**DEVELOPMENT GUIDELINE**

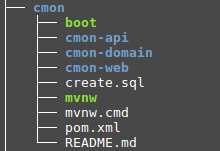
Software Framework being used in this Project is a combination of various OSS (Open Source Software) technologies around [Spring Framework](http://projects.spring.io/spring-framework/)

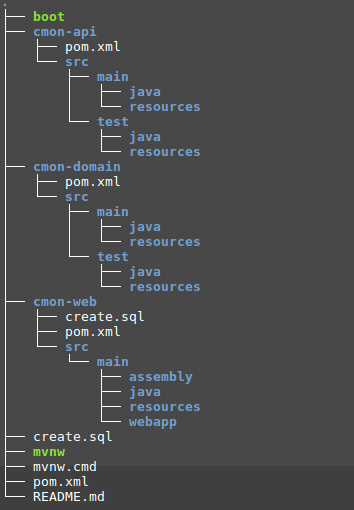
1. **Create new project:**

* Clone parent project at <https://github.com/Caltalys/lgsp-parent>, run “mvn install”
* Clone parent project at <https://github.com/Caltalys/lgsp.fw.core>, run “mvn clean generate-sources install”
* Clone sample project at <https://github.com/Caltalys/lgsp.fw.app.cmon>, rename folder, package, project maven info as you wish.

1. **Project structure**

**-** Recommended structure of project is as below (sample **cmon** project):





1. **Web module:** Module that manages the application layer (Web layer) components.

* Controller class
* Validator class for relational check
* View (JSP)
* ViewModel (ZK)
* Web resource (CSS, JavaScript file)
* Bean definition file for defining the application layer components
* Web application configuration file
* Message definition file

1. **Domain module:** Module that manages the domain layer components (entity, repository, business-service, adapter-service...).

* Domain object such as Entity
* Repository
* Service
* DTO
* JUnit for the domain layer components
* Bean definition file for defining the domain layer components

1. **Api module:** Module that manages the web service layer components if project required (RESTful/SOAP).

* Rest controller class
* Rest resource
* JUnit for the domain layer components
* Bean definition file for defining the domain layer components

1. **Domain Layer Implementation**

Domain layer implements business logic to be provided to the application layer. It is also responsible for storing the data permanently (location where data is stored such as RDBMS, NoSQL etc.), It integrates a data store other than the database (such as messaging system, Key-Value-Store, Web service, existing legacy system, external system etc.) as well as transmission of messages.

Implementation of domain layer is classified into the following.

|  |  |  |
| --- | --- | --- |
| **No.** | **Classification** | **Description** |
| 1. | Implementation of Entity | Creation of classes (Entity class) to hold business data. |
| 2. | Implementation of Repository | Implementation of the methods to operate on business data. These methods are provided to Service classes.  These are in particular the CRUD operations on Entity object. |
| 3. | Implementation of Service | Implementation of the methods for executing business logic. These methods are provided to the application layer.  Business data required by the business logic is fetched as the Entity object through the Repository. |

* 1. **Implementation of Entity**

Create an Entity using the following method.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Supplementary** |
| 1. | Create Entity class for each table. | Entity class is not required for mapping tables which represent the relationship between the tables.  Further, when the tables are **not normalized**, Entity class for each table rule may not be applicable. |
| 2. | When there is a FK (Foreign Key) in the table, the Entity class of FK destination table must be defined as one of the properties of this Entity. | When there is 1:N relationship with FK destination table, use either **java.util.List<E>** or **java.util.Set<E>**.  The Entity corresponding to the FK destination table is called as the related Entity in this guideline. |
| 3. | Treat the code related tables as **java.lang.String** rather than as an Entity. | Code related tables are to manage the pairs of code value and name.  When there is a need to bifurcate the process as per code values, **enum** class corresponding to code value should be created and it must be defined as property. |

**Example:**

<https://github.com/Caltalys/lgsp.fw.app.cmon/blob/master/src/main/java/vn/lgsp/fw/app/cmon/domain/entity/CmonDonViHanhChinh.java>

* 1. **Implementation of Repository**

Repository has following 2 roles.

- To provide to Service, the operations necessary to control Entity lifecycle (Repository interface - CRUD operations).

- To provide persistence logic for Entity (implementation class of Repository interface).

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Class(Interface)** | **Role** | **Description** |
| 1 | Repository interface | Defines methods to control Entity lifecycle required for implementing business logic (Service). | Defines methods for CRUD operations of the Entity and is not dependent on persistence layer.  Repository interface belongs to the domain layer since it plays the roles of defining the operations on Entity required for implementing business logic (Service). |
| 2 | RepositoryImpl | Implements the methods defined in Repository interface. | Implements CRUD operations of the Entity and is dependent on persistence layer. Performs actual CRUD processes using API that performs persistence provided by Spring Framework, O/R Mapper and middleware.  RepositoryImpl belongs to infrastructure (data) layer since it plays the role of implementing the operations defined in Repository interface. |

The most important purpose of creating Repository is not to exclude the persistence logic from business logic. The most important purpose is to limit the implementation scope of business logic (Service) to the implementation of business rules. This is done by separating the operations to access business data in Repository. As an outcome of this, persistence logic gets implemented in Repository instead of business logic (Service).

Repository must be created using the following policy only.

|  |  |  |
| --- | --- | --- |
| No. | Method | Supplementary |
| 1. | Create Repository for the main Entity only. | This means separate Repository for operations of related Entity is not required.  However, there are case when it is better to provide Repository for the related Entity in specific applications (for example,  application having high performance requirements etc). |
| 2. | Place Repository interface and RepositoryImpl in the same package of domain layer. | Repository interface belongs to domain layer and RepositoryImpl belongs to infrastructure layer. However,  Java package of RepositoryImpl can be same as the Repository interface of domain layer. |
| 3. | Place DTO used in Repository in the same package as Repository interface. | For example, DTO to store search criteria or summary DTO for that defines only a few items of Entity. |

**Example:**

<https://github.com/Caltalys/lgsp.fw.app.cmon/tree/master/cmon-domain/src/main/java/vn/lgsp/fw/app/cmon/domain/repository>

#### **Method definition of Repository interface:**

|  |  |  |
| --- | --- | --- |
| **No.** | **Types of methods** | **Rules** |
| 1 | Method for searching a single record | - Method name beginning with **findOneBy** to indicate that this method fetches a single record that matches with the condition.  In the method name after “findOneBy”, physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate “the kind of entity that can be fetched using this method”.  - There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided.  - Return value must be Entity class. |
| 2 | Method for searching multiple records | - Method name beginning with **findAllBy** to indicate that this method fetches all the records that matches with the condition.  In the method name after “findAllBy”, physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate “the kind of entity that can be fetched using this method”.  - There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided.  - Return value must be collection of Entity class. |
| 3 | Method for searching multiple records with pagination | - Method name beginning with **findPageBy** to indicate that this method fetches pages that matches with the condition.  In the method name after “findPageBy”, physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate “the kind of entity that can be fetched using this method”.  - There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided. Pageable provided by Spring Data should be the interface for pagination information (start position, record count, sort information).  - Return value should be Page interface provided by Spring Data. |
| 4 | Count related method | - Method name beginning with **countBy** to indicate that this method fetches count of Entities which matches with the condition.  - In the method name after “countBy”, physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate “the kind of entity that can be fetched using this method”.  - There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided.  - Return value must be long type. |
| 5 | Method for existence check | - Method name beginning with **existsBy** to indicate that this method checks the existence of Entity which matches with the condition.  - In the method name after “existsBy”physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate “the kind of entity that can be fetched using this method”.  - There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided.  - Return value must be boolean type. |

**Example**:

**public class Todo** **extends** BaseEntity<Todo> {  
 **private** String todoTitle;  
 **private** boolean finished;  
 **private** Date createdAt;  
 *// ...*  
 }

**public interface** **TodoRepository** **extends** BaseRepository<Todo, Long> {  
 *// (1)*  
 Todo findOneByTodoTitle(String todoTitle);  
 *// (2)*  
 List<Todo> findAllByUnfinished();  
 *// (3)*  
 Page<Todo> findPageByUnfinished();  
 *// (4)*  
 long countByExpired(int validDays);  
 *// (5)*  
 boolean existsByCreateAt(Date date);  
}

**Note:** These rules should be applied to method definition of Service too.

Example:

<https://github.com/Caltalys/lgsp.fw.core/blob/master/src/main/java/vn/lgsp/fw/core/BaseService.java>

* 1. **Implementation of Service**

Service plays the following 2 roles.

1. **Provides business logic to Controller.**

- Business logic consists of create, update, consistency check etc of business data as well as all the processes related to business logic.

- Create and update process of business data should be delegated to Repository(or O/R Mapper) and service should be limited to implementation of business rules.

**Note:**

**Regarding distribution (separation) of logic between Controller and Service:** The logic to be implemented by Controller and Service should be followed as per the rules given below.

1. For the data requested from the client, single item check and correlated item check is to be performed in Controller (*Bean Validation or Spring Validator*).
2. Conversion processes (*Bean conversion, Type conversion and Format conversion*) for the data to be passed to Service, must be performed in Controller instead of Service.
3. Business rules should be implemented in Service. Access to business data is to be delegated to Repository or O/R Mapper.
4. Conversion processes (*Type conversion and Format conversion*) for the data received from Service (data to respond to the client), must be performed in Controller.
5. **Declare transaction boundary**

- Declare transaction boundary when business logic is performing any operation which requires ensuring data consistency (mainly data update process).

- Even in case of logic that just read the data, often there are cases where transaction management is required due to the nature of business requirements. In such cases, declare transaction boundary.

- Transaction boundary must be set in Service layer as a principle rule. If it is found to be set in application layer (Web layer), there is a possibility that the extraction of business logic has

not been performed correctly.

1. **Application Layer Implementation**
   1. **Web application Implementation**
   2. **Web service Implementation**
2. **Security**
3. **Security setup:**

* Add security dependency (*pom.xml*)



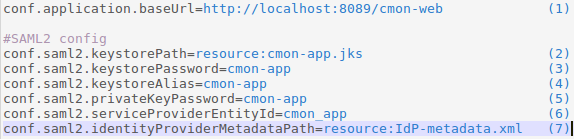
* Create keystore for application

keytool -genkey -alias <app\_alias> -keyalg RSA -keystore <app\_keystore>.jks -keysize 2048

* Download & Import Identity Provider’s certificate to application keystore

keytool -import -trustcacerts -alias <alias\_for\_IdP> -file <IdP\_certificate> -keystore <app\_keystore>.jks

* Add SAML2 properties (*application.properties*)





1. Application’s URL (context path)
2. Path to keystore
3. Keystore password
4. Keystore alias
5. Private key of keystore
6. Service provider entity Id
7. Identity provider Metadata path

* Add SAML2 config for application: The configuration (*org.pac4j.core.config.Config*) contains all the clients and authorizers required by the application to handle security.



* Add Spring security config: You can protect (authentication + authorizations) the urls of your Spring Security application by using the *SecurityFilter*



For indirect clients (SSO), the user is redirected to an external identity provider for login and then back to the application. Thus, a callback endpoint is required in the application. It is managed by the *CallbackFilter*



* For more configuration, check following document:

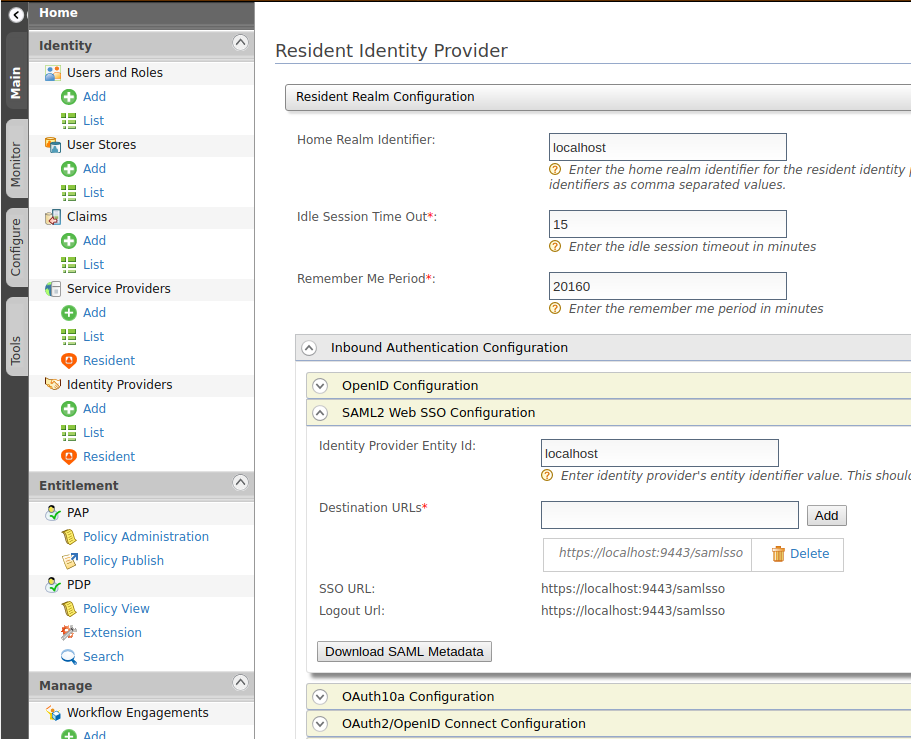
[Spring Security](https://docs.spring.io/spring-security/site/docs/4.2.6.RELEASE/reference/html/)

[Pac4j Spring Security](https://github.com/pac4j/spring-security-pac4j)

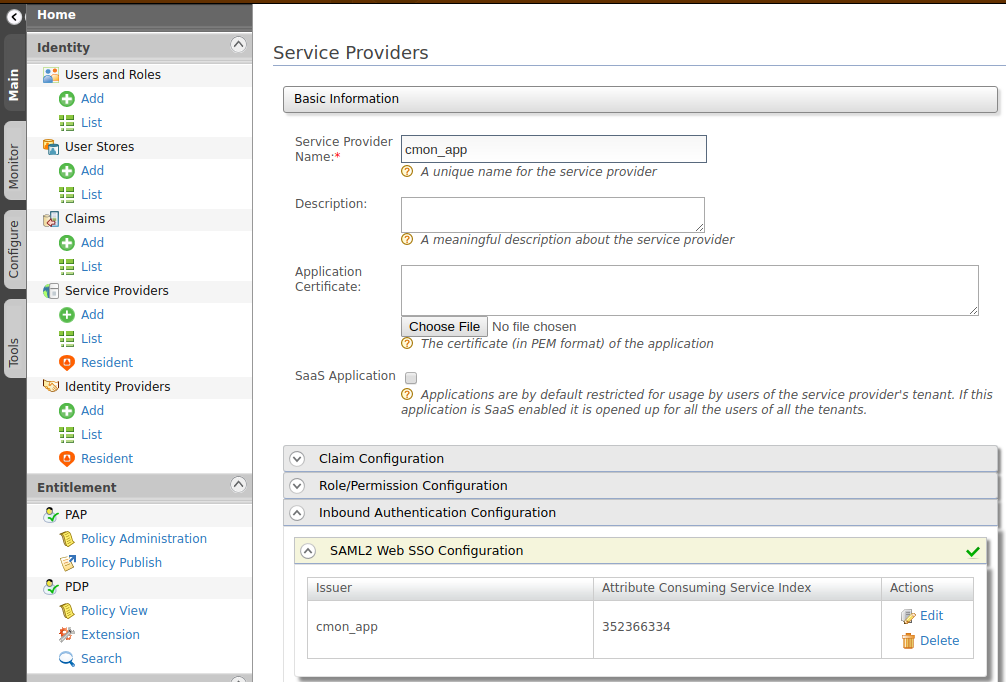
[Pac4j SAML2](https://www.pac4j.org/docs/clients/saml.html)

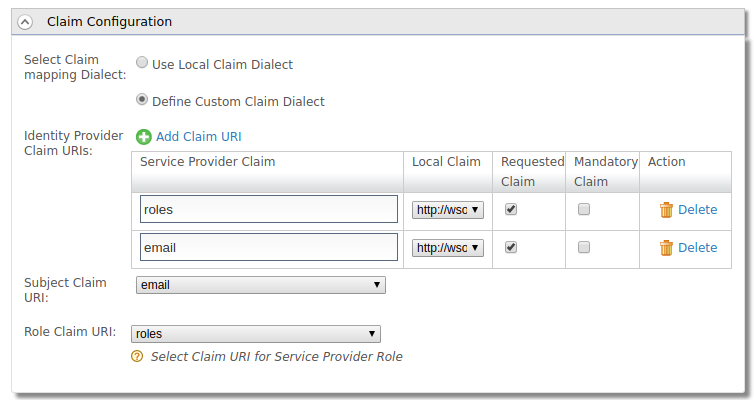
1. **Setup Identity Provider for Saml2 SSO:**

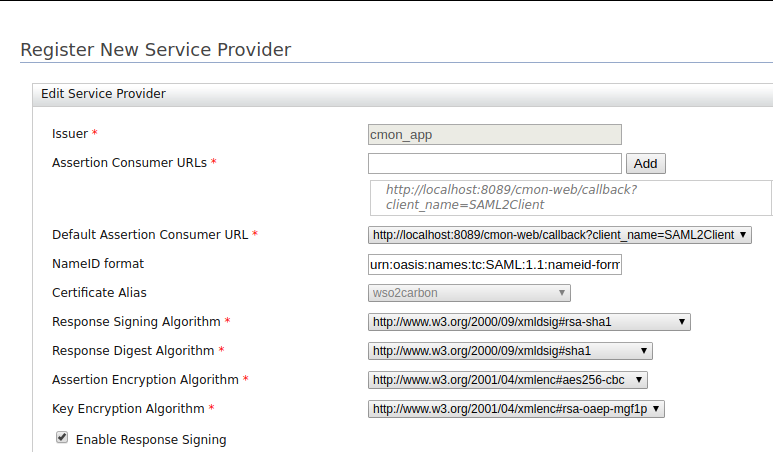
* Get IdP (Identity Provider) metadata:
  + In WSO2 Identity Server, open Main menu, chose Identity Providers > Resident.
  + Inbound Authentication Configuration > SAML2 Web SSO Configuration > Download SAML Metadata
  + After download the Idp Metadata, put it to the place where you config the ***Identity provider Metadata path*** in application above

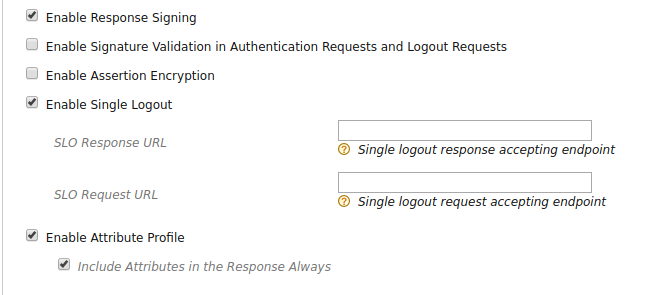


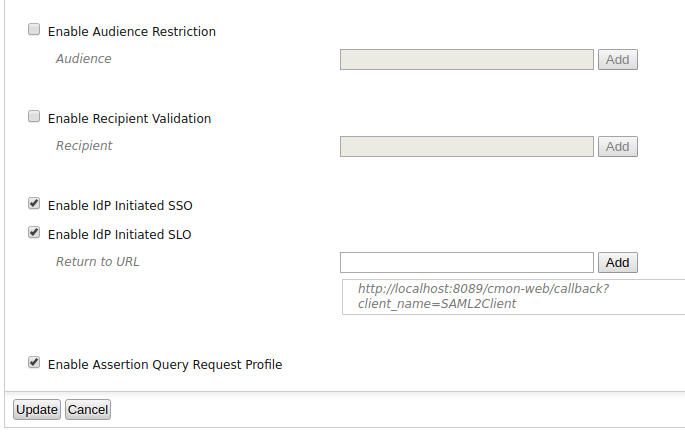
* Create & Configure Service Provider (Registering Application for SSO)
  + In WSO2 Identity Server, open Main menu, chose Service Providers > Add.
  + Filling all the required information
  + Define the claim information the the Idp the return to client when authentication success
  + Create SAML SSO for Inbound Authentication











1. **Authorization:**
2. **dsfdfd**

**APPENDIX**