Calculation Module

Overview

The Calculation module provides a framework for OpenMRS developers to author from simple to complex calculations to produce a particular piece of data for a single patient or a cohort of patients. It is a high level abstraction of rules/calculations, simple in that it provides a minimal number of useful concrete implementations of its core components that can be used out of the box but providing a much more flexible tool for authoring patient centered calculations.

Downloads

- Download the omod from the module repository: https://addons.openmrs.org/#/show/org.openmrs.module.calculation
- View/download the source code on github: https://github.com/openmrs/openmrs-module-calculation

Technical Overview

The module is designed around a couple of concepts described below:

PatientCalculation

A PatientCalculation represents a definition that can be evaluated to produce patient data. A PatientCalculation can expose parameters which control the results of its calculation. All calculations implement the super interface Calculation. It is highly recommended for calculations to extend BaseCalculation rather than directly implementing Calculation Interface.

The PatientCalculation interface exposes the method `getParameterDefinitionSet()` which should return a set of parameter definitions exposed by the calculation.

⚠️ Calculations are stateful and non-singleton, therefore it is good practice to always get a new instance of the calculation from the CalculationProvider, if you wish to share certain information between calculations, use the PatientCalculationContext.

The PatientCalculation interface exposes the method below:

`evaluate(Collection<Integer> cohort, Map<String, Object> parameterValues, PatientCalculationContext context)`: This where all the logic for the PatientCalculation goes.

⚠️ Evaluate implementations must account on the fact that they may be executed by PatientCalculationService in manageable batches e.g. when the service is passed 10,000 patients, the evaluate method will be called 10 times and passed 1,000 patients each time.

Examples:

HelloWorldCalculation.java

```java
public class HelloWorldCalculation extends BaseCalculation implements PatientCalculation {
    @Override
    public CohortResult evaluate(Collection<Integer> cohort, Map<String, Object> parameterValues, PatientCalculationContext context) {
        CohortResult results = new CohortResult();
        if (cohort != null) {
            PatientService ps = Context.getPatientService();
            for (Integer patientId : cohort) {
                Patient patient = ps.getPatient(patientId);
                if (patient != null) {
                    results.put(patientId, new SimpleResult("Hello World from " + patient.getPersonName().getFullName(), this, context));
                }
            }
        }
        return results;
    }
}
```

HelloWorldConfigurableCalculation.java
public class HelloWorldCalculation extends BaseCalculation implements PatientCalculation, ConfigurableCalculation {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void setConfiguration(String configuration) throws InvalidCalculationException {
        if (configuration == null)
            throw new InvalidCalculationException("The name is required");
        name = configuration;
    }

    @Override
    public CohortResult evaluate(Collection<Integer> cohort, Map<String, Object> parameterValues,
                                  PatientCalculationContext context) {
        CohortResult results = new CohortResult();
        if (cohort != null) {
            PatientService ps = Context.getPatientService();
            for (Integer patientId : cohort) {
                Patient patient = ps.getPatient(patientId);
                if (patient != null) {
                    results.put(patientId, new SimpleResult("Hello World from " + patient.getPersonName().
                        getFullName(),
                        this, context));
                }
            }
        }
        return results;
    }
}

CalculationProvider

A CalculationProvider is responsible for retrieving a Calculation instance given a calculation name and an optional configuration string.

Note: CalculationProviders have to be registered as spring beans so as to be found by the framework. A calculationProvider should typically invoke the setConfiguration() method of the calculation instance before returning it in case it is a subclass of a ConfigurableCalculation. The module comes with a ClasspathCalculationProvider that can return a Calculation given its fully qualified class name and is on the classpath, its is limited to Calculations that have constructors that require no arguments.

A typical implementation for a calculation provider would be such that the calculationName is the calculation class to instantiate and configuration represents the serialized property values that need to be configured on this calculation instance, however it is totally up to the Provider to define this. For example, to retrieve the Calculation for "Most Recent Weight":

Calculation Name: org.openmrs.calculation.definition.MostRecentObsCalculation
Configuration: concept=<UUID for Weight (KG) concept>

There would then be a CalculationProvider registered to handle this type of Calculation which would know it needed to first instantiate a new instance of MostRecentObsCalculation, configure it's properties via the parsed values from the configuration string, and then return configured Calculation instance. CalculationProviders will likely be wired to Calculation classes either via a registry or via annotations.

A CalculationProvider is responsible for retrieving a Calculation instance given a calculation name and an optional configuration string. A typical implementation would be such that Calculation name is the Calculation class to instantiate or just a logical name that uniquely identifies a single Calculation from all the provider provides and the configuration represents the serialized property values that need to be configured on this Calculation instance, however it is totally up to the Provider to define this. For example, to retrieve the Calculation for "Most Recent Weight":

Calculation Name: org.openmrs.calculation.definition.MostRecentObsCalculation
Configuration: concept=<UUID for Weight (KG) concept>

There would then be a calculation provider registered to handle this type of calculation which would know how to create a new instance of MostRecentObsCalculation, configure it's properties via the parsed values from the configuration string, and then return configured Calculation instance.
Example: TestCalculationProvider.java (This could be registered as a spring bean)

```java
public class TestCalculationProvider implements CalculationProvider {
    // Just for testing purposes, this provider has an in-memory source of calculations in form of a map,
    // typically it should be from anywhere known by the provider e.g from the DB
    private Map<String, Class<? extends Calculation>> calculations = new HashMap<String, Class<? extends Calculation>>();

    public TestCalculationProvider() {
        // This could be map of fully qualified/unique logical names and Classes
        calculations.put("HelloWorldCalculation", HelloWorldCalculation.class);
    }

    @Override
    public Calculation getCalculation(String calculationName, String configuration) {
        if (calculationName != null) {
            Class<? extends Calculation> clazz = calculations.get(calculationName);
            if (clazz != null) {
                try {
                    Calculation calculation = clazz.newInstance();
                    if (StringUtils.isNotBlank(configuration)) {
                        // do further initialization.............
                    }
                    return calculation;
                } catch (Exception e) {} 
            }
        } 
        // If this is a ConfigurableCalculation, try to configure it
        if (calculation instanceof ConfigurableCalculation) {
            ((ConfigurableCalculation)calculation).setConfiguration(configuration);
        } else {
            if (StringUtils.isNotBlank(configuration)) {
                throw new InvalidCalculationException(this, calculationName, configuration);
            }
        }
        return null;
    }
}
```

**CalculationResult**

A CalculationResult is the data that is produced from evaluating a Calculation for a single patient. CalculationResults are strongly typed, but provide a convenience method for casting to other appropriate datatypes. The module provides a SimpleResult that can be used out of the box though one could extend it to add extra features. There is also a DateBaseResult interface that can be implemented for domain objects that have a date of occurrence e.g Encounter has encounterDatetime. The module ships with the following date based results; EncounterResult, ObsResult.

**ParameterDefinition**

This is an abstraction of the parameters that a calculation exposes, they encapsulate information about the values against which a calculation can be evaluated. These can be optional or required. All parameter definitions should extend the class ParameterDefinitionSet which is technically a wrapper for a HashSet.

Say you have a Calculation that requires some sort of user input e.g An AgeCalculation that requires the user to specify the units in which to express the patient age i.e. months Vs years, a ParameterDefinition could be useful encapsulate certain attributes. For instance, to specify whether it is required, the datatype of the allowed values plus the name and description to display in the user for the user. The key property is intended to be used only programmatically.

The module comes with SimpleParameterDefinition as basic implementation of the interface and should be good enough for most use cases.

You can add parameters by calling:
myAgeCalculation.addParameterDefinition(new SimpleParameterDefinition("units", "java.lang.String", "Months", true));

PatientCalculationContext

The CalculationContext contains any contextual information that may be shared across one or more Calculation evaluations. This includes the date to base on when evaluating the calculation and a cache for storing the results for previously evaluated calculations. It should essentially replace any call for "new Date()" in evaluation code. To can create your own CalculationContext class by implementing the interface CalculationContext.

To obtain a PatientCalculationContext instance, call PatientCalculationService.createCalculationContext() how it all comes together

First you need an instance of a CalculationProvider, query the provider for the calculation you are interested in. There are 2 ways to evaluate a calculation as shown below:

1. Calling the evaluate method of the calculation instance

Example:

```java
PatientCalculation calculation = (PatientCalculation)new TestCalculationProvider().getCalculation(
    "HelloWorldCalculation", null);
Collection<Integer> cohort = new ArrayList<Integer>();
    cohort.add(6);
    cohort.add(7);
CohortResult result = calculation.evaluate(cohort, null, null);//you can specify parameter values and a CalculationContext

for (Map.Entry<Integer, CalculationResult> entry : cohortResult.entrySet()) {
    System.out.println("Patient with id:"+entry.getKey()+" says " +entry.getValue().asType(String.class));
}
```

2. Using the PatientCalculationService

By calling one of the evaluate methods in the PatientCalculationService, the evaluate method in the calculation service is overloaded. You can call the appropriate method and you have to specify the patient or cohort of patients, a calculation, an optional map of parameter values and a PatientCalculationContext. This is the recommended way to evaluate a calculation because the service does the extra job of enforcing the required parameters and parameter datatypes. It provides convenience methods for performing evaluations for a single patient, creates the CalculationContext under the hood if none is provided and also performs batch processing for cohorts of more than 1000 members.

Example:

```java
public void sayHelloWorld() {
    PatientCalculation calculation = (PatientCalculation)new TestCalculationProvider().getCalculation(
        "HelloWorldCalculation", null);
    Collection<Integer> cohort = new ArrayList<Integer>();
    cohort.add(6);
    cohort.add(7);
    CohortResult cohortResult = Context.getService(PatientCalculationService.class).evaluate(cohort, calculation);

    for (Map.Entry<Integer, CalculationResult> entry : cohortResult.entrySet()) {
        System.out.println("Patient with id:"+entry.getKey()+" says " +entry.getValue().asType(String.class));
    }
}
```

About TokenRegistrations

A TokenRegistration represents a saved Calculation instance in the database, and includes a unique name, the CalculationProvider, the calculationName, and the configuration for the calculation. The intention is to allow a fully configured Calculation instance to be retrieved given a unique name String.

Example: Getting a Calculation instance of a TokenRegistration, assuming you have a saved token
TokenRegistrationService service = Context.getService(TokenRegistrationService.class);
MySavedCalculation calculation = service.getCalculation(myTokenName, MySavedCalculation.class);

**Release Notes**

- **1.0 (Latest major release) - rev.29203**
  - Download module [here](#)
  - Changes since 0.9
    - CALC-38 - PatientCalculation.evaluate method should take a Collection<Integer> instead of a Cohort
    - CALC-32 - Made change to clear the token cache on save and delete so that it gets rebuild
    - CALC-39 - Rename CohortResult to CalculationResultMap
    - CALC-41 - Calculation maven artifacts have a non-standard groupid
    - CALC-44 - ListResults should have a getValues() method
    - CALC-45 - ListResult should refer to Calculation and CalculationContext, not their Patient variants
    - CALC-43 - ResultUtil.convert and CalculationUtil.cast do not correctly convert things to Boolean
    - CALC-46 - Utility methods on CalculationResultMap for getting results as boolean or isEmpty without needing null tes

- **0.9 (RC)**
  - Got everything working taking into consideration all use cases and design described [here](#)