Alfresco developers create functions which check node state and perform simple tasks on nodes quite often. With lambda functions, one may maintain a nice collection of these common tasks in a single utility bean:

```
public class NodeRefLambdaLib {
    ...

    Predicate<NodeRef> hasEmailAspect = node -> serviceRegistry.getNodeService()
            .hasAspect(node, ContentModel.ASPECT_EMAILED);

    Predicate<NodeRef> hasEmailPropertyValue = node -> serviceRegistry.getNodeService()
            .getProperty(node, ContentModel.PROP_EMAIL) != null;

    Predicate<NodeRef> containsNameWithTarget = node -> serviceRegistry.getNodeService()
            .getProperty(node, ContentModel.PROP_NAME).toString().contains(targetName);

    Consumer<NodeRef> addEmailProperty = node -> serviceRegistry.getNodeService()
            .setProperty(node, ContentModel.PROP_EMAIL, adminAddress);

    Consumer<NodeRef> emailNode = node -> serviceRegistry.getNodeService()
            .addAspect(node, ContentModel.ASPECT_EMAILED,null);

    Stream<NodeRef> delveNodeTree(NodeRef root) {
        Stream<NodeRef> childStream = serviceRegistry.getNodeService().getChildAs-
socs(root).stream().map(ChildAssociationRef::getChildRef);
        return Stream.concat(Stream.of(root), childStream.flatMap(this::delveNodeTree));
    }
... public setters ...
}
```

Figure 9. A library of lambda functions and stream methods

Consider this use of the lambda library:

```
public class SearchAndFilter {
    private NodeRefLambdaLib lambdaLib;

    public List<NodeRef> getAllEmailNodes(NodeRef root) {
        // recursively delve all nodes starting from root
        return lambdaLib.delveNodeTree(root)
                // while that is happening, grab all that have the checkout aspect
                .filter(lambdaLib.hasEmailAspect)
                // then, grab all that have the email property value set
                .filter(lambdaLib.hasEmailPropertyValue)
                // finally, collect the positive results to a new list
                .collect(Collectors.toList());
    }

    public void setLambdaLib(NodeRefLambdaLib lambdaLib) {
        this.lambdaLib = lambdaLib;
    }
}
```

Figure 10. Advanced stream-based filtering mechanism

This code recursively searches for a state given each node from a certain point; one can achieve this using the **delveNodeTree** method in the lambda library, which recursively maps the tree-like structure to a flat list. This is the power of the **flatMap** method in the Streams library when applied to a list. It is worth noting that the evaluation of the recursive structure is not completely lazy due to an implementation bug in **flatMap**. The **filter** method takes a lambda function as a parameter. This method allows developers to use a lambda function as a predicate to lazily filter out each node with a given aspect and property. The **collect** method then eagerly gathers all the floating results into a concrete new list. The lambda functions which an Alfresco developer creates can be used in other places within the application as well and is not limited to the domain of the search.

Consider the following example:

```
public class EnrichMetadata {
    private NodeRefLambdaLib lambdaLib;

    public void enrichNodesWithEmailData(NodeRef root) {
        // recursively gather all nodes starting from root
        lambdaLib.delveNodeTree(root)
                // while that is happening, grab all that have the given target string within the name
                .filter(lambdaLib.containsNameWithTarget)
                // then, grab all that do not yet have the email property value set
                .filter(lambdaLib.hasEmailPropertyValue.negate())
                // check out each of the nodes
                .peek(lambdaLib.emailNode)
                // then, add the email property to each of them
                .forEach(lambdaLib.addEmailProperty);
    }

    public void setLambdaLib(NodeRefLambdaLib lambdaLib) {
        this.lambdaLib = lambdaLib;
    }
}
```

Figure 11. Find, filter and enrich nodes with particular metadata using advanced streams.

In this example, there is clear re-use of the filter functions, especially the re-use of the **hasEmailPropertyValue** which is negated to show that there is no need for elements with this property in the current search. The **peek** function adds the aspect for each of the discovered nodes. The **forEach** terminator function adds the email property to each of the nodes upon termination of the process. The **forEach** function is what is called a terminator, which allows the stream to end. The Stream interface uses what is called the Builder pattern, commonly used to compose complex objects and business logic.

Conclusion

Java 8 provides many benefits to the functionality and readability of Alfresco Platform Extensions written with the Alfresco Java API. Builder, Observer and other design patterns used in computer programming can be expressed more fluidly with lambda functions. Alfresco Certified Engineers may now utilize lambda functions in addition to their usual practice to aid in the development of solutions. They may also refactor old solutions to use the new Java 8 language features. In all, the use of Java 8 features in Alfresco should enhance the code rather than make it more confusing. With continued use and study of lambda functions in Alfresco, the developer will create best-practices and new uses of lambdas within their own extensions.

Contact us

Discover Technologies is an Alfresco partner and has multiple Alfresco Certified Engineers available for questions regarding the concepts in this paper. Contact us at info@discovertechnologies.com.

Coming Soon

A functional programming review using Scala within Alfresco.

References

Alfresco Software Inc. (2017, February 3). *Compatibility Matrix*. Retrieved from Alfresco Documentation: http://docs.alfresco.com/community/concepts/alfresco-sdk-compatibility.html

Alfresco Software Inc. (n.d.). *SearchService*. Retrieved 02 14, 2017, from Alfresco Documentation: http://docs.alfresco.com/5.1/references/dev-services-search.html

Get started with the Java Collections Framework. (n.d.). Retrieved 2 14, 2017, from JavaWorld: http://www.javaworld.com/jw-11-1998/jw-11-collections.html

Oracle. (2016, July 18). *Anonymous Classes*. Retrieved from Java Documentation: https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Oracle. (2016, January 12). *Package java.util.stream*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Warburton, R. (2014, October 7). *Java 8 Lambdas - A Peek Under the Hood*. Retrieved from InfoQ: https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood

Wolf, D. (2016, August 30). *The top 5 Java 8 features for developers.* Retrieved from InfoWorld: http://www.infoworld.com/article/3113752/java/the-top-5-java-8-features-for-developers.html