AppEng Aluminum Reference Guide

Introduction

AppEng Aluminum is a framework for building applications at Apple. It is primarily concerned with building microservice based SOA style applications although it can be used to build plain java projects or even monolithic web applications. The foundation of the framework is Spring Boot which provides a powerful configuration mechanism for building applications using the Spring Framework. Additionally security is provided by extending the Spring Security framework with additional capabilities that are specific to Apple. For more information on Spring Boot refer to the Spring Boot Reference Guide **you should read this in its entirety**. For more information on Spring Security refer to the Spring Security Reference Guide.

Getting Started

Creating Your Project

To start using Aluminum generate a project using AppEng Composer. If you have an existing project based on the previous AppEng Platform Shared Components framework, you can create a new module and copy over your business logic, but the easiest method is to use the AppEng Composer Migration Tool.

Configuring Your Project

First you should understand how Spring Boot manages configuration files and the various options that are available to you. Refer to the Spring Boot Reference Guide for details. Additionally this guide provides an appendix containing common configuration properties.

There are a number of configuration properties that you can use to control the behavior of AppEng Aluminum. These are in addition to the properties provided by Spring Boot. For more information see Aluminum Config Properties

You can bundle a properties file inside your deployable artifact for configuration items that do not change across environments:

- src/main/resources/application.properites - This file will be packaged inside your application and as such should only contain configuration properties that are invariant across environments. Since it is checked into source code you should not place any sensitive information in this file.

External configuration should either be placed outside of your deployable artifact and referenced on startup or provided in environment variables. Refer to the Spring Boot Reference Guide for details on externalized configuration.

AppEng Composer will generate a sample properties file with all of the required properties for the features you selected. This will be done in the src/main/app/conf/ directory which will be included outside of your application. It is **highly recommended** that you take the properties that won't change across environments out of this file, and embed them into your JAR in src/main/resources/application.properties. You can then use the template as a guide for how to configure environment-specific properties. This will allow you to keep the mandatory properties down to a minimum and make switching between environments easier.

Note regarding property name binding

Spring Boot binds properties to Java bean objects during configuration. It allows property names to be bound using loose matching. As a result you can use hyphenated names (e.g. "this-is-my-property") or camel case (e.g. "thisIsMyProperty"). The exception to this is where the property name component is a key in a Map. For example:

```
fooBar.things.myThing=foo
fooBar.things.anotherThing=bar
```

In this example the "things" component will be bound as a Map<String, Object> which makes "myThing" and "anotherThing" keys in the map. While the following configuration is equivalent:

```
foo-bar.things.myThing=foo
foo-bar.things.anotherThing=bar
```

the following is not:

```
fooBar.things.my-thing=foo
fooBar.things.another-thing=bar
```

In general it is best to pick a consistent properties format and stick to it for both keys and other property components. In the examples that follow we will use camel case.

Spring Boot also supports binding from environment variables. This works for most properties however there is an issue with Map objects containing complex objects. For example the following property:

```
foo.bar.things.one.name=value
```

will bind to a Map<String, ?> called "things" with a mapping key "one" to a complex object with a property "name". The corresponding environment variable:

```
FOO_BAR_THINGS_ONE_NAME=value
```

will instead bind to a Map<String, ?> called "things" with a mapping key "ONE_NAME". The suggested workaround is to use then known environment variable SPRING_APPLICATION_JSON.

This may be addressed in future releases of Spring Boot. See:

https://github.com/spring-projects/spring-boot/issues/2880 https://github.com/spring-projects/spring-boot/issues/5315

for more details.

Configuring SSL

For information on configuring SSL please refer to the Spring Boot Reference Guide. If you are running a standalone Java application you can refer to Configuring SSL for the RestClientFactory in this guide.

Running Your Project

There are multiple ways to run your project depending on what you're trying to accomplish:

Running in your IDE

Your application contains a main class that you can run in your IDE. You will need to configure your IDE to point to any configuration overrides that you wish to use.

Running from the command line

If you generated your project using AppEng Composer you will be setup to use the Spring Boot Maven Plugin. From the command line type:

```
mvn spring-boot:run
```

The plugin is configured to allow remote debugging. The port is specified as a property in your pom.xml file. If you want to override the port you can use the following:

```
mvn spring-boot:run -Ddebug.port=9876
```

Finally if you want to have the JVM pause before you connect use the following:

```
mvn spring-boot:run -Ddebug.suspend=y
```

Running as a deployed application

The output of your build should be an executable jar or war. The Maven Spring Boot plugin will package your application for you. You can use AppEng Composer to choose a target deployment platform and have the necessary startup/shutdown scripts provided for you.

Building Your Project in Rio

Rio is Apple's preferred build platform for managing your project's builds, including release and pull request verification builds. It also offers compatibility with their Quality Platform for integration tests. To get started, create a file named *rio.yml* in the root of your project like the one below:

```
schemaVersion: '2.0'

builds:
  - branchName: <release-branch-name>
    build:
      template: buildozer:v1:prb
  - branchName: <release-branch-name>
    build:
      template: buildozer:v1:publish
    publish:
      release: true
    finally:
      tag:
        expression: '<service-name>-${CODE_VERSION}'

  - branchName: <development-branch-name>
    build:
      template: buildozer:v1:prb
    publish:
      release: false
      reports:
        junit:
          path: '**/surefire-reports/*.xml'
        findbugs: true
        checkstyle: true
        jacoco:
          exec: '**/target/jacoco.exec'
          src: '**/src/main/java'
  - branchName: <development-branch-name>
    build:
      template: buildozer:v1:publish
    publish:
      release: false
      reports:
        junit:
          path: '**/surefire-reports/*.xml'
        findbugs: true
        checkstyle: true
        jacoco:
          exec: '**/target/jacoco.exec'
          src: '**/src/main/java'

machine:
  java:
    version: 1.8
```

This sets up pull request verification builds for both your release and your development branches, sets up a release build for your release branch, and setus up a non-release build for your development branch. After committing this to your development branch, register this as a new project on Rio.

Additionally, if you want your Rio build to deploy to a target system, you will need to package your modules as Rio Applications. This can be done by configuring the proper Rio Assembly Descriptor for each application you want to deploy.

Any new module created by AppEng Composer will generate a sample rio.yml file as well as the necessary assembly descriptors for buiding a Rio application. The above is only necessary for existing applications.

Deploying to PIE Compute

When generating a new module in AppEng Composer or updating an existing module, you can select the "PIE Compute" deploy option, which will generate a sample apps.yml file for deploying to PIE Compute. Included in this is a sample start command:

```
java
-jar my-app-*.war
--spring.config.location=${PLATFORM_SECRETS_PATH}/application.properties
--server.port=$PORT_WEB
```

This expects that a secret named `application.properties` has been defined for the environment using PIE Secrets. This allows you to push configuration to your instance on a per-environment basis. Since this file must end in `.properties` or `.yml` in order for Spring to recognize it, the secret can only be exposed as a file and not an environment variable. This can be achieved by writing a secret similar to:

```
secrets write -expose-env=false -file /path/to/config/my-app-dev.properties "workspace-app-dev/org-workspace-app/my-app/application.properties"
```

For more information on setting up your workspace and environments, read the PIE Compute and PIE Secrets documentation.

Aluminum Rest Client Module

In a microservice world a lot of your code will be concerned with calling other services. We provide a factory that allows you to get a JAX-RS Client that can be customized to call a specific service.

To obtain a JAX-RS Client inject the RestClientFactory object into your object and call the clientForService method:

```
@Named
public class MyObject {
    private final Client client;

    @Inject
    public MyObject(final RestClientFactory clientFactory) {
        this.client = clientFactory.clientForService("myservicename");
    }

    public String getSomeData() {
        final WebTarget target = client.target("http://foo.bar");
        return target.request().get(String.class);
    }

//...
}
```

The JAX-RS api is somewhat clunky and using it can result in a lot of duplicated code. If you prefer to work at a higher level of abstraction you can use the ServiceCaller class.

The RestClientFactory provides a convenience method for getting a ServiceCaller:

```
@Named
public class MyObject {
    private final ServiceCaller serviceCaller;
```

```
    @Inject
    public MyObject(final RestClientFactory clientFactory) {
        this.serviceCaller = clientFactory.serviceCallerForService("myservicename");
    }

    public String getSomeData() {
        return serviceCaller.callGet("http://foo.bar", String.class);
    }
//...
}
```

Customizing your service call

There are several properties that you can use to customize how calls are made to an HTTP service:

```
appeng.aluminum.rest.client.services.theServiceName.baseUrl=https://foo.bar
appeng.aluminum.rest.client.services.theServiceName.authenticationType=HMAC
```

Customizing connection properties

You can customize the default settings for REST client connections by setting the following properties:

```
    appeng.aluminum.rest.client.connectionProperties.maxConnectionsTotal=10
    appeng.aluminum.rest.client.connectionProperties.maxConnectionsPerRoute=10
    appeng.aluminum.rest.client.connectionProperties.connectionTimeoutMillis=15000
    appeng.aluminum.rest.client.connectionProperties.socketTimeoutMillis=30000
    appeng.aluminum.rest.client.connectionProperties.defaultKeepAliveTimeoutMillis=-1
    appeng.aluminum.rest.client.connectionProperties.idleConnectionTimeoutMillis = 30000
```

It is also possible to customize some connection settings on a per-service basis. This is highly dependent on the underlying JAX-RS implementation as well as the connection engine underneath (e.g. HttpClient). Aluminum makes a best effort to set these properties with the underlying implementations however it is not always possible to support this behavior entirely:

```
    appeng.aluminum.rest.client.services.theServiceName.connectionProperties.connectionTimeoutMillis=45000;
    appeng.aluminum.rest.client.services.theServiceName.connectionProperties.socketTimeoutMillis=75000;
```

Connection Pooling

By default Aluminum configures a RESTeasy based JAX-RS client with Apache HttpClient as the underlying HTTP client/pooling implementation. Each configured service will be configured with its own set of pooled connections. The number of connections may be manipulated by using the following properties:

```
    appeng.aluminum.rest.client.connectionProperties.maxConnectionsTotal=10
    appeng.aluminum.rest.client.connectionProperties.services.foo.maxConnectionsTotal=10
```

The maxConnectionsPerRoute setting may additionally limit the number of connections by route for each pool. This setting should only be necessary for the default service or in the case where the legacy shared pool is used. To disable per service connection pooling and revert to a global shared pool use the following property:

```
    appeng.aluminum.rest.client.enginePerService.enabled=false
```

Connection Pooling Metrics

The following metrics metrics are captured for each configured HTTP connection pool:

Metric Name	Description
common.http.pooling..max	the maximum number of connections allowed in the pool
common.http.pooling..available	the number of available connections in the pool
common.http.pooling..leased	the number of connections currently checked out of the pool

Metric Name	Description
common.http.pooling..pending	the number of requests waiting for a leased connection

Configuring SSL for the RestClientFactory

The RestClientFactory supports the following SSL modes:

- **SERVER** - Defer to the server.ssl configuration of Spring Boot. If no SSL is configured using server.ssl then **JVM** mode is used.
- **JVM** - Defer to the default settings of the JVM. This roughly translates to using either the default cacerts file or whatever is configured using the javax.ssl system properties.
- **CUSTOM** - Specify settings using appeng.aluminum.rest.client.ssl configuration. This is useful if you want to define custom SSL outside of a server context.
- **NONE** - SSL is completely disabled. This is unlikely to be terribly useful.

By default the RestClientFactory will use **SERVER** mode and defer to Spring Boot for configuration. If you have not configured Spring Boot SSL then the configuration will degrade to **JVM** mode.

For more information about configuring SSL with Spring Boot refer to the Spring Boot Reference Guide.

In a standalone Java application it seems odd to use the Spring Boot server context to define properties. For this reason the **CUSTOM** mode is provided. The **CUSTOM** mode is also useful in the rare case where you want to a different SSL context for inbound and outbound calls.

Replacing the HTTP Engine

You can replace the HTTP engine altogether by providing a bean that implements the ClientEngineProvider interface or a bean that implements the RestClientProvider interface.

Aluminum Auth Module

The aluminum-auth module provides tools for authentication and authorization. Authentication is built as a series of Spring Security AuthenticationProvider implementations. The Aluminum Framework will look for specific headers in the request among other characteristics (presence of jars, etc) to determine which authentication method to use.

Using the Authentication Mode property

The Aluminum Framework provides the ability to switch the authentication mode used by your service. The supported modes are as follows:

- MOCK - Inbound requests are treated as authenticated and the authentication context is populated with a mock user
- OPTIONAL - Inbound requests are not required to be authenticated. If credentials are provided they are used to populate the authentication context, otherwise it will be empty.
- REQUIRED - All secure paths require authentication. This is the default.

The authentication mode can be set using the following property:

```
appeng.aluminum.auth.mode=REQUIRED
```

Note that the authentication mode applies only to inbound calls to your service. Outbound calls are governed by the configuration of the Rest Client Factory. Additionally, the authentication mode has no effect on the CSRF filter so if your service is user facing you may still receive 403 responses if you fail to provide the correct CSRF headers. See the CSRF Configuration section for further details.

Providing Your Own User Object

If you wish to customize your user object you can provide an implementation of UserFactory. The type of authentication your application uses will dictate which interface to implement:

- Simple authentication: com.apple.ist.appeng.auth.UserFactory
- IdMS AuthWeb only: com.apple.ist.appeng.auth.idms.IDMSUserFactory
- IdMS Cocoa client authentication: com.apple.ist.appeng.auth.idms.IDMSRpcUserFactory

Providing Your Own Roles

Create a bean that implements RoleProvider. This interface will be used during the authorization stage of login.

Some authentication providers conflate the notion of authentication and authorization. As such a user's groups may be provided during the authentication process instead of the authorization process. For this reason RoleProvider accepts a collection of known IDs representing groups to which the current user belongs. This list of group IDs can be used to make very simple authorization decisions. For finer-grained authorization decisions an authorization service such as AppEng Entitlements should be used.

Making Authenticated Service Calls

The auth module provides an abstraction built on top of the ServiceCaller/RestClientFactory infrastructure which allows you to make service calls using an optional User object. The factory is called AuthenticatedServiceCallerFactory:

```
@Named
@Path("/api/v1")
public class MyAwesomeController {
    private static final GenericType<Map<String, String>> MAP_STRING_STRING_TYPE = new GenericType<Map<Stri
    };

    private final AuthenticatedServiceCaller serviceCaller;
    private final AuthenticatedWebPrincipalProvider principalProvider;

    @Inject
    public MyAwesomeController(final AuthenticatedWebPrincipalProvider principalProvider, final Authenticat
        this.principalProvider = principalProvider;
        this.serviceCaller = serviceCallerFactory.authenticatedServiceCallerForService("myAwesomeService");
    }

    @Path("/message")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Map<String, String> message(@Context final HttpServletRequest request) {
        final Optional<User> user = principalProvider.retrieveUser(request);
        return serviceCaller.callGet("http://localhost:1234/api/v1/hello", user, MAP_STRING_STRING_TYPE);
    }
}
```

Aluminum Auth IdMS tools

Aluminum provides many different options for interacting with IdMS.

You should set the IdMS for your application as illustrated below:

```
appeng.aluminum.auth.idms.appId=12345678
appeng.aluminum.auth.idms.appIdKey=foobar
appeng.aluminum.auth.idms.appAdminPassword=ImNoTtElLiNg
```

Selecting the Appropriate Environment

In order to identify defaults for IdMS login and validation, and for querying Person and Group data, it is important to set the mode for IdMS operations. This is set via the `appeng.aluminum.auth.idms.mode` property and the valid values are:

- UAT_CORPORATE - For Corporate applications in a non-Prod environment.
- PROD_CORPORATE - For Corporate applications in a Prod environment.
- UAT - For all other applications in a non-Prod environment.
- PROD - For all other applications in a Prod environment.

If you are unsure as to how your application should be classified, contact IdMS to find out how your AppID is configured.

IdMS API Authorizations

Even if you're not using the IdMS API directly, by virtue of authenticating with and getting basic person data from IdMS via Aluminum, your AppID needs to be authorized for such requests by Account Security. For Aluminum, this means that, at minumum, your AppID must have the following authorizations:

- `person:processFetchRequest / com.apple.ist.ds2.pub.person.PersonRequest`
- `group:processFetchGroupsIn / com.apple.ist.ds2.pub.group.GroupMemberRequest`

Your AppID may need additional authorizations depending on how your application will be interacting with IdMS. If you still get any authorization errors from IdMS, be sure to contact them or Account Security directly for assistance.

Web Clients

IdMS AuthWeb

To use IdMS AuthWeb authentication is enabled by default in a Service Facade application.

If for some reason you want to use different credentials than the global set, specifically for web clients you can configure them like this:

```
appeng.aluminum.auth.idms.authweb.appId=12345678
appeng.aluminum.auth.idms.authweb.appIdKey=foobar
appeng.aluminum.auth.idms.authweb.appAdminPassword=ImNoTtElLiNg
```

You can also explicitly disable AuthWeb authentication using the following:

```
appeng.aluminum.auth.idms.authweb.enabled=false
```

IdMS OAuth

IDMS provides an implementation of OAuth 1.0a. At the time of writing you need to make a special request to have it enabled for your AppID.

To enable OAuth configure the following:

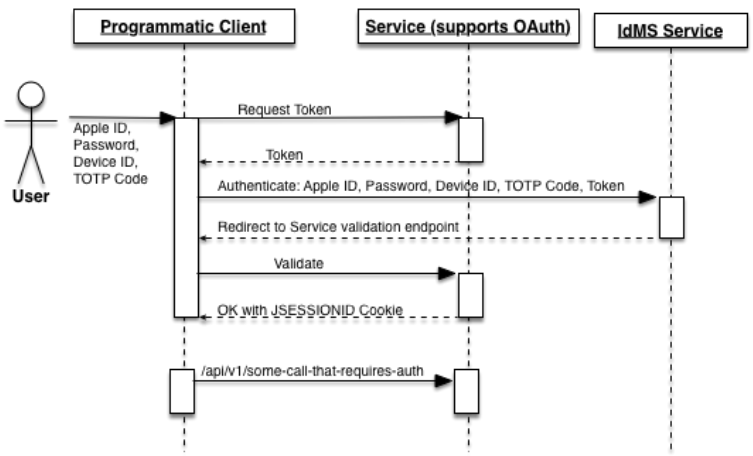
```
appeng.aluminum.auth.idms.oauth.enabled=true
appeng.aluminum.auth.idms.oauth.callbackUrl=https://foo.bar.corp.apple.com:8443
appeng.aluminum.auth.idms.oauth.consumerSecret=ImNoTtElLiNg
```

If you want to support programmatic clients (e.g. Java applications, Python scripts, etc.) you can enable the request token endpoint:

```
appeng.aluminum.web.security.idms.oauth.controller.enabled=true
```

This enables an endpoint that is freely accessible to anyone. The endpoint serves up a service specific request token that can be used to initiate the programmatic authentication flow.

Clients can be implemented using the following flow:



The most important detail of the above flow is that the authentication credentials should **never** be sent to the service. The credentials should only ever be sent directly from the client application to IdMS. IdMS will take care of validating the credentials and triggering the appropriate redirect.

Java based clients can simply use the *IdmsOAuthServiceProgrammaticAuthenticationClient* class which encapsulates the above logic. An example of this can be seen in the oauth-client application in the Aluminum Samples. There is also a Python example included.

Cocoa Based Clients

Cocoa clients should use the AppEng Cocoa Client library to authenticate with Aluminum based services.

IdMS iOS Authentication

If for some reason you want to use different credentials for iOS clients:

```
appeng.aluminum.auth.idms.ios.appId=12345678
appeng.aluminum.auth.idms.ios.appIdKey=foobar
appeng.aluminum.auth.idms.ios.appAdminPassword=ImNoTtElLiNg
```

IdMS OSX Authentication

If for some reason you want to use different credentials for OS X clients:

```
appeng.aluminum.auth.idms.osx.appId=12345678
appeng.aluminum.auth.idms.osx.appIdKey=foobar
appeng.aluminum.auth.idms.osx.appAdminPassword=ImNoTtElLiNg
```

IdMS HSA2 Authentication

If for some reason you want to use different credentials for HSA2 clients:

```
appeng.aluminum.auth.idms.hsa2.appId=12345678
appeng.aluminum.auth.idms.hsa2.appIdKey=foobar
appeng.aluminum.auth.idms.hsa2.appAdminPassword=ImNoTtElLiNg
```

IdMS App To App Authentication

If for some reason you want to use different credentials for app to app clients:

```
appeng.aluminum.auth.idms.appToApp.appId=12345678
appeng.aluminum.auth.idms.appToApp.appIdKey=foobar
appeng.aluminum.auth.idms.appToApp.appAdminPassword=ImNoTtElLiNg
```

A3 Token based authentication

Calling a Service using A3 App To App authentication

Configure the following minimal properties (all values should be replaced with real values)

```
#
# Auth Config
#
appeng.aluminum.auth.idms.appId=12345678
appeng.aluminum.auth.idms.appIdKey=foobar
appeng.aluminum.auth.idms.appAdminPassword=ImNoTtElLiNg
#
# A3 Config
#
appeng.aluminum.auth.idms.appToApp.a3.senders.someAwesomeService.receiverAppId=987654
appeng.aluminum.auth.idms.appToApp.a3.senders.someAwesomeService.contextString=sOmEtHiNgSeCrEt
#
# Rest Client Config
#
appeng.aluminum.rest.client.services.someAwesomeService.authenticationType=A3
```

The first section (Auth Config) specifies your IdMS credentials. It is possible to use different credentials from your applications credentials for App To App Authentication but it is not recommended.

The second section (A3 Config) configures your application to be a sender for the service "someAwesomeService". This name is something that you choose to represent your service and is what you will use in your code to get a client for that service. There are other settings that you can use to customize the A3 interaction. We are only showing the required settings.

The final section (Rest Client Config) tells the Rest Client classes that you want a JAX-RS Client for a service named "someAwesomeService" and that you want it to use A3 authentication.

In your code you would do something similar to the following:

```
@Named
public class MyObject {
    private final AuthenticatedServiceCaller serviceCaller;

    @Inject
    public MyObject(final AuthenticatedServiceCallerFactory clientFactory) {
        this.serviceCaller = clientFactory.authenticatedServiceCallerForService("someAwesomeService");
    }

    public String getSomeData(final Optional<? extends User> possibleUser) {
        return serviceCaller.callGet("http://foo.bar", possibleUser, String.class);
    }

//...
}
```

Receiving A3 calls

By default there is little that needs to be done to accept A3 calls. So long as the a3tokenclient jar is present and you've configured your IdMS credentials it should work. One thing you may want to configure is your context strings:

```
appeng.aluminum.auth.idms.appToApp.a3.receiver.defaultContextStringVersion=1
appeng.aluminum.auth.idms.appToApp.a3.receiver.contextStrings.1=foo
appeng.aluminum.auth.idms.appToApp.a3.receiver.contextStrings.2=bar
appeng.aluminum.auth.idms.appToApp.a3.receiver.contextStrings.3=baz
```

Legacy App To App

Legacy App To App is enabled by default. To disable it configure the following property:

```
appeng.aluminum.auth.idms.appToApp.legacy.enabled=false
```

Calling a Service using Legacy App To App authentication

Configure the following minimal properties (all values should be replaced with real values)

```
#
# Auth Config
#
appeng.aluminum.auth.idms.appId=12345678
appeng.aluminum.auth.idms.appIdKey=foobar
appeng.aluminum.auth.idms.appAdminPassword=ImNoTtElLiNg
#
# Legacy App To App Config
#
appeng.aluminum.auth.idms.appToApp.legacy.enabled=true
appeng.aluminum.auth.idms.appToApp.legacy.services.someAwesomeService.serviceAppId=987654
#
# Rest Client Config
#
appeng.aluminum.rest.client.services.someAwesomeService.authenticationType=LEGACY_APP_TO_APP
```

The first section (Auth Config) specifies your IdMS credentials. It is possible to use different credentials from your applications credentials for App To App Authentication but it is not recommended.

The second section (Legacy App To App Config) configures your application to be a sender for the service "someAwesomeService". This name is something that you choose to represent your service and is what you will use in your code to get a client for that service. There are other settings that you can use to customize the Legacy interaction. We are only showing the required settings. Note that Legacy App To App is disabled by default so you must enable it to use it.

The final section (Rest Client Config) tells the Rest Client classes that you want a JAX-RS Client for a service named "someAwesomeService" and that you want it to use A3 authentication.

To use this client you could use the exact same code as listed above in the A3 section. The caller does not need to know how the authentication is configured, the framework will take care of it.

Aluminum Auth HMAC tools

In many cases you may want to make a call from one microservice in an application to another microservice in the same application. If this call conforms to all of the following:

- SSL
- Within the same VLAN
- Between services of the same application (e.g. same App ID)

you can use HMAC to secure the call.

Implementation/Approach

Our current HMAC auth implementation works by both client and server generating a cryptographic hash (http://en.wikipedia.org/wiki/Hash-based_message_authentication_code) of the http request in question. Both client and server will be configured with a shared secret, thereby ensuring that the hash produced by both sides will be the same. The client attaches its HMAC to the outgoing request, and the server will generate its own HMAC upon receiving the request. If the HMACs are equal the server considers the message authenticated.

The basic algorithm is as follows:

- Client (app id 1) wants to talk to Service (app id 1). Client and Service have been configured with the same shared secret. HMAC can only be used to authenticate between web components in the same app ecosystem (same app id). Authentication between two apps with different app ids should be done using DS App to App Authentication.
- Client generates an HMAC of the outgoing request based on the following attributes of the http request:
  - method
  - url
  - body
  - headers (not all headers, you configure a whitelist of headers to be included in the HMAC, see property hmac.includedHeaders)
  - sent timestamp
- Client calls Service, passing the HMAC in the header X-Apple-HMAC-Digest. The timestamp from the previous step is also supplied in the header X-Apple-HMAC-Sent-Timestamp.
- Service retrieves the client HMAC from header & generates its own HMAC. Service compares HMAC codes to see if they are equal. If HMAC codes are equal, request is considered authenticated. If HMAC codes not equal, a 403 is returned.

Guidance on Choosing an HMAC Secret

InfoSec policy dictates that the HMAC secret must meet the following requirements:

- Hexadecimal represented.
- At least 32 hexadecimal characters long (i.e. 16 binary bytes long).
- The decoded key (i.e. textual representation of the hexadecimal) must not have any blacklisted words (e.g. Apple12, Apple@12, apple12, Apple12, Infy@12, infy@12).
- At least one of the characters in the decoded key must not be a printable character (i.e. at least one of the hexadecimal values must not be between 0x20 through 0x7E).

If you provide an HMAC secret Aluminum will enforce these requirements at startup. If you don't provide an HMAC secret then HMAC will be disabled.

As implied by the above requirements, your HMAC secret should be a randomly generated value and not a value chosen by a human (i.e. not a password). The recommended way to generate your HMAC secret is to use the following openssl command:

```
openssl rand -hex 16
```

Finally, when distributing the shared secret, one should exercise the same level of precaution as transmitting any private key within Apple. Namely, never send a shared secret to someone else via insecure methods like email or instant messenger. Instead, if you must send it, use something like Attache.

Basic configuration

To enable HMAC authentication you must configure an HMAC secret:

```
appeng.aluminum.auth.hmac.secret=NoTaVaLiDhMaCsEcReT
```

Guidance on Rotating Your HMAC Secret

InfoSec requires that HMAC secrets be rotated twice a year by app teams. You may rotate your secret, with zero downtime, by using the following properties:

```
#
# Updated clients send this
#
appeng.aluminum.auth.hmac.secret=AlSoNoTaVaLiDhMaCsEcReT
#
# Older clients send this. Better update before it expires!
#
appeng.aluminum.auth.hmac.previousSecret=NoTaVaLiDhMaCsEcReT
appeng.aluminum.auth.hmac.previousSecretExpirationTimestamp=1970-01-01T00:00:00-00:00
```

The procedure would be pretty simple: when you decide you want to rotate your secret, you do the following:

- Move your current secret to the *appeng.aluminum.auth.hmac.previousSecret* property
- Set *appeng.aluminum.auth.hmac.previousSecretExpirationTimestamp* property to define how long your previous secret will be valid for.
- Update the *appeng.aluminum.auth.hmac.secret* property with your new secret.

You would update these three properties in your app config file at the same time, and then restart your receiving servers. Your servers should now accept HMACs generated using both the old and new secrets for a period of time (defined by *appeng.aluminum.auth.hmac.previousSecretExpirationTimestamp*). This allows your clients time to rotate to the new secret without causing any downtime. Once past the expiration time, only the new HMAC secret will work when authenticating.

HMAC Auth Limitations & Known Issues

Client & Server Clock Synchronization

Because the AppEng HMAC auth implementation attaches a timestamp to the HMAC'ed message, there is an inherent dependency on the system clock for both the client and server. If the client and server clocks fall too far out of sync with each other (also called called clock skew), then the effective expiration window for an HMAC'ed message could be extended or shortened beyond the intended thresholds. In a properly functioning environment each server should be periodically synching it's system clock using NTP (https://en.wikipedia.org/wiki/Network_Time_Protocol), mitigating the potential for clock skew. However this process could break down under the following scenarios:

- NTP server could be the target of DoS attack, effectively making it unavailable
- NTP server could be compromised, effectively causing servers to synchronize their clocks to unreliable values
- NTP server might be acting normally but still cause the system time on a server to change abruptly if the two are out of sync by a wide margin
- System time on a server might be changed by an administrator
- etc.

Even in a properly functioning system, some clock skew is inevitable - you will never have all servers in a distributed environment perfectly in sync with each other with regards to time. In order to consider the implications, we can consider two scenarios:

Scenario 1: Client clock is behind server clock In this case the HMAC'ed request would "look older" than it really is to the server. In a worst case scenario all client requests might be rejected by the server until the clock skew issue is addressed, as the server would consider all client requests expired. This would effectively create a DoS attack.

Scenario 2: Client clock is ahead of server clock In this case the HMAC'ed request would look like it was sent from the future. In a worst case scenario all client requests would be considered valid indefinitely, effectively never expiring, thereby opening up the server to replay attacks.

Accounting for Clock Skew

The AppEng HMAC auth server filter can tolerate some clock skew between the client and server. How much it can tolerate depends on how you configure the following properties:

```
# The expirationMillis property is used by the server to determine whether or not the client request is exp
appeng.aluminum.auth.hmac.expirationMillis=30000

# The timestampToleranceMillis property will be used by the server to flag suspicious client timestamps whi
appeng.aluminum.auth.hmac.timestampToleranceMillis=15000
```

The expirationMillis property is used by the server to determine whether or not the client request is expired, so naturally a larger expiration window will allow for more clock skew on the client side as it relates to scenario 1. The default is 30 seconds, so assuming negligible network latency, the client clock could fall behind the server by roughly 30 seconds before you would start to see any issues. More network latency would equate to less skew tolerance.

The timestampToleranceMillis property will be used by the server to flag suspicious client timestamps which deviate too far from the server time. For example, a client request with a timestamp which is 60 minutes ahead of the server time suggests a serious clock skew issue between the client and server, possibly malicious. In such a case we would like to reject that request and log it as suspicious. In the opposite case, where the client timestamp is 60 minutes behind the server, the expiration policy would already ensure the request is rejected, but we'd still like to flag it as suspicious. More formally, any client timestamp which is less than serverTime-timestampToleranceMillis or greater than serverTime+timestampToleranceMillis will be considered suspicious. The default is 15 seconds, meaning a client timestamp that is older than 15 seconds from the server time would be logged as suspicious, while a client timestamp that is more than 15 seconds ahead of the server time would be logged a suspicious and rejected. The client clock would therefore need to be at least 15 seconds ahead of the server clock before you would start to see any requests being rejected, and likely more depending on network latency.

Tracking Client Timestamps

Client timestamps are tracked per ip address in the HMAC server filter. If two requests from the same ip appear to be going back in time, the server filter will log this as suspicious and reject the request. Unfortunately determining what constitutes "back in time" is a bit complicated, since in a highly concurrent system two client requests might reasonably be received and processed by the server out of order from when they were generated. We therefore will use the timestampToleranceMillis parameter to account for this as well; only a subsequent request which is less than previousRequestTimestamp-timestampToleranceMillis will be considered out of order.

Monitoring NTP

Both the AppEng client and server components will start a background thread which will periodically attempt to check the status of the NTP server which is being used to synchronize the local system clock. If the NTP server is unreachable, or an anomalous timestamp is returned from the NTP server, a warning will be logged. We will determine what's an "anomalous" timestamp by tracking the times reported by the NTP server(s) and detecting if the NTP time appears to have jumped forwards or backwards by more than what's allowed for in the timestampToleranceMillis parameter. The following additional properties will be used to control the behavior of this feature:

```
# Enables / disables the background NTP monitoring thread.  Default is enabled.
appeng.aluminum.auth.hmac.ntpMonitoringEnabled=true

# How often the NTP server(s) will be checked.  Default is 15 minutes.  Can't be set to less than 15 minute
appeng.aluminum.auth.hmac.ntpMonitoringIntervalMillis=900000
```

```
# The timeout used when pinging the NTP server(s).  Default is 10 seconds.
appeng.aluminum.auth.hmac.ntpMonitoringShutdownTimeoutMillis=10000
```

Splunk Recommendations

It is strongly recommended that you setup Splunk to trigger alerts for any errors or warnings related HMAC or NTP so that you can catch clock issues early and proactively address them.

You've Been Warned...

You should not be relying on the above features to somehow magically keep all your system clocks in sync; they might catch some issues but cannot replace proactive monitoring and maintenance of your servers.
It's important to re-iterate that the AppEng HMAC auth implementation assumes the client and server clocks are in sync to a relatively close degree (in general a few seconds).
It is your responsibility, or whoever is maintaining your servers, to ensure the system clocks remain in sync on your client and server machines.
If the system clocks greatly fall out of sync then issues with HMAC auth could arise as described in previous sections.

Aluminum Auth X509 Mutual Authentication

Support for mutual authentication is enabled using the built in Spring Boot property:

```
# Either WANT or NEED
server.ssl.clientAuth=...
```

Your service can receive requests from other services using X509 certificate based authentication. There are two variations on mutual authentication supported:

- Client application connects directly to your application
- Client application connects to an intermediate (likely a load balancer). The intermediate connects to your application using mutual authentication. The intermediate also provides the original client's certificate through an agreed upon mechanism (e.g. in a web application this is likely a header)

Authentication

Authentication is based on the standard CAcerts trust store mechanism of Java. To accept a certificate from a given client make sure that you have the certificate's issuer in your trusted key store. By default an instance of DefaultCertificateApplicationDetailsService will be used to build the Application object that will be stored in the security context. You can override this behavior by providing a bean that implements CertificateApplicationDetailsService.

Authorization

Mutual authentication uses an extension of GrantedAuthorityProvider to determine which roles to grant to the client application. By default an instance of DefaultMutualAuthAwareRoleProvider will be used to grant roles based on a token embedded in the certificate. The default implementation does not allow any roles. You can provide a bean that extends DefaultMutualAuthAwareRoleProvider and provide roles based on the extracted token.

Which token is extracted is controlled using the following properties:

```
appeng.aluminum.auth.mutual.tokenRdnType=UID
appeng.aluminum.auth.mutual.tokenPrefix=identity:idms.group.
```

Given a certificate with a SubjectDN of the form:

```
DC=Certificate Manager, O=Apple Inc., OU=management:idms.group.98765, CN=some great team,
UID=identity:idms.group.12345
```

The token extracted would be *12345*.

You can also completely replace the implementation by providing a bean that implements either RoleProvider or MutualAuthAwareRoleProvider.

Related Samples

See the mutual-auth-ssl-business-service sample for an example of the default behavior. See the mutual-auth-ssl-custom-business-service sample for an example of customizing the authorization.

Aluminum Auth Rest Client tools

AuthenticatedServiceCaller

The Aluminum Auth module builds on top of the Aluminum Rest Client module by providing the concept of an authenticated service. Instead of injecting the RestClientFactory into your class you can inject the AuthenticatedServiceCallerFactory to get an AuthenticatedServiceCaller. The AuthenticatedServiceCaller contains methods similar to the ServiceCaller class but takes an Optional<? extends User> on most calls. Providing this optional user object allows the framework to call services with the correct headers to communicate which user or application is making the call.

```
@Named
public class MyObject {
    private final AuthenticatedServiceCaller serviceCaller;

    @Inject
    public MyObject(final AuthenticatedServiceCallerFactory clientFactory) {
        this.serviceCaller = clientFactory.authenticatedServiceCallerForService("myservicename");
    }

    public String getSomeData(final Optional<? extends User> possibleUser) {
        return serviceCaller.callGet("http://foo.bar", possibleUser, String.class);
    }

//...
}
```

We will look at one possible way of obtaining the User object in the Web Module section.

JAX-RS Client Filters

The Aluminum Auth Module provides JAX-RS Client Filters that plug into the RestClientFactory to provide authentication for REST calls. Use of these filters is triggered by configuring the REST client with a specific authentication type using the *appeng.aluminum.rest.client.services.some_service_name.authenticationType* property. The following values are supported:

- HMAC - Uses HMAC authentication
- A3 - Uses IdMS A3 token based App To App Authentication
- LEGACY_APP_TO_APP - Uses IdMS legacy App To App Authentication In each case the underlying authentication framework is configured separately using its configuration properties.

Aluminum Web Module

The Aluminum Web module provides tools for building web apps, primarily RESTful services. It provides integration with Spring Security to leverage Spring Security's pluggable security model. Spring Security also provides protection for things like XSS, Session Fixation, etc. For more information see the Spring Security Reference Guide

In addition to Spring Security integration the web module provides an InfoSec approved CSRF implementation and other tools for building web applications.

Retrieving the current user

A common use case is to get the current user from the execution context. Aluminum provides the **AuthenticatedWebPrincipalProvider** for this purpose:

```
@Named
@Singleton
@Path("/api/v1")
public class MyController {
    private final AuthenticatedWebPrincipalProvider principalProvider;

    @Inject
    public MyController(final AuthenticatedWebPrincipalProvider principalProvider) {
        this.principalProvider = principalProvider;
```

```
    }

    @GET
    @Path("/message")
    @Produces(MediaType.APPLICATION_JSON)
    public Map<String, String> message(@Context final HttpServletRequest request) {
        final Optional<User> possibleUser = principalProvider.retrieveUser(request);
        return Collections.singletonMap("message", "Hello " + possibleUser.get().getName());
    }
}
```

Note that the above code assumes that there is always a logged in user. In other contexts you may want to use the **Optional** to test for the presence of the user or provide a default implementation.

CSRF configuration

The Aluminum CSRF implementation utilizes the HMAC Module to generate a unique per-request token. To support Aluminum CSRF you need to configure HMAC as described in the HMAC Module section. You also need to provide an endpoint for your clients to call to get the initial CSRF token:

```
@Named
@Singleton
@Path("/api/v1")
public class MyController {
    private final CSRFManager csrfManager;
    private final AuthenticatedWebPrincipalProvider principalProvider;

    @Inject
    public MyController(final CSRFManager csrfManager, final AuthenticatedWebPrincipalProvider principalPro

        this.csrfManager = csrfManager;
        this.principalProvider = principalProvider;
    }


    @GET
    @Path("/sessions/current")
    @Produces(MediaType.APPLICATION_JSON)
    public UserSessionDetails currentUserSessionDetails(@Context final HttpServletRequest request, @Context
        csrfManager.generateCSRFTokenAndAddToHeader(request, response);
        final Optional<User> currentUser = principalProvider.retrieveUser(request);
        return new UserSessionDetails(request, currentUser.get());
    }
}
```

The rest of the CSRF manipulation is taken care of for you by the framework. To call the service using JavaScript you can use the AppEng JavaScript Plugins.

CSRF and REST clients

The CSRF filter is intended to be used in conjunction with a client library that can pass the proper required headers a part of the request. The CSRF implementation requires a valid "Referer" header to be set as well as other headers. Setting/spoofing the "Referer" header is disallowed by many browsers and REST clients (including Chrome and POSTman running in Chrome). As a result you will not be able to execute mutation HTTP verbs (e.g. PUT, POST, PATCH, DELETE) using a browser based REST client. You should test these using a proper client equipped to follow the CSRF workflow. AppEng provides JQuery and AngularJS plugins for managing CSRF requests. See AppEng JavaScript Plugins for more information.

In the event that you absolutely must use a browser based REST client to call your service you can disable the CSRF filter:

```
appeng.aluminum.web.csrfFilter.enabled=false
```

Doing so introduces a significant security hole in your application and does not test CSRF integration. As such it is not recommended in most cases.

Using Spring Security's CSRF implementation

Spring Security provides a CSRF implementation. However this implementation may not be acceptable to InfoSec. The Aluminum implementation was reviewed and approved by InfoSec. In particular InfoSec did not want us to use per session tokens but rather change the token on every interaction. Regardless if you wish to use Spring's implementation you can do so with the following configuration:

```
# Enable Spring Security CSRF
security.enableCsrf=true

# Disable Aluminum CSRF
appeng.aluminum.web.csrfFilter.enabled=false
```

Dynamic Client IP Management

In development mode you probably want to send **clientIp** to IdMS as the IP of your VPN connection. By default Aluminum will look for the value sent in the **X-Forwarded-For** header and failing that the IP of the local machine. To override this behavior specify the following:

```
appeng.aluminum.auth.idms.authweb.validation.clientIp.useRequestValue=false
```

This will cause the framework to look for your VPN connection and use that IPv4 address. By default it will look for a connection prefixed by **utun**. If you find that your VPN connection is named something different then you can override the prefix using the following:

```
appeng.aluminum.auth.idms.authweb.validation.clientIp.interfacePrefix=whatever_your_connection_is
```

Customizing Caching Headers

Spring Boot and Spring Security provide a mechanism for disabling caching of dynamic, secure resources. You can enable this by setting the appropriate security properties. For more information refer to the Spring Boot Reference Guide.

If you need more fine-grained control of caching headers you can use the **CacheHeadersFilter** provided by the Aluminum Framework. The filter allows you to group sets of resources and assign specific caching headers for each group.

Given the following configuration:

```
appeng.aluminum.web.cacheHeadersFilter.enabled=true
appeng.aluminum.web.cacheHeadersFilter.groups.data.patterns=/data/**
appeng.aluminum.web.cacheHeadersFilter.groups.data.expires=0
appeng.aluminum.web.cacheHeadersFilter.groups.data.cacheControl=no-store
appeng.aluminum.web.cacheHeadersFilter.groups.data.pragma=no-cache
appeng.aluminum.web.cacheHeadersFilter.groups.webAssets.patterns=/scripts/**,/styles/**
appeng.aluminum.web.cacheHeadersFilter.groups.webAssets.cacheControl=Max-Age=31536000
```

Two caching groups would be created: one matching /data/** and the other mathcing either /scripts/** or /styles/**. A request for */data/foo* would result in the following response headers:

```
Expires: 0
Cache-Control: no-store
Pragma: no-cache
```

while a request for */scripts/foo* would result in the following response headers:

```
Cache-Control: Max-Age=3153600
```

CORS Support

If you wish to support CORS in your application Aluminum provides a way to enable the CORS support in Spring Web. Below is a sample configuration that will enable the CORS filter and configure a path:

```
appeng.aluminum.web.corsFilter.enabled=true
appeng.aluminum.web.corsFilter.paths.api.pattern=/api/**
appeng.aluminum.web.corsFilter.paths.api.allowCredentials=true
```

```
appeng.aluminum.web.corsFilter.paths.api.maxAgeInSeconds=1800
appeng.aluminum.web.corsFilter.paths.api.exposedHeaders=X-Apple-CSRF-Token
appeng.aluminum.web.corsFilter.paths.api.allowedMethods=*
appeng.aluminum.web.corsFilter.paths.api.allowedHeaders=*
appeng.aluminum.web.corsFilter.paths.api.allowedOrigins=https://foo.bar.baz
```

This will allow CORS requests to your API from the domain https://foo.bar.baz. The above example may be too permisssive, you should review the settings to ensure you are being as restrictive as possible for your use case.

To add configuration for another path simply add another named configuration:

```
appeng.aluminum.web.corsFilter.paths.someother.pattern=/some/other/path/**
appeng.aluminum.web.corsFilter.paths.someother.allowedOrigins=https://baz.bar.foo
```

Web Authentication

Mutual Authentication

If you are using mutual authentication you may want to accept proxied client certificates from an intermediary (a load balancer, for example). By default Aluminum Web will look for the certificate in a header named "X-Client-Cert". You can override this behavior by specifying the following:

```
appeng.aluminum.web.auth.mutual.proxiedClientCertHeaderName=X-Apple-Some-Other-Header-Name
```

You can also restrict the IP addresses from which you allow proxied requests. You can do this by specifying a specific IP:

```
appeng.aluminum.web.auth.mutual.enableRequestOriginValidation=true
appeng.aluminum.web.auth.mutual.allowedRequestIps=127.0.0.1
```

You can also restrict by specifying a subnet:

```
appeng.aluminum.web.auth.mutual.enableRequestOriginValidation=true
appeng.aluminum.web.auth.mutual.allowedRequestSubnet=127.0.0.0/32
```

Aluminum Health Check Module

Aluminum provides an interface called HealthCheck which can be used to implement a health check. Additionally most of the data modules provide HealthCheck implementations for their respective data stores. HealthCheck implementations are called periodically by the HealthCoordinator which aggregates the results and determines the overall health of the system.

What Should Have a Health Check?

Resources that are core to your application should have health checks. Examples include databases, message queues, and other items without which your application is completely inoperable.

What Should *NOT* Have a Health Check?

External dependencies such as shared services should *NOT* be included in your health checks. Having a health check on an external dependency effectively couples your application to that shared service and presents the very real possibility of cascading failures which will result in an outage in one service taking down a much larger slice of the overall infrastructure.

To make your application resilient to outages in external dependencies use AppEng Orchestra.

How Often Should The Health Checks Run?

The answer to this is really very application-specific. App teams will need to decide what is right for their particular use case. The general rule of thumb is "as frequently as needed, but no more than that". Basically you need to check frequently enough that your app is responsive to outages, can mark itself unhealthy and the load balancer can remove the unhealthy instance from service before it causes problems. That being said there is a cost to running health checks and you should not put additional load on your resources just for health checks.

Using the HealthController

The HealthController is provided to query the state of the application over HTTPS. The controller provides the following endpoints:

- /health/check - Returns HTTP 200 if the node is healthy or HTTP 503 if it is unhealthy. The "check" endpoint is intended to be used by load balancers, so they are able to take individual nodes out of service when they become unhealthy.
- /health/report - Returns a JSON document showing the status of each individual health check. This endpoint can be used to troubleshoot an individual node.

The endpoints provided by the HealthController are not part of a service's public API and should **not** be used by clients. In particular, they should *never* be routed through a load balancer. A load-balanced "check" or "report" does not make sense, because in both cases the results are specific to the particular node on which they run.

If you do not want to use the HealthController or if you want to provide your own implementation it can be disabled using configuration properties:

```
appeng.aluminum.health.rest.controllerEnabled=false
```

Using Aluminum Health with Spring Boot Actuator

Spring Boot provides health checks as part of their Actuator module. If you add the Actuator module to your classpath Aluminum will detect it and create a bridge between the Aluminum HealthCheck instances and Actuator. Note that Spring Boot's Actuator endpoints require the use of Spring MVC. Spring MVC may not work well with JAX-RS frameworks and getting the mappings configuration right is outside the scope of this document.

Aluminum Metrics Module

Aluminum provides metrics gathering for many common things. The framework Most of the data modules provide metrics for their respective data stores. The underlying implementation is provided by the Dropwizard Metrics library. Additionally the following metric reporters are provided:

- Slf4j
- Hubble
- Ganglia
- JMX
- Dispatch

Each of the reporters can be enabled or disabled using configuration properties.

Note: If more than one reporter is enabled then the Dispatch reporter will be enabled (by default) which will override any of the other reporter's settings.

Dispatch Reporter

The Dispatch reporter resolves the issue of different values being reported at different intervals to different reporters. This reporter is enabled by default if more than one reporter is enabled and its settings override the reporting intervals configured for other reporters. The appeng.aluminum.metrics.reporter.dispatch.* configuration properties control these settings. Optionally, the dispatch reporter may be disabled which will revert to the previous behavior of each reporter reporting values on their own intervals. Note that when using the reset setting for HDR histograms this is reporter should be enabled.

HDR Histogram Backed Metrics

Aluminum also optionally supports Timers backed by HDR histograms, https://hdrhistogram.github.io/HdrHistogram/. To enable this feature, include the following runtime dependencies:

```
<dependency>
    <groupId>org.mpierce.metrics.reservoir</groupId>
    <artifactId>hdrhistogram-metrics-reservoir</artifactId>
    <version>1.1.2</version>
    <type>jar</type>
    <scope>runtime</scope>
</dependency>
<dependency>
```

```
        <groupId>org.hdrhistogram</groupId>
        <artifactId>HdrHistogram</artifactId>
        <version>2.1.9</version>
        <type>jar</type>
        <scope>runtime</scope>
    </dependency>
```

When this is enabled the MetricRegistry injected will by default create Timers/Histograms that are capable of tracking values between 1 microsecond up to 1 hour with up to 3 significant digits of precision.
These settings also configure the Timers to reset their values upon being read such that the values reported for each interval include only those readings for that interval and not over the lifetime of the process. For instance, if the process is idle over this time period then the values recorded for that time period will be 0. To ensure proper reporting of values to all configured reporters, the dispatch reporter should be configured which will dispatch snapshots of all values to all other available reporters.

NOTE: If an attempt is made to update a Timer with a value outside of the configured range of values this will result in an ArrayOutOfBoundsException

PIE Commons Metrics SPI

If you would prefer to interact with metrics through PIE's Java Commons Metrics SPI instead of directly with the `MetricRegistry` itself, simply include the java-commons dependency:

```
    <!-- PIE Frameworks Commons Metrics -->
    <dependency>
        <groupId>com.apple.jvm.commons</groupId>
        <artifactId>commons-dropwizard-metrics</artifactId>
        <version>${version.pie.java.commons}</version>
    </dependency>
```

If this is detected, and no bean of the type `DropwizardMetricsEngine` is present, an instance of `MetricsEngine` that incorporates the supplied `MetricRegistry` and all of the configured reporters, will be exposed as a bean for you to use in your application.

Aluminum Data Modules

Spring Boot provides many out of the box configuration properties for most data sources. The Aluminum Framework provides support for more elaborate configuration, in particular multiple datasources of a given type.

Aluminum Data Cassandra

You can specify multiple connections to Cassandra by simply naming them and providing the connection information. The connection will be bound to the Spring context using the name you provide so ensure that your names are unique. Each connections configuration settings can be set independently. For more information on which properties can be set refer to the Aluminum Config Properties documentation or refer to the **AluminumDataCassandraProperties#CassandraProperties** class in the Aluminum Javadocs.

```
appeng.aluminum.data.cassandra.connections.cassandraFoo.contactPoints=localhost
appeng.aluminum.data.cassandra.connections.cassandraFoo.keyspaces=foo
appeng.aluminum.data.cassandra.connections.cassandraFoo.user=user
appeng.aluminum.data.cassandra.connections.cassandraFoo.password=password

appeng.aluminum.data.cassandra.connections.cassandraBar.contactPoints=remoteHost1,remoteHost2
appeng.aluminum.data.cassandra.connections.cassandraBar.keyspaces=foo,bar
appeng.aluminum.data.cassandra.connections.cassandraBar.user=user
appeng.aluminum.data.cassandra.connections.cassandraBar.password=password
appeng.aluminum.data.cassandra.connections.cassandraBar.useSsl=true

appeng.aluminum.data.cassandra.connections.customPort.contactPoints=remoteHost1,remoteHost2
appeng.aluminum.data.cassandra.connections.customPort.port=9876
appeng.aluminum.data.cassandra.connections.customPort.user=user
appeng.aluminum.data.cassandra.connections.customPort.password=password
appeng.aluminum.data.cassandra.connections.customPort.useSsl=true

appeng.aluminum.data.cassandra.connections.differentPorts.contactPoints=localhost:9042,localhost:9043
appeng.aluminum.data.cassandra.connections.differentPorts.contactPointsContainPorts=true
appeng.aluminum.data.cassandra.connections.differentPorts.user=user
```

```
appeng.aluminum.data.cassandra.connections.differentPorts.password=password
appeng.aluminum.data.cassandra.connections.differentPorts.useSsl=true
```

This configuration will result in four Cluster beans being bound to the context under the names "cassandraFoo" and "cassandraBar", "customPort" and "differentPorts". There will be a Session bound to the name "cassandraFoo.foo" and two Session beans named "cassandraBar.foo" and "cassandraBar.bar".

Note about default Spring Boot Cassandra Configuration

If the Cassandra driver is detected, the default CassandraAutoConfiguration provided by Spring Boot will attempt to connect to localhost on the default Cassandra port even if no configuration properties are provided. This is likely not what you want. To disable the default configuration you can specify the following in your main application class:

```
@SpringBootApplication(exclude=CassandraAutoConfiguration.class)
```

Aluminum Data Kafka

You can specify multiple connections to Kafka by simply naming them and providing the connection information. A **Producer** and a **Consumer** (depending on configuration) will be bound to the Spring context using the name you provide to ensure that your names are unique. Each connections configuration settings can be set independently. A special property called **props** can allow you to pass any Kafka-specific properties to the Producer or Consumer. For more information on which properties can be set refer to the Aluminum Config Properties documentation or refer to the **KafkaProperties** class in the Aluminum Javadocs.

```
appeng.aluminum.data.kafka.connections.kafkaFoo.bootstrapServers=localhost:9092

appeng.aluminum.data.kafka.connections.kafkaFoo.producer.props.key.serializer=org.apache.kafka.common.seria
appeng.aluminum.data.kafka.connections.kafkaFoo.producer.props.value.serializer=org.apache.kafka.common.ser
appeng.aluminum.data.kafka.connections.kafkaFoo.producer.props.request.timeout.ms=2000
appeng.aluminum.data.kafka.connections.kafkaFoo.producer.props.max.block.ms=2000

appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.topics=my-topic, another-topic
appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.props.key.deserializer=org.apache.kafka.common.ser
appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.props.value.deserializer=org.apache.kafka.common.s
appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.props.group.id=my-dev-client
```

This configuration will result in a Producer bean being bound to the context under the name "kafkaFoo.producer", and two Consumer beans (one for each consumer topic) under the names "kafkaFoo.consumer.my-topic", and "kafkaFoo.consumer.another-topic".

Alternatively, you can define "default" Kafka properties that will apply to all producers, or all consumers. This can be helpful if you have to define several consumers with similar properties. You can override the default properties, and provide additional properties for each individual producer and consumer.

```
appeng.aluminum.data.kafka.consumerProps.bootstrap.servers=localhost:9092
appeng.aluminum.data.kafka.consumerProps.auto.offset.reset=earliest
appeng.aluminum.data.kafka.consumerProps.group.id=common-consumer-group-id
appeng.aluminum.data.kafka.consumerProps.enable.auto.commit=true

appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.topics=my-topic, another-topic
appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.props.key.deserializer=org.apache.kafka.common.ser
appeng.aluminum.data.kafka.connections.kafkaFoo.consumer.props.value.deserializer=org.apache.kafka.common.s

appeng.aluminum.data.kafka.connections.kafkaBar.consumer.topics=something-or-other-topic
appeng.aluminum.data.kafka.connections.kafkaBar.consumer.props.key.deserializer=org.apache.kafka.common.ser
appeng.aluminum.data.kafka.connections.kafkaBar.consumer.props.value.deserializer=org.apache.kafka.common.s

appeng.aluminum.data.kafka.connections.kafkaBaz.consumer.topics=whatever-topic
appeng.aluminum.data.kafka.connections.kafkaBaz.consumer.props.key.deserializer=org.apache.kafka.common.ser
appeng.aluminum.data.kafka.connections.kafkaBaz.consumer.props.value.deserializer=com.apple.this.that.or.ar
appeng.aluminum.data.kafka.connections.kafkaBaz.consumer.props.group.id=whatever-group-id
```

Note about Kafka Health Checks

If the aluminum-health module is detected, a health check will be configured for each Producer configured, and a combined health check for all Consumers of the same connection. These Health Checks will attempt to determine the health of Kafka by sending a small message to a configurable health check topic. You can disable these health checks if not desired. If they are enabled, it is advised to pre-create this health check topic with a single partition and a TTL of one hour to minimize the amount of data used on the system. For more information on which properties can be set refer to the Aluminum Config Properties documentation or refer to the **KafkaConsumerProperties** and **KafkaProducerProperties** classes in the Aluminum Javadocs.

Aluminum Data MongoDB

You can specify multiple connections to MongoDB by simply naming them and providing the connection information. The connection will be bound to the Spring context using the name you provide to ensure that your names are unique. Each connections configuration settings can be set independently. For more information on which properties can be set refer to the Aluminum Config Properties documentation or refer to the **AluminumDataMongoDBProperties#MongoDBProperties** class in the Aluminum Javadocs.

```
appeng.aluminum.data.mongodb.connections.mongoFoo.instances=localhost:2181
appeng.aluminum.data.mongodb.connections.mongoFoo.database=mongoFooDB
appeng.aluminum.data.mongodb.connections.mongoFoo.collections=questions,answers
appeng.aluminum.data.mongodb.connections.mongoFoo.user=user
appeng.aluminum.data.mongodb.connections.mongoFoo.password=password

appeng.aluminum.data.mongodb.connections.mongoBar.instances=localhost:2182,localhost:2183,localhost:2184
appeng.aluminum.data.mongodb.connections.mongoBar.database=mongoFooDB
appeng.aluminum.data.mongodb.connections.mongoBar.authDatabase=mongoBarDB
appeng.aluminum.data.mongodb.connections.mongoBar.user=user
appeng.aluminum.data.mongodb.connections.mongoBar.password=password
appeng.aluminum.data.mongodb.connections.mongoBar.ssl.enabled=true
```

This configuration will result in two MongoClient beans being bound to the context under the names "mongoFoo_mongoClient" and "mongoBar_mongoClient". There will be a MongoDatabase bound to the name "mongoFoo" and two MongoCollection beans named "mongoFoo.questions" and "mongoFoo.answers".

Using the Async Drivers

If you want to use either of the MongoDB Async Drivers, use AppEng Composer to update an existing module and add the feature, or select the feature when creating a new module.

With the right dependencies added, you can use the driverType property to indicate which type of driver you wish to use for each connection. (The standard driver is the default if unspecified.)

```
appeng.aluminum.data.mongodb.connections.mongoBaz.instances=localhost:2181
appeng.aluminum.data.mongodb.connections.mongoBaz.database=mongoFooDB
appeng.aluminum.data.mongodb.connections.mongoBaz.driverType=async
appeng.aluminum.data.mongodb.connections.mongoBaz.collections=questions,answers
appeng.aluminum.data.mongodb.connections.mongoBaz.user=user
appeng.aluminum.data.mongodb.connections.mongoBaz.password=password
```

You can mix and match driver types in the same module by selecting all of the driver types in AppEng Composer, and defining multiple connections with different driverType properties.

Special Notes For the Async Driver and a URI Connection String:

1. Due to current limitation in the MongoDB Async Drivers, if you are using a URI instead of the individual properties, the driver will only pick up the default JVM SSL settings, and not the SpringBoot SSL properties. To get around this when enabling SSL, either use the individual connection properties above, or set the keystore/truststore information on startup with the JVM.
2. If using a URI connection string in the MongoDB Async Drivers, the option `streamtype=netty` must be present in the URI in order for SSL to be enabled.

Defining Custom CodecProviders

If you have a custom **CodecProvider** you want automatically configured into your Collection instances, simply define a bean of type **CodecProvider** and Aluminum will apply it to the **MongoDatabase** and **MongoCollection** instances. For example:

```
public class MyAppConfiguration {

    @Bean
    public CodecProvider myCodecProvider() {
        return new MyCodecProvider();
    }
}
```

Enabling MongoDB's PojoCodecProvider

You can have MongoDB's built-in **PojoCodecProvider** applied to all **MongoDatabase** and **MongoCollection** instances automatically by simply specifying which packages MongoDB should scan for POJO classes which should be included. The property is:

```
appeng.aluminum.data.mongodb.pojoCodecProviderPackages=com.apple.myapp.something.model,com.apple.myapp.othe
```

Reading the Oplog

If you've been granted Oplog access by your administrator, the setup is a little different. While the MongoDB Oplog belongs to the database *local*, authentication must be performed against the database *admin* in order to even access it. To define the Oplog collection:

```
appeng.aluminum.data.mongodb.connections.mongoOplog.instances=mongohost:2182
appeng.aluminum.data.mongodb.connections.mongoOplog.database=local
appeng.aluminum.data.mongodb.connections.mongoOplog.authDatabase=admin
appeng.aluminum.data.mongodb.connections.mongoOplog.user=user
appeng.aluminum.data.mongodb.connections.mongoOplog.password=password
appeng.aluminum.data.mongodb.connections.mongoOplog.collections=oplog.rs
appeng.aluminum.data.mongodb.connections.mongoOplog.ssl.enabled=true
```

This will create a MongoCollection bean with the name "mongoOplog.oplog.rs".

Using x.509 Certificates for Authentication

If you need to connect to MongoDB using something other than the default authentication mechanism, you can specify the authMode. Currently, x.509 is the only other authMode supported. Ensure that the client certificate key pair is in your SSL keystore (the keystore alias should be the same as the MongoDB user DN and must match for the private/public keys) and the server certificate is in your SSL trustkeystore and/or from a source trusted by the keystore (for instance an Apple CA cert for a truststore created with the trustcacerts option, the default). For more information on configuring SSL please refer to the Spring Boot Reference Guide. The user should be provided through the user property or encoded in the connection URI:

```
appeng.aluminum.data.mongodb.connections.mongoFoo.instances=localhost:2181
appeng.aluminum.data.mongodb.connections.mongoFoo.database=mongoFooDB
appeng.aluminum.data.mongodb.connections.mongoFoo.user=emailAddress=johnny_appleseed@apple.com,CN=myApp,OU=
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.enabled=true
appeng.aluminum.data.mongodb.connections.mongoFoo.authMode=X509

appeng.aluminum.data.mongodb.connections.mongoTwo.uri=mongodb://emailAddress%3Djohnny_appleseed%40apple.com
appeng.aluminum.data.mongodb.connections.mongoTwo.ssl.enabled=true
```

x.509 and Multiple Certificates

By default, the MongoDB Driver uses the default certificate of the keystore for the x.509 authentication. If you have multiple certificates and you want to use x.509 authentication, you will need to add custom SSL configuration and specify the alias of the cert to use.

```
appeng.aluminum.data.mongodb.connections.mongoFoo.instances=localhost:2181
appeng.aluminum.data.mongodb.connections.mongoFoo.database=mongoFooDB
appeng.aluminum.data.mongodb.connections.mongoFoo.user=emailAddress=johnny_appleseed@apple.com,CN=myApp,OU=
appeng.aluminum.data.mongodb.connections.mongoFoo.authMode=X509
appeng.aluminum.data.mongodb.connections.mongoFoo.sslMode=CUSTOM
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.enabled=true
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.keyAlias=foobar-user-cert
```

```
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.keyStore=/Users/me/some/path/to/my-keystore/keystore-
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.keyStorePassword=fubar
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.trustStore /Users/me/some/path/to/my-truststore/trust
appeng.aluminum.data.mongodb.connections.mongoFoo.ssl.trustStorePassword=blahblahblah
```

Note about default Spring Boot Mongo Configuration

If the MongoDB driver is detected, the default MongoAutoConfiguration provided by Spring Boot will attempt to connect to localhost on the default MongoDB port even if no configuration properties are provided. This is likely not what you want. To disable the default configuration you can specify the following in your main application class:

```
@SpringBootApplication(exclude=MongoAutoConfiguration.class)
```

Aluminum Data SQL

You can specify multiple connections to a SQL database by simply naming them and providing the connection information. The connection will be bound to the Spring context using the name you provide so ensure that your names are unique. Each connections configuration settings can be set independently. For more information on which properties can be set refer to the Aluminum Config Properties documentation or refer to the **AluminumDataSqlProperties#DataSourceProperties** class in the Aluminum Javadocs.

```
appeng.aluminum.data.sql.connections.sqlFoo.url=jdbc:foo:bar
appeng.aluminum.data.sql.connections.sqlFoo.user=foo
appeng.aluminum.data.sql.connections.sqlFoo.password=bar
appeng.aluminum.data.sql.connections.sqlFoo.maxPoolSize=10

appeng.aluminum.data.sql.connections.sqlBar.url=jdbc:foo:baz
appeng.aluminum.data.sql.connections.sqlBar.user=bar
appeng.aluminum.data.sql.connections.sqlBar.password=baz
appeng.aluminum.data.sql.connections.sqlBar.maxPoolSize=10
```

This configuration will result in two DataSource beans being bound to the context under the names "sqlFoo" and "sqlBar".

Aluminum Data ZooKeeper

You can specify multiple connections to ZooKeeper by simply naming them and providing the connection information. The connection will be bound to the Spring context using the name you provide so ensure that your names are unique. Each connections configuration settings can be set independently. For more information on which properties can be set refer to the Aluminum Config Properties documentation or refer to the **AluminumDataZooKeeperProperties#ZooKeeperProperties** class in the Aluminum Javadocs.

```
appeng.aluminum.data.zookeeper.connections.zooKeeperFoo.connectString=localhost:2181

appeng.aluminum.data.zookeeper.connections.zooKeeperBar.connectString=localhost:2182,localhost:2183,localhc
```

This configuration will result in two CuratorFramework beans being bound to the context under the names "zooKeeperFoo" and "zooKeeperBar".

Additional Features

Transaction Tracing Features

Aluminum provides a few different components to assist in tracing transaction from end to end:

Transaction ID Propagation

Aluminum will associate a unique ID with a thread to allow you to correlate log statements within an application and across compatible services. This is accomplished using a web filter on inbound requests and a JAX-RS client filter for outbound requests.

If an inbound request provides the **X-Apple-Txn-ID** header the value will be used, otherwise one will be generated. The Transaction ID is stored in the Slf4J MDC and can be accessed using the **txnId** key, for example:

```
%X{txnId}
```

By default, outbound requests will add the **X-Apple-Txn-ID** header to any request sent through a client retrieved from the RestClientFactory.

Transaction Logging

You can enable web transaction logging by setting the following property:

```
appeng.aluminum.web.transactionLoggingFilter.enabled=true
```

With Transaction Logging enabled you will see entries in the log file similar to the following:

```
2016-01-27T14:48:42.936-0800 INFO  : loggerName="c.a.i.a.a.w.TransactionLoggingFilter"
threadName="qtp620030769-27" txnId="98c3a269-8199-4669-96c5-da2ab7650808" event="requestActivity",
appName="sample", clientAppId="", clientIP="0:0:0:0:0:0:0:1", serverIP="hostname/127.0.0.1",
httpPort="8443", accept="text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
contentType="", contentLength="-1", requestURI="/loggedout.html", httpMethod="GET"
2016-01-27T14:48:42.943-0800 INFO  : loggerName="c.a.i.a.a.w.TransactionLoggingFilter"
threadName="qtp620030769-27" txnId="98c3a269-8199-4669-96c5-da2ab7650808" event="responseActivity",
appName="sample", contentType="text/html; charset=UTF-8", httpStatus="200"
```

Transaction Metrics Timing

Timing for JAX-RS endpoint methods is enabled by default. This will result in entries in the metrics log like the following (for a method named currentUserSessionDetails):

```
2016-01-27T15:03:47.375-0800 INFO  : type=TIMER, name=common.transaction.all.response.time, count=14,
min=0.401759, max=703.853524, mean=142.1461904758101, stddev=168.11848946947015, median=111.56903,
p75=128.437469, p95=703.853524, p98=703.853524, p99=703.853524, p999=703.853524,
mean_rate=0.012606044691885933, m1=4.5169809631969716E-9, m5=0.0013335513581467472,
m15=0.004754945923813472, rate_unit=events/second, duration_unit=milliseconds
2016-01-27T15:03:47.375-0800 INFO  : type=TIMER,
name=common.transaction.currentUserSessionDetails.response.time, count=2, min=0.401759, max=15.979989,
mean=8.190873999999999, stddev=7.789115, median=15.979989, p75=15.979989, p95=15.979989, p98=15.979989,
p99=15.979989, p999=15.979989, mean_rate=0.0018575979659885573, m1=7.1968380636597334E-9,
m5=0.011299945248502781, m15=0.12182379022156752, rate_unit=events/second, duration_unit=milliseconds
```

Appendix A - Configuration

Spring Boot provides many configuration properties that can be used to control the behavior of your application. For more information refer to the Spring Boot Reference Guide.

Aluminum also provides many configuration properties. You can find a list of them in the Aluminum Config Properties documentation.

Appendix B - Samples

Aluminum provides many sample applications which also double as integration tests. See Aluminum Samples for more information.

Appendix C - Contributing

Visit the Contributor Guide for more information on contributing.