European Commission

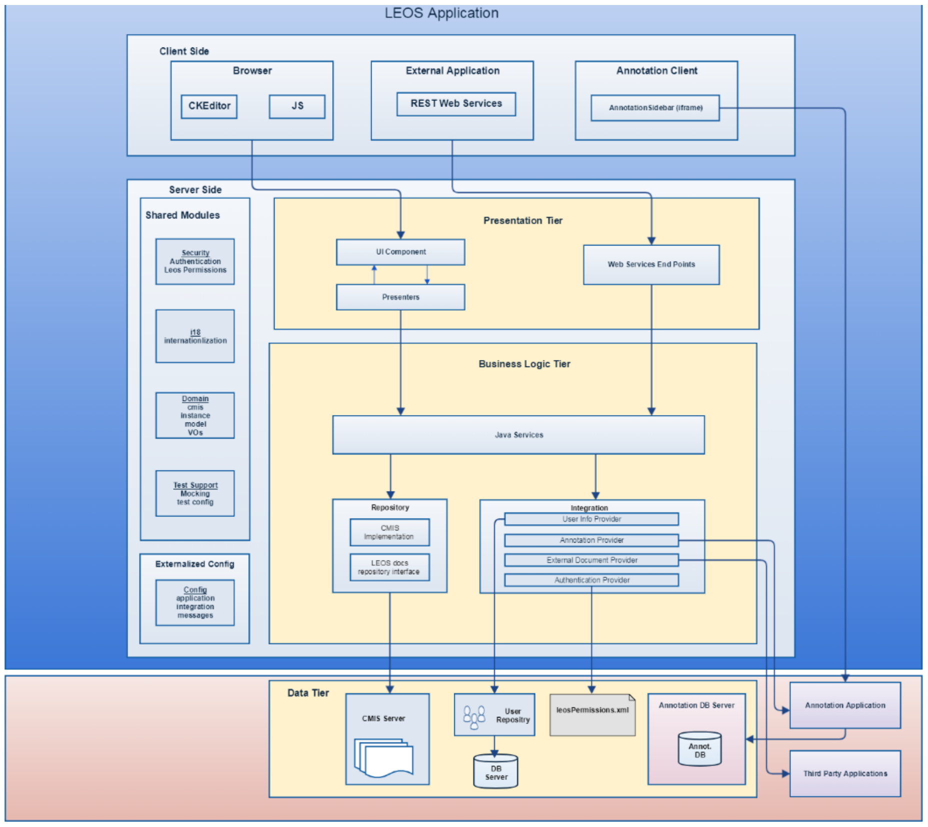
LEOS

Application Architecture

Leos version:	3.0
Modified on:	06.04.2020
Version of the manual:	1.0

1. Leos Component Structure

Leos release package is divided into 3 major tiers: presentation, business logic and data tier.



1.1 Leos Application Architecture

The Leos application architecture is spread across two tiers: presentation and business logic tier.

The **presentation tier** includes:

- GUI components (web-presentation) – groups all the UI related components (screens, components, windows)
- Presenters (web-presentation) – encapsulates all the logic for web-actions: calls specific services from business logic layer and prepares the required data for the next GUI component

- WS Entry-points (WebServices) – validates the input data and calls specific BusinessObjects to execute the actual logic if the input is valid and the caller has access to the requested function
- Business Objects (WebServices) – calls specific services from business logic layer and prepares the required data for the response (if any)

The **business logic tier** includes:

- Services –it encapsulates all logic for handling various requests
- Repositories – hold the read/write access to all the data used by the application
- Integration – encapsulates all the logic for accessing third party applications

The **data tier** includes:

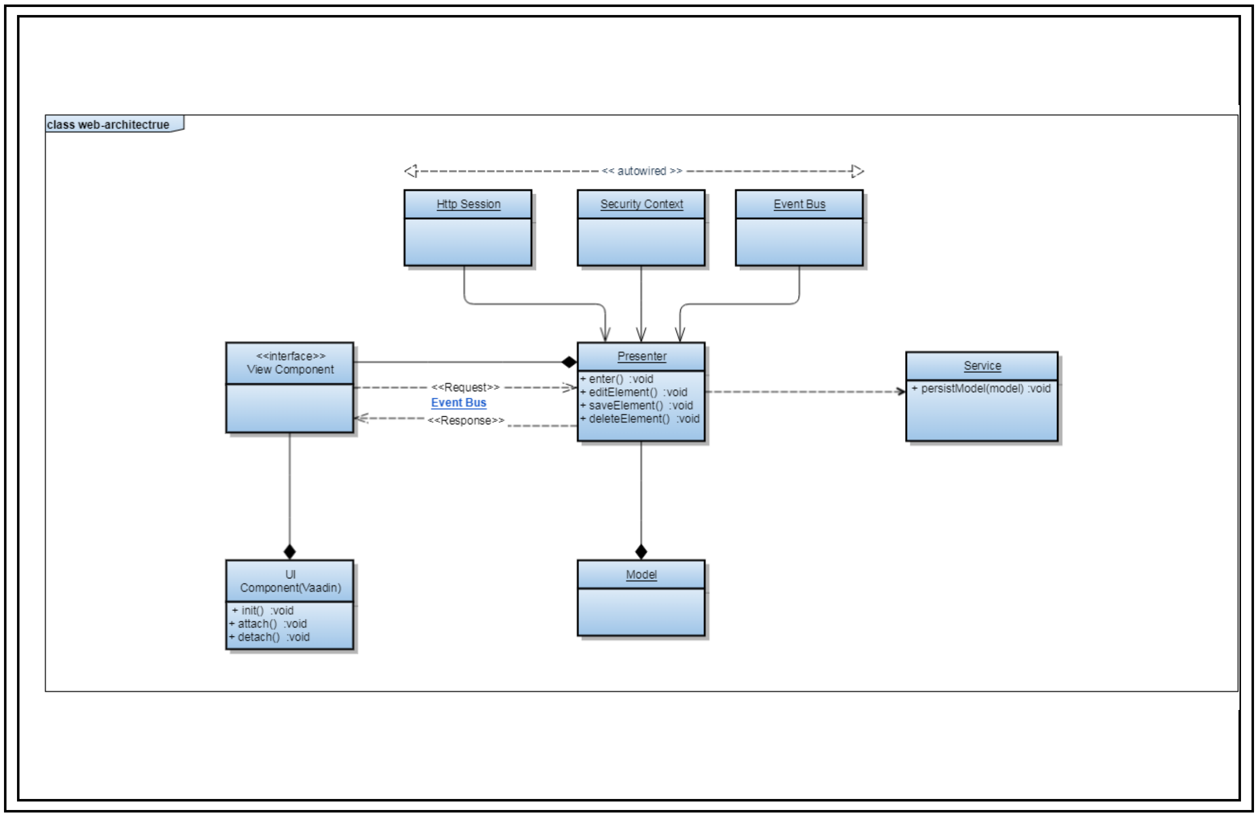
- User Repo DB – holds all the data required for the application to run (users, configurations, documents metadata, etc).
- CMIS Server – holds the documents content.
- leosPermissions.xml - this defines the mapping between leos application roles and permissions.
- Annotation DB server - holds the annotation content.

1.2 Technologies used:

- Vaadin 8
- Spring for wiring
- Spring data - jpa
- Hibernate
- CMIS
- WS-Provider: JRC standard for REST
- Mockito
- CkEditor
- AngularJS - annotation client

2. Web Component Architecture

The web is using the MVP (Model-View-Presenter) pattern, the Supervising Controller variant. The below diagram depicts the interactions that exist in Leos using the MVP-SC strategy:



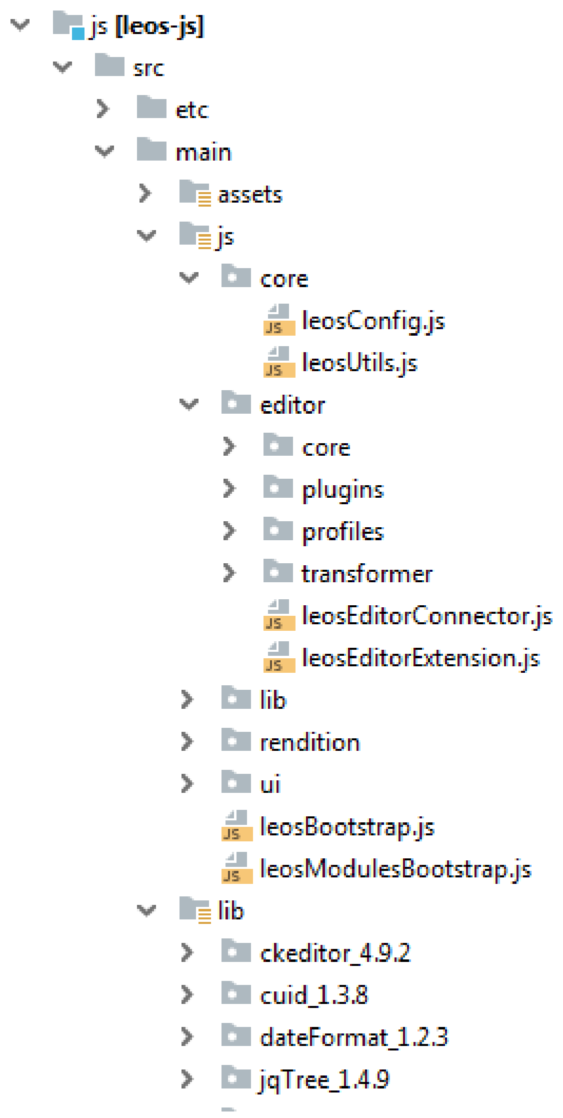
The purpose is to decouple the building of the view from the logic around the view. Moving all the logic of handling user events inside the controller makes the Leos implementation scalable, ensures a clear separation of concerns, and facilitates unit-testing of the behavior of the view.

2.1 JavaScript

The *Document* edition is done using CkEditor. Besides the CkEditor library, we are providing custom plugins for CKEditor in order to provide additional features that fit the purpose of AkomaNtoso XML.

2.1.1 Structure

The JavaScript files for CKEditor edition feature is part leos-js module and is present in *war.*



```
js [leos-js]
  src
    etc
    main
      assets
      js
        core
          leosConfig.js
          leosUtils.js
        editor
          core
          plugins
          profiles
          transformer
          leosEditorConnector.js
          leosEditorExtension.js
        lib
        rendition
        ui
        leosBootstrap.js
        leosModulesBootstrap.js
      lib
        ckeditor_4.9.2
        cuid_1.3.8
        dateFormat_1.2.3
        jqTree_1.4.9
```

In *leos-js* module, we have:

- lib folder – consists of all the distributions used out of the box for e.g. (CKEditor, jQuery, requirejs, stampit, jsTree, lodash etc.);

- profiles folder – consists of the files used to configure/create different editing profiles for the akn elements that can be modified using CKEditor (eg, for Article editing profile is inlineAknArticle.js);

- transformer – encapsulates all the files related to akn-to-html transformations;

- core – consists of the common logic used across the editor;

- plugins folder – encapsulates all the custom plugins specifically designed for edition functionality for akn elements like: paragraphs, subparagraphs, points, alinea etc. It also consists of additional plugins like: table support, cross-references, authorial note, comments, highlights etc.;

- test folder – consists of all the jasmine spec tests (Integration and unit);

- leosBootstrap.js – Implementation of CKEditor and all the javascript bootstrap logic;

- leosMouleBootstrap.js – Additional configuration to be passed to specified modules;

2.1.2 Implementation strategy

There are two main areas where implementation is done: *Plugin* and *Transformer*.

- <u>Plugin</u> implementation works as plug-n-play. If the plugin is loaded in the profile the feature will be available for the editor. There are two types of plugins used in Leos.
  - CKEditor internal plugins, which comes with the distribution of CKEditor library.
  - Custom plugins, which are implemented to support the edition and transformation from akn to html and vice-versa. The plugin consists of transformation configuration used by the transformer and additional logic.

- <u>Transformer</u> encapsulates the logic to transform an element from akn-to-html and vice-versa based on the transformation configuration provided to it as input from the custom plugins.

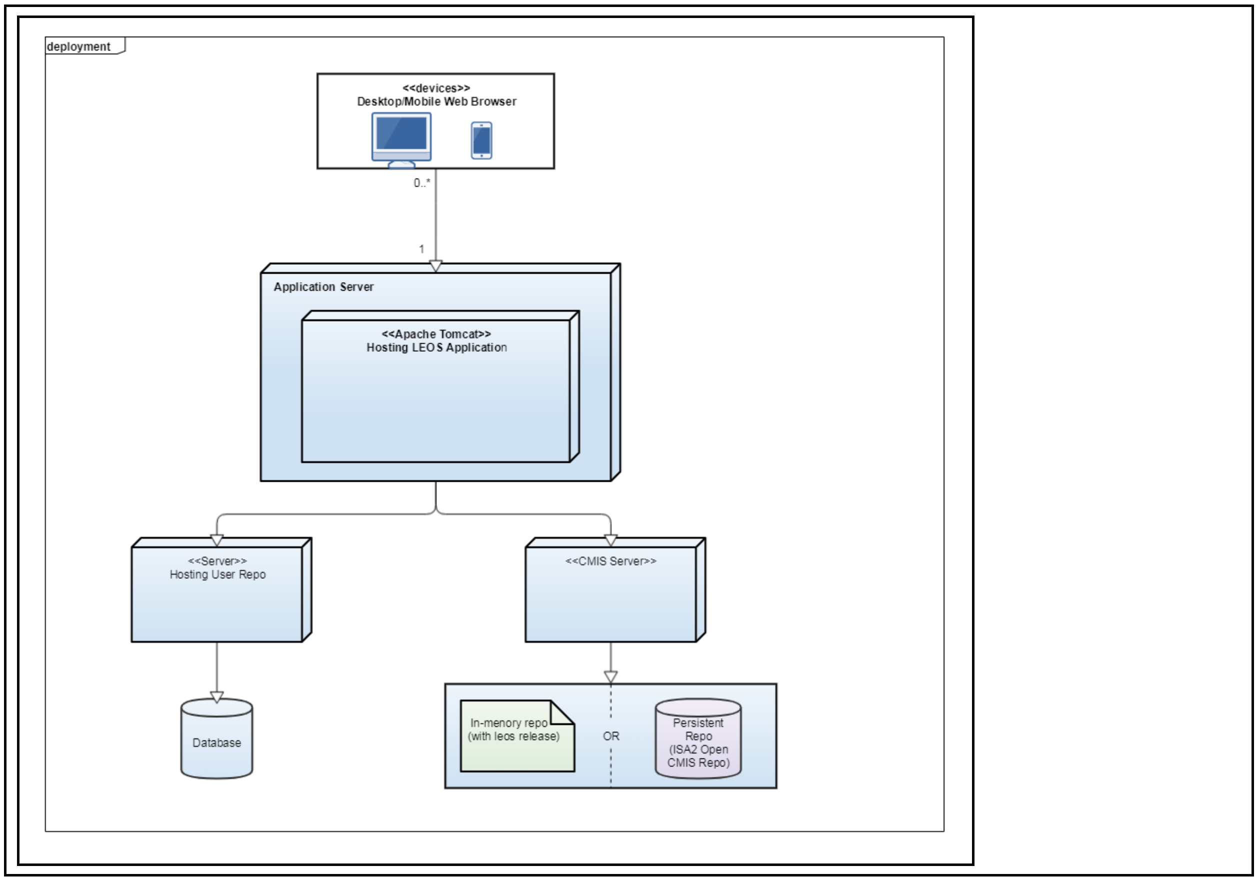
3. Deployment

Each module except the web module will be built as a jar. The packaging will be made as a single war and all the jars corresponding to the Leos modules will be copied as dependencies.

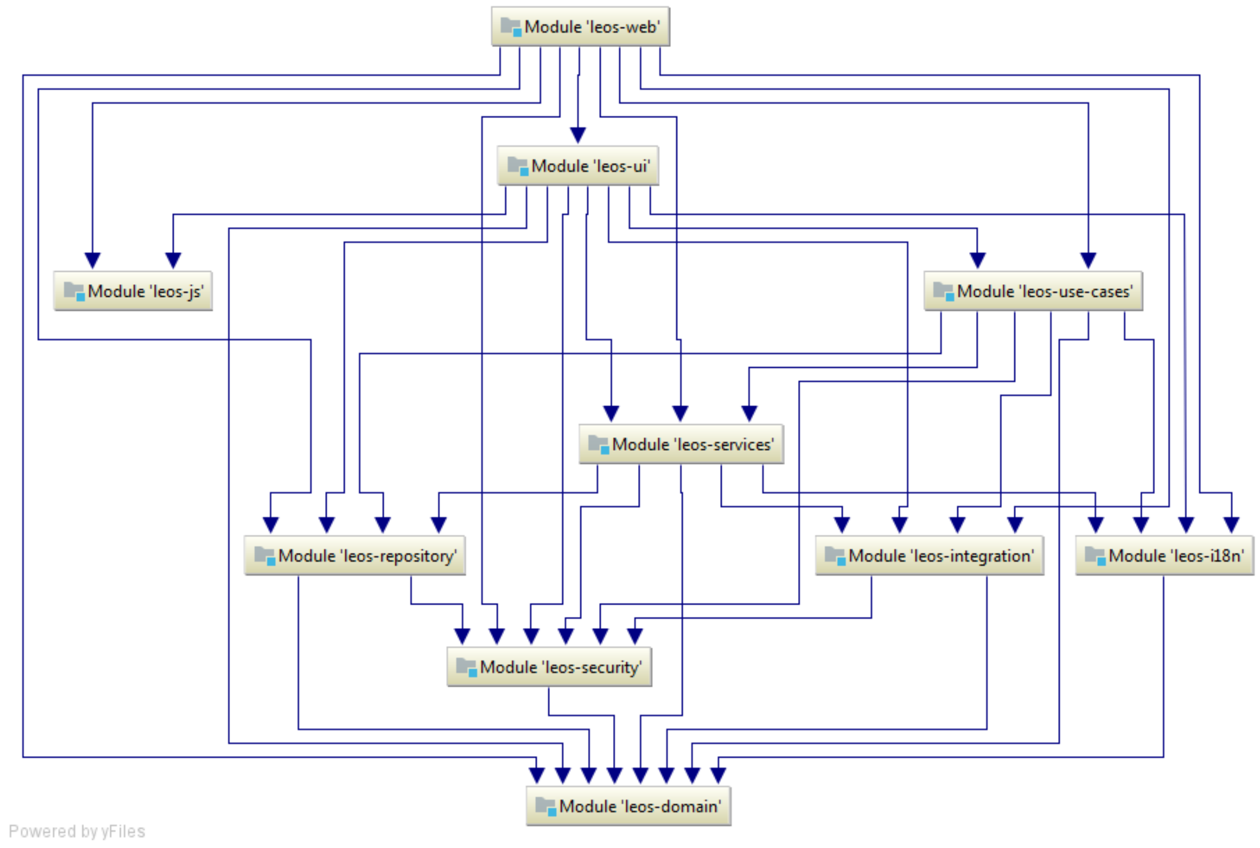
The resulting WAR archive will be deployed in a JEE 5 compliant application server.

In order to run the application, a database and a CMIS server are required.

The below image shows the deployment nodes required to run Leos application.



4. Leos Modules dependency



5. Leos External APIs

5.1 APIs

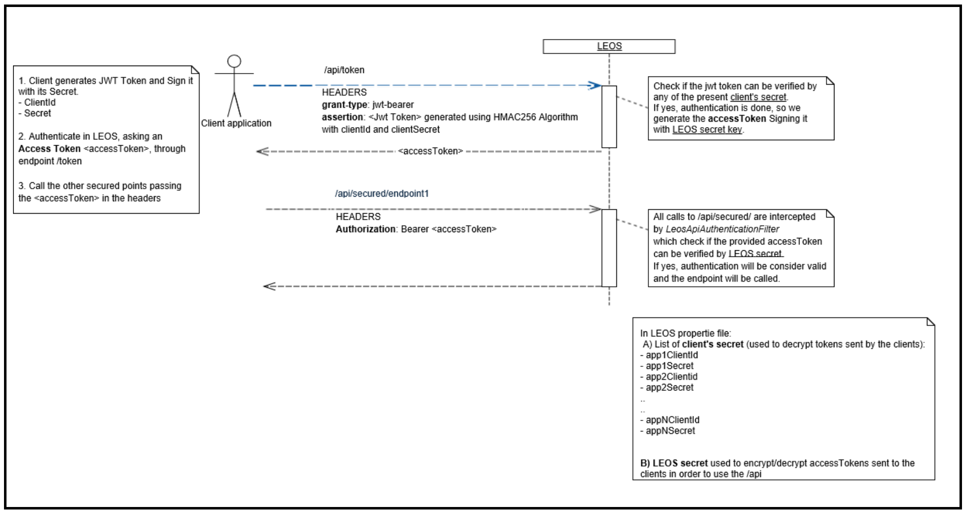
This API has been created for the integration of LEOS with other external systems.

The current status is the following.

Path	**"/token"**
Params	
Signature	@RequestMapping(value = **"/token"**, method = RequestMethod.***GET***, produces = MediaType.***APPLICATION_JSON_VALUE***) @ResponseBody **public** ResponseEntity<Object> getToken(HttpServletRequest request, HttpServletResponse response)
Response	Json object with the Token
Path	**"/secured/compare"**
Params	**Mode -** values accepted: SINGLE_COLUMN_MODE = 1; TWO_COLUMN_MODE = 2; **firstContent** – in xml **secondContent** - in xml
Signature	@RequestMapping(value = **"/secured/compare"**, method = RequestMethod.***POST***, produces = MediaType.***APPLICATION_JSON_UTF8_VALUE***) @ResponseBody **public** ResponseEntity<Object> compareContents(HttpServletRequest request, @RequestParam(**"mode"**) **int** mode, @RequestParam(**"firstContent"**) MultipartFile firstContent, @RequestParam(**"secondContent"**) MultipartFile secondContent)
Response	Json object with the result of the comparison in html.
Path	**"/secured/search/{userId}"**
Params	**userId**
Signature	@RequestMapping(value = **"/secured/search/{userId}"**, method = RequestMethod.***GET***, produces = MediaType.***APPLICATION_JSON_VALUE***) @ResponseBody **public** ResponseEntity<Object> getProposalsForUser(@PathVariable(**"userId"**) String userId)
Response	Json object with the proposals and leg files that the userid can export

Path	**"/secured/searchlegfile/{legFileId}"**
Params	**legFileId**
Signature	@RequestMapping(value = **"/secured/searchlegfile/{legFileId}"**, method = RequestMethod.***GET***, produces = MediaType.***APPLICATION_OCTET_STREAM_VALUE***) @ResponseBody **public** ResponseEntity<Object> getLegFile(@PathVariable(**"legFileId"**) String legFileId)
Response	Leg file requested
Path	**"/secured/renditionfromleg"**
Params	**legFile -** values accepted: PDF = "pdf"; LEGISWRITE = "lw";
Signature	@RequestMapping(value = **"/secured/renditionfromleg"**, method = RequestMethod.***POST***, produces = MediaType.***APPLICATION_OCTET_STREAM_VALUE***) @ResponseBody **public** ResponseEntity<Object> getPdfFromLegFile(@RequestParam(**"legFile"**) MultipartFile legFile, @RequestParam(**"type"**) String type) {}
Response	The PDF or the LegisWrite file

5.2 External APIs Authentication



1. Client generates JWT Token and Sign it with its Secret.
- ClientId
- Secret

2. Authenticate in LEOS, asking an **Access Token** <accessToken>, through endpoint /token

3. Call the other secured points passing the <accessToken> in the headers

HEADERS
grant-type: jwt-bearer
assertion: <Jwt Token> generated using HMAC256 Algorithm with clientId and clientSecret

Check if the jwt token can be verified by any of the present client's secret.
If yes, authentication is done, so we generate the **accessToken** Signing it with LEOS secret key.

HEADERS
Authorization: Bearer <accessToken>

All calls to /api/secured/ are intercepted by *LeosApiAuthenticationFilter* which check if the provided accessToken can be verified by LEOS secret.
If yes, authentication will be consider valid and the endpoint will be called.

In LEOS propertie file:
A) List of **client's secret** (used to decrypt tokens sent by the clients):
- app1ClientId
- app1Secret
- app2Clientid
- app2Secret
..
..
- appNClientId
- appNSecret

B) LEOS secret used to encrypt/decrypt accessTokens sent to the clients in order to use the /api