INCEpTION Developer Guide

The INCEpTION Team

Version 0.12.1

Table of Contents

This document targets developers working on INCEpTION.

Introduction

This document describes how INCEpTION internally works and how it can be extended to fit your use case or project. It is targeted to software developers. At first, we will give a brief overview of the used technology in INCEpTION, then describe how to setup the working environment including version control, IDE and software requirements. Then, the architecture itself with core services and extension points is presented.

Core technology

INCEpTION is written as a Java application and heavily relies on Spring Boot. Its user interface is a web application that is powered by Apache Wicket. The Natural Language Processing components are mostly based on DKPro Core. This includes the tokenization, import and export to many different standard formats, as well as recommenders i.e. machine learning tools that provide annotation support. The internal data format heavily relies on UIMA and its CAS format.

Setup

This section covers setting up a development environment.

Source code management

We use git as our source code management system and collaborate via the INCEpTION repository on Github.

Development workflow

Every feature or bugfix needs to be tracked in an issue on Github. Development is done in branches. Based on the milestone (see the issue description on Github), the new branch is either created from master (if the code should be in the next major release) or from a bugfix release branch (if the code should be in the next minor release). In order to get the code in production, you need to create a pull request on Github of your branch into the target branch (as described before).

In order to contribute to INCEpTION, you need to create a pull request. This section briefly guides you through the best way of doing this:

- Every feature or bugfix needs to be tracked in an issue on Github. If there is no issue for the feature yet, create an issue first.
- Create a branch based on the branch to which you wish to contribute. Normally, you should create this branch from the master branch of the respective project. In the case that you want to fix a bug in the latest released version, you should consider to branch off the latest maintenance branch (e.g. 0.10.x). If you are not sure, ask via the issue you have just created. Do **not** make changes directly to the master or maintenance branches. The name of the branch should be e.g. `feature/[ISSUE-NUMBER]-[SHORT-ISSUE-DESCRIPTION]` or `bugfix/[ISSUE-NUMBER]-[SHORT-ISSUE-DESCRIPTION]`.
- Now you make changes to your branch. When committing to your branch, use the format shown below for your commit messages. Note that # normally introduces comments in git. You may have to reconfigure git before attempting an interactive rebase and switch it to another comment character.

```
#[ISSUE NUMBER] - [ISSUE TITLE]
[EMPTY LINE]
- [CHANGE 1]
- [CHANGE 2]
- [...]
```

You can create the pull request any time after your first commit. I.e. you do not have to wait until you are completely finished with your implementation. Creating a pull request early tells other developers that you are actively working on an issue and facilitates asking questions about and discussing implementation details.

Git configuration

Before committing, make sure that you specified your email and name in the git config so that commits can be attributed to you. This can e.g. be done as described in the Git Documentation.

All sources files are stored using UNIX line endings. If you develop on Windows, you have to set the `core.autocrlf` configuration setting to `input` to avoid accidentally submitting Windows line endings to the repository. Using `input` is a good strategy in most cases, thus you should consider setting this as a global (add `--global`) or even as a system (`--system`) setting.

Configure git line ending treatment

```
C:\> git config --global core.autocrlf input
```

After changing this setting, best do a fresh clone and check-out of the project.

Checkstyle and Formatting

We use a style for formatting the source code in INCEpTION. Our approach consists of two steps:

- DKPro code formatting profile - the profile configures your IDE to auto-format the code according to our guidelines as you go.
- Checkstyle - this tool is used to check if the source code is actually formatted according to our guidelines. It is run as part of a Maven build and the build fails if the code is not formatted properly.

Here is a brief summary of the formatting rules:

- no tabs, only spaces
- indenting using 4 spaces in Java files and 2 spaces in XML files
- maximum 100 characters per line (with a few exceptions)
- curly braces on the next line for class/method declarations, same line for logic blocks (if/for/…)
- parameter names start with `a` (e.g. `void foo(String aValue)`)

Setting up for the development in Eclipse

This is a guide to setting up a development environment using Eclipse on Mac OS X. The procedure should be similar for other operation systems.

First, you need to follow some steps of the user [InstallationGuide installation guide]. It is recommended to configure a MySQL-server.

We recommend you start from a **Eclipse IDE for Java Developers** package.

Use a JDK

On Linux or OS X, the following setting is not necessary. Having a full JDK installed on your system is generally sufficient. You can skip on to the next section.

On Windows, you need to edit the `eclipse.ini` file and directly before the `-vmargs` line, you have to add the following two lines. Mind to replace `C:/Program Files/Java/jdk1.8.0_144` with the actual location of the JDK on your system. Without this, Eclipse will complain that the `jdk.tools:jdk.tools` artifact would be missing.

Force Eclipse to run on a JDK

```
-vm
C:/Program Files/Java/jdk1.8.0_144/jre/bin/server/jvm.dll
```

Eclipse Plug-ins

- **Maven Integration:** m2e , already comes pre-installed with the Eclipse IDE for Java Developers. If you use another edition of Eclipse which does not have m2e pre-installed, go to **Help**→**Install New Software**, select "--All available sites--" and choose **Collaboration** → **m2e - Maven Integration for Eclipse**
- **Apache UIMA tools:** Update site: `http://www.apache.org/dist/uima/eclipse-update-site/`
- **Eclipse Web Development Tooling:** go to **Help**→**Install New Software**, select "--All available sites--" and select the following plug-ins for installation from the section **Web, XML, Java EE and OSGi Enterprise Development**:
    - Eclipse Java Web Developer Tools
    - Eclipse Web Developer Tools
    - Eclipse XML Editors and Tools
    - JST Server Adapters
    - JST Server Adapters Extensions
    - JST Server UI
    - m2e-wtp - Maven Integration for WTP
    - WST Server Adapters

Eclipse Workspace Settings

- You should check that Text file encoding is UTF-8 in **Preferences** → **General** → **Workspace** of your Eclipse install.

Importing INCEpTION into the Workspace

Checkout out the INCEpTION git repository with your favorite git client. If you use the command-line client, use the command

```
$ git clone https://github.com/inception-project/inception.git
```

In Eclipse, go to **File** → **Import**, choose **Existing Maven projects**, and select the folder to which you have cloned INCEpTION. Eclipse should automatically detect all modules.

Eclipse Tomcat Integration

Download Apache Tomcat from `http://tomcat.apache.org/` (we're using version 8.5). Then, you need to add the Tomcat server to your runtime configuration. Go to preferences and go to **Servers** → **Runtime environments**:

When prompted for an installation path, specify the folder where you extracted (or installed) Apache Tomcat v8.5 into.

Change the runtime configuration for the project. On the left side of the dialog, you should now be able to select Apache Tomcat. Change its VM arguments and include the definition `-Dinception.home="/srv/inception"` to specify the home directory for the application. Also add `-Dwicket.core.settings.general.configuration-type=development` to enable the development mode. This adds additional debugging features to the UI and disables UI caches.

Head to the servers pane. If you cannot locate it in your eclipse window, add it by going to **Window** → **Show View** → **Other…** and select **Servers**. Right click on **Tomcat v8.5 localhost** and click on **Add and remove…**:

INCEpTION should now be configured to start with Tomcat.

In the **Servers** view, double-click on the Tomcat instance you have configured. Activate the checkbox **Serve modules without publishing**. Go to the **Modules** tab, select the INCEpTION module and disable auto-reloading. After these changes, you will have to manually restart the Tomcat server in order for changes to Java class files to take effect. However, as a benefit, changes to HTML, CSS or JavaScript files take effect immediately and you just have to refresh the browser to see the changes.

Setting up Checkstyle and Formatting

We use a style for formatting the source code in INCEpTION (see Checkstyle and Formatting. The following section describes how to use it with Eclipse.

First, obtain the DKPro code formatting profile from the DKPro website (Section "Code style"). In Eclipse, go to **Preferences** → **Java** → **Code Style** → **Formatter** to import the file. Apparently, the files can also be used with IntelliJ via the [Eclipse Code Formatter](https://plugins.jetbrains.com/plugin/6546-eclipse-code-formatter) plugin.

> ℹ The parameter prefix `a` needs to be configured manually. In Eclipse go to **Preferences** → **Java** → **Code Style** set the **prefix list** column in the **parameters** row to `a`.

Second, install the Checkstyle plugin for Eclipse as well as the Maven Checkstyle plugin for Eclipse. These plugins make Eclipse automatically pick up the checkstyle configuration from the Maven project and highlight formatting problems directly in the source code editor.

- Install **Checkstyle Eclipse plugin** from here: `http://eclipse-cs.sourceforge.net`
- Install the **Checkstyle configuration plugin for M2Eclipse** from here: `http://m2e-code-quality.github.com/m2e-code-quality/site/latest/`
- Select all INCEpTION projects, right click and do a **Maven** → **Update project**

> ℹ Should the steps mentioned above not have been sufficient, close all the INCEpTION projects in Eclipse, then remove them form the workspace (not from the disk), delete any `.checkstyle` files in the INCEpTION modules, and then re-import them into Eclipse again using **Import**→**Existing Maven projects**. During the project import, the Checkstyle configuration plugin for M2Eclipse should properly set up the `.checkstyle` files and activate checkstyle.
> If the Maven project update cannot be completed due to missing .jars, execute a Maven install via right click on the inception project **Run as** → **Maven build…**, enter the goal `install` and check **Skip Tests**. Alternatively, use the command `mvn clean install -DskipTests`.

Setting up for the development in IntelliJ IDEA

This is a guide to setting up a development environment using IntelliJ IDEA. We assume that the Community Version is used, but this guide should also apply to the Enterprise Version.

After checking out INCEpTION from Github, open Intellij and import the project. The easiest way is to go to `File → Open` and the select the `pom.xml` in the INCEpTION root directory. IntelliJ IDEA will then guide you through the import process, the defaults work out of the box. INCEpTION can now be started via running `inception-app-webapp/src/main/java/de/tudarmstadt/ukp/inception/INCEpTION.java`.

Checkstyle and Formatting

We use a style for formatting the source code in INCEpTION (see Checkstyle and Formatting. The following section describes how to use it with IntelliJ IDEA.

First, install the Checkstyle-IDEA plugin. In `File | Settings | Other Settings | Checkstyle`, navigate to the **Checkstyle** tab. Start to add a new configuration file by clicking on the `+` on the right, navigate to `inception-build/src/main/resources/inception/checkstyle.xml` and apply the changes. Make sure to check the box next to the newly created configuration and apply it as well.

In order to achieve the same formatting and import order as Eclipse, install the Eclipse Code Formatter. Download the DKPro Eclipse Code Style file. In `File | Settings | Other Settings | Eclipse Code Formatter`, create a new profile using this file.

Also make sure to enable auto import optimization in `File | Settings | Editor | General | Auto Import`.

IntelliJ IDEA Tomcat Integration

This requires IntelliJ IDEA Ultimate. Using Tomcat allows editing HTML,CSS and JavaScript on the fly without restarting the application. First, download Apache Tomcat from http://tomcat.apache.org/ (we're using version 8.5). Then, you need to create a Tomcat server runtime configuration in `Run | Edit Configurations…`. Click on the `+` icon, select `Tomcat Server → Local`. Click on the `Deployment` tab and then on the `+` icon to select an artifact to deploy. Choose the exploded war version. Select the `Server` tab, navigate to the path of your Tomcat server, and update the `on Update` action to `Update classes and resources` for both. Make sure that all port settings are different. You now can start or debug your web application via Tomcat. If starting throws a permission error, make sure that the mentioned file, e.g. `catalina.sh` is marked as executable.

Experimental: If desired, you can also use hot-code replacement via HotswapAgent. This allows you to change code, e.g. adding methods without needing to restart the Tomcat server. For this, follow the excellent HotSwap IntelliJ IDEA plugin guide.

Building documentation

The documentation can be built using a support class in `inception-doc/src/test/java/de/tudarmstadt/ukp/inception/doc/GenerateDocumentation.java`. To make it usable from Intellij IDEA, you need to build the whole project at least once. Run the class. If it fails, alter the run configuration and add a new environment variable `INTELLIJ=true` and check that the working directory is the INCEpTION root directory. The resulting documentation will be in `target/doc-out`.

Developing INCEpTION and webanno together

INCEpTION builds on Webanno. Therefore, it can be desirable to work on Webanno and INCEpTION at the same time so that changes in Webanno are directly visible in your INCEpTION development. For this, check out Webanno from Github. Then, import it as a Maven project: first, open the Maven sidebar on the right. Then, click on the `+` and select the Webanno `pom.xml` . In `File | Settings | Build, Execution, Deployment | Build Tools | Maven`, make sure to check `Always update snapshots`.

Architecture

INCEpTION uses a standard 3-layer architecture with the presentation layer using Wicket at the top, the business layer heavily relying on Spring Boot and the data layer which is interfaced with Hibernate at the bottom.

Wicket pages

Wicket can only inject components that are interfaces. A pattern for these cases is to create an `ExampleComponent` interface and implement it in an `ExampleComponentImpl` class.

Services

Services encode the core logic of INCEpTION. They can be injected into Wicket pages and other services to interact with the rest of the application. Services can inject Spring components via autowiring. A good example of a service can e.g. be seen in the `SchedulingService.java`.

Database

The database can be accessed via Hibernate. The schema itself and migrations are managed by Liquibase.

Migration

When changing the database schema, migrations from the current schema to the new one need to be defined. They describe how the schema needs to be modified. This way, INCEpTION can be upgraded to newer versions without needing to manually alter the database schema. The migration process determines the current version of the schema and only applies transformations from there to the newest one. Each module defines its own data base tables and migrations in a file called `db-changelog.xml`. These are automatically discovered by Liquibase and used when starting INCEpTION.

Modules

Documents

Source documents

The original document uploaded by a user into a project. The document is preserved in its original format.

Annotation documents

Annotations made by a particular user on a document. The annotation document is persisted separately from the original document. There is one annotation document per user per document. Within the tool, a CAS data structure is used to represent the annotation document.

Annotation Schema

Layers

The layers mechanism allows supporting different types of annotation layers, e.g. span layers, relation layers or chain layers. It consists of the following classes and interfaces:

- The `LayerSupport` interface provides the API for implementing layer types.
- The `LayerSupportRegistry` interface and its default implementation `LayerSupportRegistryImpl` serve as an access point to the different supported layer types.
- The `LayerType` class which represents a short summary of a supported layer type. It is used when selecting the type of a feature in the UI.
- The `TypeAdapter` interface provides methods to create, manipulate or delete annotations on the given type of layer.

To add support for a new type of layer, create a Spring component class which implements the `LayerSupport` interface. Note that a single layer support class can handle multiple layer types. However, it is generally recommended to implement a separate layer support for every layer type. Implement the following methods:

- `getId()` to return a unique identifier for the new layer type. Typically the Spring bean name is returned here.
- `getSupportedLayerTypes()` to return a list of all the supported layer types handled by the new layer support. This values returned here are used to populate the layer type choice when creating a new layer in the project settings.
- `accepts(AnnotationLayer)` to return `true` for any annotation layer that is handled by the new layer support. I.e. `AnnotationLayer.getType()` must return a layer type identifier that was produced by the given layer support.
- `generateTypes(TypeSystemDescription, AnnotationLayer)` to generate the UIMA type system for the given annotation layer. This is a partial type system which is merged by the application with the type systems produced by other layer supports as well as with the base type system of the application itself (i.e. the DKPro Core type system and the internal types).
- `getRenderer(AnnotationLayer)` to return an early-stage renderer for the annotations on the given layer.

> ℹ The concept of layers is not yet fully modularized. Many parts of the application will only know how to deal with specific types of layers. Adding a new layer type should not crash the application, but it may also not necessarily be possible to actually use the new layer. In particular, changes to the TSV format may be required to support new layer types.

Span layer

A span layer allows to create annotations over spans of text.

If `attachType` is set, then an annotation can only be created over the same span on which an annotation of the specified type also exists. For span layers, setting `attachFeature` is mandatory if a `attachType` is defined. The `attachFeature` indicates the feature on the annotation of the `attachType` layer which is to be set to the newly created annotation.

For example, the `Lemma` layer has the `attachType` set to `Token` and the `attachFeature` set to `lemma`. This means, that a new lemma annotation can only be created where a token already exists and that the `lemma` feature of the token will point to the newly created lemma annotation.

Deleting an annotation that has other annotations attached to it will also cause the attached annotations to be deleted.

> This case is currently not implemented because it is currently not allowed to create spans that attach to other spans. The only span type for which this is relevant is the `Token` type which cannot be deleted.

Relation layer

A relation layer allows to draw arcs between span annotations. The `attachType` is mandatory for relation types and specifies which type of annotations arcs can be drawn between.

Arcs can only be drawn between annotations of the same layer. It is not possible to draw an arc between two spans of different layers.

Only a single relation layer can attach to any given span layer.

If the `annotation_feature` is set, then the arc is not drawn between annotations of the layer indicated by `annotation_type`, but between annotations of the type specified by the feature. E.g. for a dependency relation layer, `annotation_type` would be set to `Token` and `annotation_feature` to `pos`. The `Token` type has no visual representation in the UI. However, the `pos` feature points to a `POS` annotation, which is rendered and between which the dependency relation arcs are then drawn.

Deleting an annotation that is the endpoint of a relation will also delete the relation. In the case that `annotation_feature`, this is also the case if the annotation pointed to is deleted. E.g. if a POS annotation in the above example is deleted, then the attaching relation annotations are also deleted.

Document Metadata

A document metadata layer can be used to create annotations that apply to an entire document instead of to a specific span of text.

Document metadata types inherit from the UIMA `AnnotationBase` type (text annotations inherit from `Annotation`). As such, they do not have begin/end offsets.

Layers Behaviors

Layer behaviors allow to customize the way a layer of a particular span behaves, e.g. whether a span is allowed to cross sentence boundaries, whether it anchors to characters or tokens, whether the tree of relations among annotations is valid, etc. The layer behaviors tie in with the specific `LayerSupport` implementations. The mechanism itself consists of the following classes and interfaces:

- The `LayerBehavior` interface provides the API necessary for registering new behaviors. There are abstract classes such as `SpanLayerBehavior` or `RelationLayerBehavior` which provide the APIs for behaviors of specific layer types.
- The `LayerBehaviorRegistry` and its default implementation `LayerBehaviorRegistryImpl` serve as an access point to the different supported layer behaviors. Any Spring component implementing the `LayerBehavior` interface is loaded, and will be named in the logs when the web app is launched. The classpath scanning used to locate Spring beans is limited to specific Java packages, e.g. any packages starting with `de.tudarmstadt.ukp.clarin.webanno`.

A layer behavior may have any of the following responsibilities:

- Ensure that new annotations that are created conform with the behavior. This is done via the `onCreate` method. If the annotation to be created does not conform with the behavior, the method can cancel the creation of the annotation by throwing an `AnnotationException`.
- Highlight annotations not conforming with the behavior. This is relevant when importing pre-annotated files or when changing the behavior configuration of an existing layer. The relevant method is `onRender`. If an annotation does not conform with the behavior, a error marker should be added for problematic annotation. This is done by creating a `VComment` which attaches an error message to a specified visual element, then adding that to the repsonse `VDocument`. Note that `onRender` is unlike `onCreate` and `onValidate` in that it only has indirect access to the CAS: it is passed a mapping from `AnnotationFS` instances to their corresponding visual elements, and can use `.getCAS()` on the FS. The annotation layer can be identified from the visual element with `.getLayer().getName()`.
- Ensure that documents being marked as **finished** conform with the behavior. This is done via the `onValidate` method, which returns a list of `LogMessage, AnnotationFS` pairs to report errors associated with each FS.

Features

The features mechanism allows supporting different types of annotation features, e.g. string features, numeric features, boolean features, link features, etc. It consists of the following classes and interfaces:

- The `FeatureSupport` interface provides the API for implementing feature types.
- The `FeatureSupportRegistry` interface and its default implementation `FeatureSupportRegistryImpl` serve as an access point to the different supported feature types.
- The `FeatureType` class which represents a short summary of a supported feature type. It is used when selecting the type of a feature in the UI.
- The `TypeAdapter` interface provides methods to create, manipulate or delete annotations on the given type of layer.

To add support for a new type of feature, create a Spring component class which implements the

`FeatureSupport` interface. Note that a single featre support class can handle multiple feature types. However, it is generally recommended to implement a separate layer support for every feature type. Implement the following methods:

- `getId()` to return a unique identifier for the new feature type. Typically the Spring bean name is returned here.
- `getSupportedFeatureTypes()` to return a list of all the supported feature types handled by the new feature support. This values returned here are used to populate the feature type choice when creating a new feature in the project settings.
- `accepts(AnnotationLayer)` to return `true` for any annotation layer that is handled by the new layer support. I.e. `AnnotationLayer.getType()` must return a layer type identifier that was produced by the given layer support.
- `generateFeature(TypeSystemDescription, TypeDescription, AnnotationFeature)` add the UIMA feature definition for the given annotation feature to the given type.

If the new feature has special configuration settings, then implement the following methods:

- `readTraits(AnnotationFeature)` to extract the special settings form the given annotation feature definition. It is expected that the traits are stored as a JSON string in the `traits` field of `AnnotationFeature`. If the `traits` field is `null`, a new traits object must be returned.
- `writeTraits(AnnotationFeature, T)` to encode the layer-specific traits object into a JSON string and store it in the `traits` field of `AnnotationFeature`.
- `createTraitsEditor(String, IModel<AnnotationFeature>` to create a custom UI for the special feature settings. This UI is shown below the standard settings in the feature detail editor on the **Layers** tab of the project settings.

Search

The search module contains the basic methods that implement the search service and search functionalities of INCEpTION.

The `SearchService` and `SearchServiceImpl` classes define and implement the search service as a Spring component, allowing other modules of INCEpTION to create an index for a given project, and to perform queries over that index.

The indexes have two different aspects: the conceptual index, represented by the Index class, and the physical index, represented by a particular physical implementation of an index. This allows different search providers to be used by INCEpTION. Currently, the default search implementation uses Mtas (https://github.com/meertensinstituut/mtas), a Lucene / Solr based index engine that allows to annotate not only raw texts but also different linguistic annotations.

Every search provider is defined by its own index factory, with a general index registry to hold all the available search providers.

Mtas Index

The Mtas index is implemented in the **MtasDocumentIndex** and **MtasDocumentIndexFactory** classes. Furthermore, the **MtasUimaParser** class provides a parser to be used by Lucene when adding a new document to the index.

- MtasDocumentIndexFactory

The factory allows to build a new MtasDocumentIndex through the getNewIndex method, which is called by the search service.

- MtasDocumentIndex

This class holds the main functionalities of a Mtas index. Its methods are called by the search service and allow to create, open close and drop a Mtas index. It allows to add or delete a document from an index, as well as to perform queries on the index.

Each index is related to only one project, and every project can have only one index from a given search provider.

When adding a document to a Mtas index, the Lucene engine will use the class MtasUimaParser in order to find out which are the tokens and annotations to be indexed.

- MtasUimaParser

The parser is responsible for creating a new TokenCollection to be used by Lucene, whenever a new document is being indexed. The token collection consists of all the tokens and annotations found in the document, which are transformed into Mtas tokens in order to be added to the Lucene index. The parser scans the document CAS and goes through all its annotations, finding out which ones are related to the annotation layers in the document's project - those are the annotations to be indexed. Currently, the parser only indexes span type annotations.

Recommendation

For information on the different recommenders, please refer to user guide.

Implementing a custom recommender

This section describes the overall design of internal recommenders in INCEpTION and gives a tutorial on how to implement them. Internal recommenders are created by implementing relevant Java interfaces and are added via Maven dependencies. These are then picked up during application startup by the Spring Framework.

For this tutorial, we will add a recommender for named entities that uses the data majority label for predicting, i.e. it predicts always the label that appears most often in the training data. The full code for this example can be found in the **inception-example-imls-data-majority** module.

Setting up the environment

To get started, check out the most recent source code of INCEpTION from Github and import it as a Maven project in the IDE of your choice. Add a new module to the INCEpTION project itself, we will call it **inception-example-imls-data-majority**.

In the root `pom.xml` of the INCEpTION project, add your recommender as a dependency. Update the version of the dependency entry you just created to the version you find in the *pom.xml* of the INCEpTION project. It should look like this:

```
<dependencies>
…
    <dependency>
        <groupId>de.tudarmstadt.ukp.inception.app</groupId>
        <artifactId>inception-imls-data-majority</artifactId>
        <version>0.12.1</version>
    </dependency>
…
</dependencies>
```

Add the same entry in `inception-app-webapp`, but omit the version number. It then uses automatically the version in the parent POM file. Also add it to `usedDependencies` there.

To add a new recommender to INCEpTION, two classes need to be created. These are described in the following.

Implementing the RecommendationEngine

Recommenders give suggestions for possible annotations to the user. In order to do that, they need to be able be to trained on existing annotations, predict annotations in a document and be evaluated for a performance estimate. This is what the `RecommendationEngine` abstract class is for. It defines the methods that are used to train, test and evaluate a machine learning algorithm and offers several helper methods. Instances of this class often wrap external machine learning packages like OpenNLP or Deeplearning4j.

Recommenders in INCEpTION heavily rely on Apache UIMA types and features. A recommender is configured for a certain layer and a certain feature. A layer can be seen as the type of annotation you want to to, e.g. `POS`, `NER`. Layers correspond to UIMA types. A feature is one piece of information that should be annotated, e.g. the POS tag. One layer can have many features. When extending `RecommendationEngine`, the predicted layer/type can be obtained by `getPredictedType`, the feature to predict respectively by `getPredictedFeature`.

Annotations are given to a recommender in the form of a UIMA CAS. One CAS corresponds to one document in INCEpTION. Annotations from a CAS can be read and manipulated via the CasUtil.

We start by creating a new class `de.tudarmstadt.ukp.inception.recommendation.imls.datamajority.DataMajorityNerRecommender` that implements `RecommendationEngine`. Please see the JavaDoc of the respective methods for their semantics.

Class and member definition for the DataMajorityNerRecommender

```
public class DataMajorityNerRecommender
        extends RecommendationEngine
{
    public static final Key<DataMajorityModel> KEY_MODEL = new Key<>("model");

    private final Logger log = LoggerFactory.getLogger(getClass());

    public DataMajorityNerRecommender(Recommender aRecommender)
    {
        super(aRecommender);
    }

    /**
     * Given training data in {@code aCasses}, train a model. In order to save data
between
     * runs, the {@code aContext} can be used.
     * This method must not mutate {@code aCasses} in any way.
     * @param aContext The context of the recommender
     * @param aCasses The training data
     */
    public abstract void train(RecommenderContext aContext, List<CAS> aCasses)
            throws RecommendationException;

    /**
     * Given text in {@code aCas}, predict target annotations. These should be written
into
     * {@code aCas}. In order to restore data from e.g. previous training, the {@code
aContext}
     * can be used.
     * @param aContext The context of the recommender
     * @param aCas The training data
     */
    public abstract void predict(RecommenderContext aContext, CAS aCas)
            throws RecommendationException;

    /**
     * Evaluates the performance of a recommender by splitting the data given in
{@code aCasses} in
     * training and test sets by using {@code aDataSplitter}, training on the training
set and
     * measuring performance on unseen data on the training set. This method must not
mutate
     * {@code aCasses} in any way.
     *
     * @param aCasses
     *            The CASses containing target annotations
     * @param aDataSplitter
     *            The splitter which determines which annotations belong to which set
     * @return Scores available through an EvaluationResult object measuring the
performance
     *         of predicting on the test set
     */
    public abstract EvaluationResult evaluate(List<CAS> aCasses, DataSplitter
aDataSplitter)
        throws RecommendationException;

    private static class DataMajorityModel
    {
        private final String majorityLabel;
        private final double confidence;
        private final int numberOfAnnotations;

        private DataMajorityModel(String aMajorityLabel, double aConfidence,
                int aNumberOfAnnotations)
        {
            majorityLabel = aMajorityLabel;
            confidence = aConfidence;
            numberOfAnnotations = aNumberOfAnnotations;
        }
    }

    private static class Annotation
    {
        private final String label;
        private final double score;
        private final String explanation;
        private final int begin;
        private final int end;

        private Annotation(String aLabel, int aBegin, int aEnd)
        {
            this(aLabel, 0, 0, aBegin, aEnd);
        }

        private Annotation(String aLabel, double aScore, int aNumberOfAnnotations, int
aBegin,
                int aEnd)
        {
            label = aLabel;
            score = aScore;
            explanation = "Based on " + aNumberOfAnnotations + " annotations";
            begin = aBegin;
            end = aEnd;
        }
    }
}
```

For the constructor, we take the `Recommender` object which contains the recommender configuration, e.g. the layer and the name of the feature to recommend. The next step is to implement the required methods.

`DataMajorityModel` and `Annotation` are internal data classes to simplify the code.

RecommenderContext

Instances of RecommendationEngine itself are stateless. If data like trained models need to be saved and loaded, it can be saved in the `RecommenderContext` that is given in the interface methods. When needed again, e.g. for prediction, it then can be loaded again. The `Key` class is used in order to ensure type safety.

Training

Training consists of extracting annotations followed by training and saving the model. The platform needs to know whether the recommender is ready for prediction, this is done by overriding `RecommendationEngine::isReadyForPrediction`.

Training routine

```
@Override
public RecommendationEngineCapability getTrainingCapability()
{
    return TRAINING_REQUIRED;
}

@Override
public void train(RecommenderContext aContext, List<CAS> aCasses)
        throws RecommendationException
{
    List<Annotation> annotations = extractAnnotations(aCasses);

    DataMajorityModel model = trainModel(annotations);
    aContext.put(KEY_MODEL, model);
}

@Override
public boolean isReadyForPrediction(RecommenderContext aContext)
{
    return aContext.get(KEY_MODEL).map(Objects::nonNull).orElse(false);
}
```

Extracting annotations itself is done by iterating over all documents and selecting all annotations for each. Here, we need to use the layer name and feature for which the recommender is configured to extract the correct annotations.

Extracting annotations from the documents

```
private List<Annotation> extractAnnotations(List<CAS> aCasses)
{
    List<Annotation> annotations = new ArrayList<>();

    for (CAS cas : aCasses) {
        Type annotationType = CasUtil.getType(cas, layerName);
        Feature predictedFeature = annotationType.getFeatureByBaseName(featureName);

        for (AnnotationFS ann : CasUtil.select(cas, annotationType)) {
            String label = ann.getFeatureValueAsString(predictedFeature);
            if (isNotEmpty(label)) {
                annotations.add(new Annotation(label, ann.getBegin(), ann.getEnd()));
            }
        }
    }

    return annotations;
}
```

The training itself is done by counting the number of occurrences for each label that was seen in the documents. The label is then the one which occurred the most in the training documents.

Training the model

```
private DataMajorityModel trainModel(List<Annotation> aAnnotations)
        throws RecommendationException
{
    Map<String, Integer> model = new HashMap<>();
    for (Annotation ann : aAnnotations) {
        int count = model.getOrDefault(ann.label, 0);
        model.put(ann.label, count + 1);
    }

    Map.Entry<String, Integer> entry = model.entrySet().stream()
            .max(Map.Entry.comparingByValue())
            .orElseThrow(
                    () -> new RecommendationException("Could not obtain data majority
label")
            );

    String majorityLabel = entry.getKey();
    int numberOfAnnotations = model.values().stream().reduce(Integer::sum).get();
    double confidence = (float) entry.getValue() / numberOfAnnotations;

    return new DataMajorityModel(majorityLabel, confidence, numberOfAnnotations);
}
```

We also compute a dummy score here which is displayed in the UI and used for e.g. active learning.

Predicting

The first thing we do when predicting is to load the model we saved during training. For every candidate in the document, we assign the majority label, create a new annotation and add it to the `CAS`. From there, it will be read by INCEpTION and displayed to the user.

Predicting annotations for a CAS

```
@Override
public void predict(RecommenderContext aContext, CAS aCas) throws
RecommendationException
{
    DataMajorityModel model = aContext.get(KEY_MODEL).orElseThrow(() ->
            new RecommendationException("Key [" + KEY_MODEL + "] not found in context
"));

    // Make the predictions
    Type tokenType = CasUtil.getAnnotationType(aCas, Token.class);
    Collection<AnnotationFS> candidates = CasUtil.select(aCas, tokenType);
    List<Annotation> predictions = predict(candidates, model);

    // Add predictions to the CAS
    Type predictedType = getPredictedType(aCas);
    Feature scoreFeature = getScoreFeature(aCas);
    Feature scoreExplanationFeature = getScoreExplanationFeature(aCas);
    Feature predictedFeature = getPredictedFeature(aCas);
    Feature isPredictionFeature = getIsPredictionFeature(aCas);

    for (Annotation ann : predictions) {
        AnnotationFS annotation = aCas.createAnnotation(predictedType, ann.begin, ann
.end);
        annotation.setStringValue(predictedFeature, ann.label);
        annotation.setDoubleValue(scoreFeature, ann.score);
        annotation.setStringValue(scoreExplanationFeature, ann.explanation);
        annotation.setBooleanValue(isPredictionFeature, true);
        aCas.addFsToIndexes(annotation);
    }
}
```

For a document, we consider possible candidates for a named entity to be tokens that are upper case. In a real recommender, the step of candidate extraction should be more elaborate than that, but for this tutorial, it is sufficient.

When making predictions, we also set the score feature to put a number on the quality of the annotation. The UIMA score feature to set can be obtained by calling `getScoreFeature` inside a `RecommendationEngine`. When creating predictions, make sure to call `annotation.setBooleanValue(isPredictionFeature, true);` so that INCEpTION knows it is a prediction, not a real annotation. In addition, we provide an explanation for the score through the UIMA feature obtained by calling `getScoreExplanationFeature` inside a `RecommendationEngine`.

Making the predictions

```
private List<Annotation> predict(Collection<AnnotationFS> candidates,
                                 DataMajorityModel aModel)
{
    List<Annotation> result = new ArrayList<>();
    for (AnnotationFS token : candidates) {
        String tokenText = token.getCoveredText();
        if (tokenText.length() > 0 && !Character.isUpperCase(tokenText.codePointAt(0)
)) {
            continue;
        }

        int begin = token.getBegin();
        int end = token.getEnd();

        Annotation annotation = new Annotation(aModel.majorityLabel, aModel.
confidence,
                aModel.numberOfAnnotations, begin, end);
        result.add(annotation);
    }

    return result;
}
```

We use the dummy score here from the training as the confidence.

Evaluating

When configuring a recommender, it can be specified that it needs to achieve a certain score before the recommendations are shown to the user. For that, the platform regularly evaluates recommenders in the background. We use macro-averaged F1-score as an evaluation score. In code, the evaluation is implemented in the `evaluate` method.

Evaluation is done on a set of documents. In order to properly divide the annotations into training and test set, a `DataSplitter` is given which tells you to which data set an annotation belongs.

For the actual evaluation, we collect the true label and the predicted majority label in a `LabelPair` for each true label. A stream of these instances can then be collected with the use of an `EvaluationResultCollector` as an `EvaluationResult` object - the result of the evaluation. This object provides access to calculations for token-based accuracy, macro-averaged precision, recall and F1-score. This F1-score is later used for comparison with the user-defined threshold to activate the recommender.

```
@Override
public EvaluationResult evaluate(List<CAS> aCasses, DataSplitter aDataSplitter)
        throws RecommendationException
{
    List<Annotation> data = extractAnnotations(aCasses);
    List<Annotation> trainingData = new ArrayList<>();
    List<Annotation> testData = new ArrayList<>();

    for (Annotation ann : data) {
        switch (aDataSplitter.getTargetSet(ann)) {
        case TRAIN:
            trainingData.add(ann);
            break;
        case TEST:
            testData.add(ann);
            break;
        case IGNORE:
            break;
        }
    }

    int trainingSetSize = trainingData.size();
    int testSetSize = testData.size();
    double overallTrainingSize = data.size() - testSetSize;
    double trainRatio = (overallTrainingSize > 0) ? trainingSetSize /
overallTrainingSize : 0.0;

    if (trainingData.size() < 1 || testData.size() < 1) {
        log.info("Not enough data to evaluate, skipping!");
        EvaluationResult result = new EvaluationResult(trainingSetSize,
                testSetSize, trainRatio);
        result.setEvaluationSkipped(true);
        return result;
    }

    DataMajorityModel model = trainModel(trainingData);

    // evaluation: collect predicted and gold labels for evaluation
    EvaluationResult result = testData.stream()
            .map(anno -> new LabelPair(anno.label, model.majorityLabel))
            .collect(EvaluationResult.collector(trainingSetSize, testSetSize,
trainRatio));

    return result;
}
```

RecommendationFactory

The `RecommendationFactory` is used to create a new recommender instance. It also defines for which types of layers and features the recommender itself can be used. Here, we decided to only support token span layers without cross sentence annotations.

The RecommendationFactory

```
@ExportedComponent
@Component
public class DataMajorityRecommenderFactory
    extends RecommendationEngineFactoryImplBase<Void>
{
    // This is a string literal so we can rename/refactor the class without it
changing its ID
    // and without the database starting to refer to non-existing recommendation
tools.
    public static final String ID =

"de.tudarmstadt.ukp.inception.recommendation.imls.datamajority.de.tudarmstadt.ukp.ince
ption.recommendation.imls.datamajority.DataMajorityNerRecommender";

    @Override
    public String getId()
    {
        return ID;
    }

    @Override
    public RecommendationEngine build(Recommender aRecommender)
    {
        return new DataMajorityNerRecommender(aRecommender);
    }

    @Override
    public String getName()
    {
        return "Data Majority Recommender";
    }

    @Override
    public boolean accepts(AnnotationLayer aLayer, AnnotationFeature aFeature)
    {
        if (aLayer == null || aFeature == null) {
            return false;
        }

        return (asList(SINGLE_TOKEN, TOKENS).contains(aLayer.getAnchoringMode()))
                && !aLayer.isCrossSentence() && SPAN_TYPE.equals(aLayer.getType())
                && CAS.TYPE_NAME_STRING.equals(aFeature.getType()) || aFeature
.isVirtualFeature();
    }
}
```

External recommender

Overview

This section describes the External Recommender API for INCEpTION. An external recommender is a classifier whose functionality is exposed via a HTTP web service. It can predict annotations for given documents and optionally be trained on new data. This document describes the endpoints a web service needs to expose so it can be used with INCEpTION. The documents that are exchanged are in form of a UIMA CAS. For sending, they have to be serialized to CAS XMI. For receiving, it has to be deserialized back. There are two main libraries available that manage CAS handling, one is the UIMA Java SDK, the other one dkpro-cassis (Python).

Version information

Version : 1.0.0

Contact information

Contact Email : inception-users@googlegroups.com

License information

License : Apache 2.0
License URL : http://www.apache.org/licenses/LICENSE-2.0.html
Terms of service : https://inception-project.github.io

Paths

Predict annotations for a single document

```
POST /predict
```

Description

Sends a CAS together with information about the layer and feature to predict to the external recommender. The external recommender then returns the CAS annotated with predictions.

Parameters

Type	Name	Description	Schema
Body	**body** *required*	Document CAS for which annotations will be predicted	PredictRequest

Responses

HTTP Code	Description	Schema
200	Successful prediction	PredictResponse

Consumes

- `application/json`

Produces

- `application/json`

Tags

- predict

Example HTTP request

Request path

```
/predict
```

Request body

```
{
  "metadata" : {
    "layer" : "de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity",
    "feature" : "value",
    "projectId" : 1337,
    "anchoringMode" : "tokens",
    "crossSentence" : false
  },
  "document" : {
    "xmi" : "<?xml version=\"1.0\" encoding=\"UTF-8\"?> <xmi:XMI xmlns:tcas=
\"http:///uima/tcas.ecore\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:cas=
\"http:///uima/cas.ecore\" xmlns:cassis=\"http:///cassis.ecore\" xmi:version=\"2.0\">
<cas:NULL xmi:id=\"0\"/> <tcas:DocumentAnnotation xmi:id=\"8\" sofa=\"1\" begin=\"0\"
end=\"47\" language=\"x-unspecified\"/> <cas:Sofa xmi:id=\"1\" sofaNum=\"1\" sofaID=
\"mySofa\" mimeType=\"text/plain\" sofaString=\"Joe waited for the train . The train
was late .\"/> <cas:View sofa=\"1\" members=\"8\"/> </xmi:XMI>",
    "documentId" : 42,
    "userId" : "testuser"
  },
  "typeSystem" : "<?xml version=\"1.0\" encoding=\"UTF-8\"?> <typeSystemDescription
xmlns=\"http://uima.apache.org/resourceSpecifier\"> <types> <typeDescription>
<name>uima.tcas.DocumentAnnotation</name> <description/>
<supertypeName>uima.tcas.Annotation</supertypeName> <features> <featureDescription>
<name>language</name> <description/> <rangeTypeName>uima.cas.String</rangeTypeName>
</featureDescription> </features> </typeDescription> </types>
</typeSystemDescription>"
}
```

Example HTTP response

Response 200

```
{
  "document" : "<?xml version=\"1.0\" encoding=\"UTF-8\"?> <xmi:XMI xmlns:tcas=
\"http:///uima/tcas.ecore\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:cas=
\"http:///uima/cas.ecore\" xmlns:cassis=\"http:///cassis.ecore\" xmi:version=\"2.0\">
<cas:NULL xmi:id=\"0\"/> <tcas:DocumentAnnotation xmi:id=\"8\" sofa=\"1\" begin=\"0\"
end=\"47\" language=\"x-unspecified\"/> <cas:Sofa xmi:id=\"1\" sofaNum=\"1\" sofaID=
\"mySofa\" mimeType=\"text/plain\" sofaString=\"Joe waited for the train . The train
was late .\"/> <cas:View sofa=\"1\" members=\"8\"/> </xmi:XMI>"
}
```

Train recommender on a set of documents

```
POST /train
```

Description

Sends a list of CASses to the external recommender for training. No response body is expected.

Parameters

Type	Name	Description	Schema
Body	**body** *required*	List of documents CAS whose annotations will be used for training	Train

Responses

HTTP Code	Description	Schema
204	Successful training	No Content
429	Too many training requests have been sent, the sender should wait a while until the next request	No Content

Consumes

- `application/json`

Tags

- train

Example HTTP request

Request path

```
/train
```

Request body

```
{
  "metadata" : {
    "layer" : "de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity",
    "feature" : "value",
    "projectId" : 1337,
    "anchoringMode" : "tokens",
    "crossSentence" : false
  },
  "documents" : [ {
    "xmi" : "<?xml version=\"1.0\" encoding=\"UTF-8\"?> <xmi:XMI xmlns:tcas=
\"http:///uima/tcas.ecore\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:cas=
\"http:///uima/cas.ecore\" xmlns:cassis=\"http:///cassis.ecore\" xmi:version=\"2.0\">
<cas:NULL xmi:id=\"0\"/> <tcas:DocumentAnnotation xmi:id=\"8\" sofa=\"1\" begin=\"0\"
end=\"47\" language=\"x-unspecified\"/> <cas:Sofa xmi:id=\"1\" sofaNum=\"1\" sofaID=
\"mySofa\" mimeType=\"text/plain\" sofaString=\"Joe waited for the train . The train
was late .\"/> <cas:View sofa=\"1\" members=\"8\"/> </xmi:XMI>",
    "documentId" : 42,
    "userId" : "testuser"
  } ],
  "typeSystem" : "<?xml version=\"1.0\" encoding=\"UTF-8\"?> <typeSystemDescription
xmlns=\"http://uima.apache.org/resourceSpecifier\"> <types> <typeDescription>
<name>uima.tcas.DocumentAnnotation</name> <description/>
<supertypeName>uima.tcas.Annotation</supertypeName> <features> <featureDescription>
<name>language</name> <description/> <rangeTypeName>uima.cas.String</rangeTypeName>
</featureDescription> </features> </typeDescription> </types>
</typeSystemDescription>"
}
```

Definitions

Document

Name	Description	Schema
documentId *optional*	Identifier for this document. It is unique in the context of the project. **Example** : `42`	integer
userId *optional*	Identifier for the user for which recommendations should be made. **Example** : `"testuser"`	string

Name	Description	Schema
xmi *optional*	CAS as XMI **Example** : `"<?xml version=\"1.0\" encoding=\"UTF-8\"?> <xmi:XMI xmlns:tcas=\"http:///uima/tcas.ecore\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:cas=\"http:///uima/cas.ecore\" xmlns:cassis=\"http:///cassis.ecore\" xmi:version=\"2.0\"> <cas:NULL xmi:id=\"0\"/> <tcas:DocumentAnnotation xmi:id=\"8\" sofa=\"1\" begin=\"0\" end=\"47\" language=\"x-unspecified\"/> <cas:Sofa xmi:id=\"1\" sofaNum=\"1\" sofaID=\"mySofa\" mimeType=\"text/plain\" sofaString=\"Joe waited for the train . The train was late .\"/> <cas:View sofa=\"1\" members=\"8\"/> </xmi:XMI>"`	string

Metadata

Name	Description	Schema
anchoringMode *required*	Describes how annotations are anchored to tokens. Is one of 'characters', 'singleToken', 'tokens', 'sentences'. **Example** : `"tokens"`	string
crossSentence *required*	True if the project supports cross-sentence annotations, else False **Example** : `false`	boolean
feature *required*	Feature of the layer which should be predicted **Example** : `"value"`	string
layer *required*	Layer which should be predicted **Example** : `"de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity"`	string
projectId *required*	The id of the project to which the document(s) belong. **Example** : `1337`	integer

PredictRequest

Name	Description	Schema
document *required*	**Example** : `"Document"`	Document
metadata *required*	**Example** : `"Metadata"`	Metadata
typeSystem *required*	Type system XML of the CAS **Example** : `"<?xml version=\"1.0\" encoding=\"UTF-8\"?> <typeSystemDescription xmlns=\"http://uima.apache.org/resourceSpecifier\"> <types> <typeDescription> <name>uima.tcas.DocumentAnnotation</name> <description/> <supertypeName>uima.tcas.Annotation</supertypeName> <features> <featureDescription> <name>language</name> <description/> <rangeTypeName>uima.cas.String</rangeTypeName> </featureDescription> </features> </typeDescription> </types> </typeSystemDescription>"`	string

PredictResponse

Name	Description	Schema
document *required*	CAS with annotations from the external recommender as XMI **Example** : `"<?xml version=\"1.0\" encoding=\"UTF-8\"?> <xmi:XMI xmlns:tcas=\"http:///uima/tcas.ecore\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:cas=\"http:///uima/cas.ecore\" xmlns:cassis=\"http:///cassis.ecore\" xmi:version=\"2.0\"> <cas:NULL xmi:id=\"0\"/> <tcas:DocumentAnnotation xmi:id=\"8\" sofa=\"1\" begin=\"0\" end=\"47\" language=\"x-unspecified\"/> <cas:Sofa xmi:id=\"1\" sofaNum=\"1\" sofaID=\"mySofa\" mimeType=\"text/plain\" sofaString=\"Joe waited for the train . The train was late .\"/> <cas:View sofa=\"1\" members=\"8\"/> </xmi:XMI>"`	string

Train

Name	Description	Schema
documents *required*	CAS as XMI **Example** : `[ "Document" ]`	< Document > array
metadata *required*	**Example** : `"Metadata"`	Metadata
typeSystem *required*	Type system XML of the CAS **Example** : `"<?xml version=\"1.0\" encoding=\"UTF-8\"?> <typeSystemDescription xmlns=\"http://uima.apache.org/resourceSpecifier\"> <types> <typeDescription> <name>uima.tcas.DocumentAnnotation</name> <description/> <supertypeName>uima.tcas.Annotation</supertypeName> <features> <featureDescription> <name>language</name> <description/> <rangeTypeName>uima.cas.String</rangeTypeName> </featureDescription> </features> </typeDescription> </types> </typeSystemDescription>"`	string

Active Learning

The active learning module aims to guide the user through recommendations in such a way that the the judgements made by the user are most informative to the recommenders. The goal is to reduce the required user interactions to a minium. The module consists of the following classes and interfaces:

- The `ActiveLearningService` interface and its default implementation `ActiveLearningServiceImpl` which provide access to the ranked suggestions.
- The `ActiveLearningStrategy` interface which allows plugging in different sampling strategies.
- The `UncertaintySamplingStrategy` class which is currently the only sampling strategy available.
- The `ActiveLearningSidebar` class which provides the active learning sidebar for the annotation page. Here the user can accept/reject/correct/skip suggestions.

The active learning module relies on the recommendation module for the actual annotation recommendations. This means that the active learning module does not directly make use of the user feedback. If suggestions are accepted, they are used in the next train/predict run of the recommendation module as training data. The active learning module then samples the new annotation suggestions from this run and updates the order in which it offers the suggestions to the user.

[diag a06cf7943ca7daddfb1cc80682508375] | *diag-a06cf7943ca7daddfb1cc80682508375.png*

Events

- `ActiveLearningSuggestionOfferedEvent` - active learning has pointed the user at a recommentation
- `ActiveLearningRecommendationEvent` - user has accepted/rejected a recommendation
- `ActiveLearningSessionStartedEvent` - user has opened an active learning session
- `ActiveLearningSessionCompletedEvent` - user has closed the active learning session

Sampling strategies

Uncertainty sampling

Currently, there is only a single sampling strategy, namely the `UncertaintySamplingStrategy`. It it compares the confidence scores of the annotation suggestion. The smaller the difference between the best and the second best score, the earlier the suggestion is proposed to the user. The confidence scores produced by different recommenders can be on different scales and are therefore not really comparable. Thus, the strategy only compares suggestions from the same recommender to each other. So if recommender A produces two suggestions X and Y, they are compared to each other. However, if there are two recommenders A and B producing each one suggestion X and Y, then X and Y are not compared to each other.

Event Log

The event logging module allows catching Spring events and logging them them to the database. It consists of the following classes and interfaces:

- The `EventRepository` interface and its default implementation `EventRepositoryImpl` which serve as the data access layer for logged events.
- The `EventLoggingListener` which hooks into Spring, captures events, and then uses the `EventRepository` to log them.
- The `EventLoggingAdapter` interface. Spring components implementing this interface are used to extract information from Spring events and to convert them into a format suitable to be logged.
- The `LoggedEvent` entity class which maps the logged events to the database.
- The `LoggedEventExporter` and `ExportedLoggedEvent` which are used to export/import the event log as part of a project export/import.

The log module comes with a number of adapters for common events such as annotation manipulation, changes to the project configuration, etc. Any event for which no specific adapter exists is handled using the `GenericEventAdapter` which logs only general information (e.g. the timestamp, current user, type of event) but no event-specific details (e.g. current project, current document, or even more specific details). Note that even the `GenericEventAdapter` skips logging certain Spring events related to session management, authorization, and the Spring context life-cycle.

Event Logging Adapters

New logging adapters should be created in the module which provides the event they are logging. Logging adapters for events generated outside INCEpTION (i.e. in upstream code) are usually added to the log module itself.

To add support for a logging a new event, create a Spring component class which implements the `EventLoggingAdapter` interface. Implement the following methods depending on the context in which the event is triggered:

- `getProject(Event)` if the event is triggered in the context of a specific project (applies to most events);
- `getDocument(Event)` if the event is related to a specific source document (e.g. applies to events triggered during annotation).
- `getDocument(Event)` if the event is related to a specific annotator (e.g. applies to events triggered during annotation).

The methods `getEvent`, `getUser` and `getCreated` normally do not need to be implemented.

Most event adapters implement the `getDetails` method. This method must return a JSON string which contains any relevant information about the event not covered by the methods above. E.g. for an annotation manipulation event, it would contain information helping to identify the annotation and the state before and after the manipulation. In order to generate this JSON string, the adapter typically contains an inner class called `Details` to which the detail information from the event is copied and which is then serialized to JSON using `JSONUtil.toJsonString(…)`.

Knowledge base

Schema mapping

An IRI Schema defines the following attributes that are used for making queries in a knowledge base.

Table 1. Schema Mapping Attributes

Attribute	Description	Example Value
Class IRI	Class of resources that are classes.	rdfs:Class
Subclass IRI	Property that defines a **subclass of** relation between classes.	rdfs:subClassOf
Type IRI	Property that defines which class a resource belongs to	rdf:type
Label IRI	Property that defines a human readable label for a class or instance	rdfs:label
Description IRI	Property that defines a description for a class or instance	rdfs:comment
Property IRI	Class of resources that are properties	rdf:Property
Subproperty IRI	Property that defines a **subproperty of** relation between properties.	rdfs:subPropertyOf
Property Label IRI	Property that defines a human readable label for a property	rdfs:label
Property Description IRI	Property that defines a description for a property	rdfs:comment

There are multiple classes in the knowledge base module that model the IRI Schema of a knowledge base. All the classes share that they have a single class-attribute for each IRI in the IRI Schema. However each class has a different use case. The relevant classes are shown here.

ℹ If the structure of the general IRI Schema is changed (e.g. a new attribute is added) all the classes need to be adjusted.*

Table 2. Knowledge base & schema mapping classes

Class	Usage
`KnowledgeBase`	General model for a knowledge base in frontend and backend components.

Class	Usage
`KnowledgeBaseProfile` and `KnowledgeBaseMapping`	Read pre-configured knowledge base profiles from a yaml file. The actual IRI Schema is modeled in KnowledgeBaseMapping.java. The yaml file is located at: `…/inception-kb/src/main/resources/de/tudarmstadt/ukp/inception/kb/knowledgebase-profiles.yaml`
`SchemaProfile`	Defines some specific IRI Schemas (e.g RDF, WIKIDATA, SKOS).
`ExportedKnowledgeBase`	Export a knowledge base configuration when a project is exported.

Concept Linking

The concept linking module is used to find items from a knowledge base that match a certain query and context. It is used e.g. by the `ConceptFeatureEditor` to display items which match a concept mention and it can use the mention’s context to rank (and optimally disambiguate) the candidate items. It can also be used for non-contextualized queries, e.g. via the search field on the knowledge base browsing page. The module consists of the following classes and interfaces:

- The `ConceptLinkingService` interface and its default implementation `ConceptLinkingServiceImpl` which is the main entry point for locating KB items.
- The `EntityRankingFeatureGenerator` interface. Spring beans which implement this interface are automatically picked up by the `ConceptLinkingServiceImpl` and used to rank candidates.

Ranking

Feature generators

The module currently uses primarily the `LevenshteinFeatureGenerator` which calculate the Levenshtein distance between the mention text and the KB item label as well as between the query text (e.g. entered into the auto-complete field of the `ConceptFeatureEditor`) and the KB item label.

Ranking strategy

The ranking method is currently hard-coded in `ConceptLinkingServiceImpl.baseLineRankingStrategy()`.

Named entity linking recommender

The module also includes the `NamedEntityLinker` recommender which can be used to generate annotation recommendations. It gets triggered for any `NamedEntity` annotations and suggests which KB items to link them to.

PDF Annotation Editor

The PDF-Editor module allows the view and annotation of PDF documents. This is implemented using *PDFAnno*, *PDFExtract* and *DKPro PDF Reader*. The choice for *PDFAnno* and other implementation choices are explained in the following.

Selecting a PDF Annotation Tool

There are only few requirements to a PDF annotation tool for integration into *INCEpTION*. It must provide support for span and relation annotations and it should also be lightweight and easily modifiable to fit into *INCEpTION*.

There are two PDF annotation tools up for discussion. The first one is *PDFAnno* and the second is *Hypothes.is*. Both tools are web-based and open source software available on *GitHub*.

PDFAnno is a lightweight annotation tool that only supports the PDF format. It was created specifically to solve the lack of free open source software for annotating PDF documents which is also capable of creating relations between annotations. This is described in the publication about *PDFAnno* by Shindo et al.

Hypothes.is is a project that was created to provide an annotation layer over the web. The idea is to be able to create annotations for all content available on the internet and to share it with other people. Hence *Hypothes.is* provides the functionality to annotate PDF documents.

PDFAnno compared to *Hypothes.is* comes with a smaller code base and is less complex. Both editors feature span annotations, however only *PDFAnno* provides the functionality to create relations between span annotations which is required in *INCEpTION*. As *Hypothes.is* was designed to share annotations with others a login mechanism is part of the software.

PDFAnno provides relations, is more lightweight and does not have a login functionality, which would have to be removed. Hence *PDFAnno* fits the requirements better than *Hypothes.is* and was chosen as the PDF annotation tool for integration into *INCEpTION*.

Differences in PDF Document Text Extractions

PDFAnno uses PDF.js to render PDF documents in the browser. The tool *PDFExtract* is used to extract information about the PDF document text. It produces a file in which each line contains information about one character of the text. Information includes the page, the character and the position coordinates of the character in the PDF document, in the given order and separated by a tab character. An example:

```
1  E  0 1 2 3
1  x  4 5 6 7
1  a  8 9 10 11
1  m  12 13 14 15
1  p  16 17 18 19
1  l  20 21 22 23
1  e  24 25 26 27
2  [MOVE_TO]  28 29
2  NO_UNICODE  30 31 32 33
```

There are also draw operations included which are of no relevance for the use in *INCEpTION*. Characters which have no unicode mapping have the value `NO_UNICODE`. The *PDFExtract* file does not contain information about any whitespaces that occur in the PDF document text. *PDFAnno* requires the PDF document and the *PDFExtract* file to work. The PDF document can be obtained from the *INCEpTION* backend. To also provide the *PDFExtract* file, the tool was slightly modified so that it can be used as a library in *INCEpTION*.

PDFAnno provides an API for handling annotations. It is possible to import a list of annotations by providing an URL for download. This list has to be in the *TOML* format. Span annotations require the begin and end positions of the characters it covers. This positions are equal to the line number of characters in the *PDFExtract* file. A span annotation example in TOML format:

```
[[span]]
id = "1"
page = 1
label = ""
color = "#ff00ff"
text = "Example"
textrange = [1, 7]
```

The *Brat* editor used in *INCEpTION* works only on plain text. For PDF documents this plain text is obtained by the use of *DKPro PDF Reader*. The reader extracts the text information from the PDF document and performs operations to ensure correct text ordering and to replace certain character sequences with substitutes from a substitution table.

As the extractions between *PDFAnno* and *INCEpTION* differ a mapping of those representations must be implemented to ensure annotations can be exchanged between the frontend and the backend and are usable across all editor modes.

Preparing Representations

To use a mapping method between the text representation of *PDFAnno* and *INCEpTION* at first they must be preprocessed to have a similar structure.

As the *PDFExtract* file does not only contain the text string, first the characters of the file need to be obtained and appended to a string. All draw operations and `NO_UNICODE` lines are ignored. As *DKPro PDF Reader* uses a substitution table to sanitize the document text, the same substitution table is used to sanitize the obtained string.

The *PDFExtract* file does not contain any whitespaces present in the document text, however *DKPro PDF Reader* preserves them. The whitespaces are removed from the *DKPro PDF Reader* string to have a similar structure to the *PDFExtract* sanitized string content.

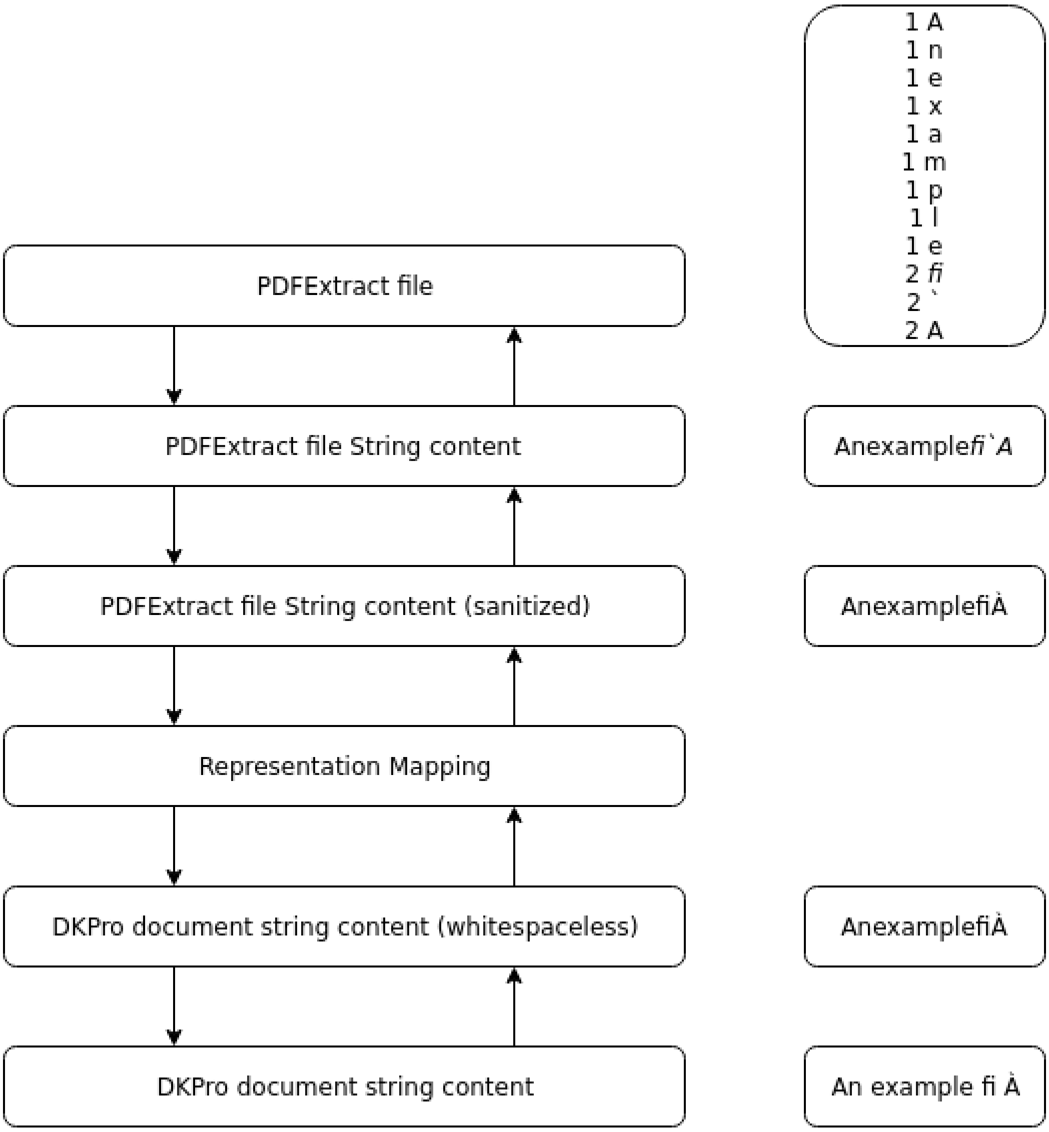
1 A
1 n
1 e
1 x
1 a
1 m
1 p
1 l
1 e
2 ﬁ
2 `
2 A
PDFExtract file
PDFExtract file String content
Anexampleﬁ`A
PDFExtract file String content (sanitized)
AnexamplefiÀ
Representation Mapping
DKPro document string content (whitespaceless)
AnexamplefiÀ
DKPro document string content
An example fi À

Figure 1. Mapping Process (left) with examples (right)

Even though both representations now are in a similar shape it can still happen that the content in both strings differs. For example ordering of text areas could be messed up which can especially happen for PDF documents that contain multiple text columns on one page. As both representations are not equal even after preprocessing, a mapping algorithm has to be implemented to find the text of annotations from one representation in the respective other representation.

Mapping Annotations

There are multiple ways to achieve a mapping between *PDFAnno* and *INCEpTION* for annotations. Two methods were tested during development: exact string search with context and sequence alignment.

The first option is to make an exact search for the annotation text. However as annotations often cover only one token an exact search for the annotation text would result in multiple occurrences. To get a unique result it is required to add context to the annotation text. As this still can yield multiple occurrences, context is expanded until a unique mapping or no mapping at all is found. Performing this for all annotations results in a lot of string search operations. However the performance can be improved by searching for all annotations in the target string at once with the help of the Aho-Corasick algorithm.

Another approach is to use sequence alignment methods which are popular in bioinformatics. PDF document texts are rather large and most sequence alignment algorithms require O(M x N) memory space, where M and N are the size of the two sequences. This results in a large memory consumption on computing the alignment, hence an algorithm should be used that works with less memory. Such an algorithm is Hirschbergs algorithm. It consumes only O(min(M,N)) memory.

The advantage of the sequence alignment method would be a direct mapping between the representation of *PDFExtract* and *DKPro PDF Reader*. However, during testing for larger documents, for example 40 pages, the duration until Hirschbergs algorithm finished was too long and would be unsatisfying for a user. The exact string search however takes increasingly longer to compute mappings the larger the document is and the more annotations have to be mapped. As discussed the Aho-Corasick algorithm reduces the time. However, this still does not scale well for larger documents. To overcome this issue a page wise rendering of annotations was introduced. When navigating through the PDF document in *PDFAnno* annotations are rendered dynamically per page. In detail, this means whenever the user moves through the document, the current page changes and the user stops movement for 500 ms, the annotations for the previous, current and next page are rendered. This way large documents can be handled by the PDF editor without long wait times for the user.

The exact string search seemed to perform well in terms of finding matching occurrences for annotations in both directions. For the manually tested documents all annotations were found and matched.

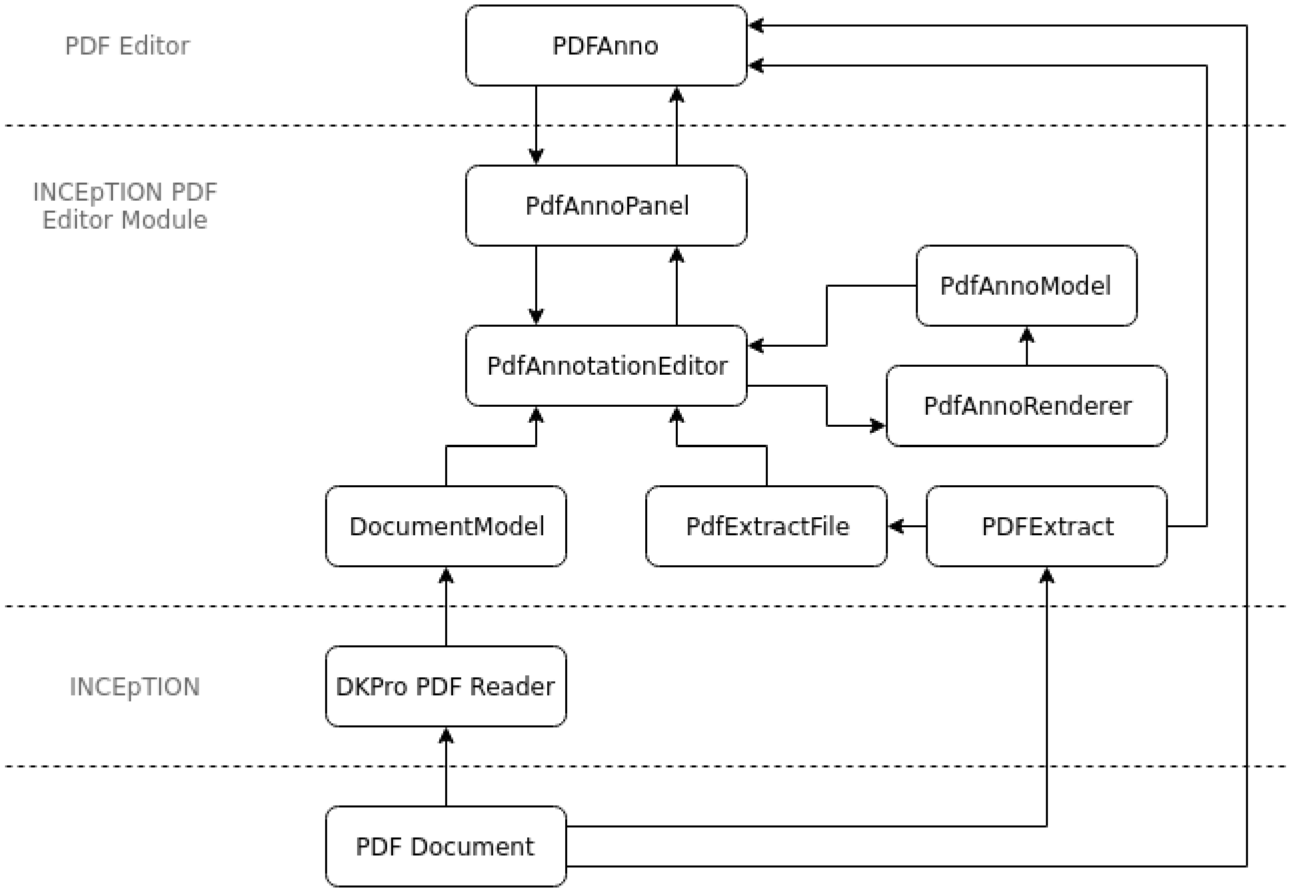

Figure 2. PDF Editor Architecture

INCEpTION Development

Releasing INCEpTION

Prerequisites

When releasing and publishing the resulting artifact, make sure that you actually have the rights to publish new artifacts to the target repository. See e.g. the Maven documentationon authentication for more information on how to do this. Additionally, this documentation assumes that you have write permissions to the protected `master` branch and to the protected maintenance branches.

Branching model

INCEpTION uses a branching model where the development towards the next feature release happens on the `master`. Additionally, there are **maintenance** (also called **stable**) branches for all past feature releases which are used to create bugfix releases when necessary (`0.1.x`, `0.2.x`, `0.3.x` etc.).

To prepare a new feature release, we first create a maintenance branch from `master`. Assuming the current development version on the `master` branch is `0.8.0-SNAPSHOT`, we create a maintenance branch by the name `0.8.x`. After this, the version on the master branch must be manually updated to the next feature version, typically by increasing the second digit of the version (e.g. `0.8.0-SNAPSHOT` becomes `0.9.0-SNAPSHOT`).

The process of doing the initial feature release on a new maintenance branch (e.g. for version `0.8.0`) is largely the same then as for performing a bugfix release (e.g. for version `0.8.1`) as described in the next section.

Changes are usually not committed directly to the `master` or maintenance branches. Instead we work with **feature** branches. Feature branches for new enhancements or refactorings are normally made from the `master` branch. Feature branches for bugfixes are made from a maintenance branch (unless the bug only exists on the `master` branch or is too difficult to fix without extensive refactoring and risk of breaking other things). After accepting a pull request for a bugfix to a maintenance branch, a developer with write access to the `master` branch should regularly merge the maintenance branch into the `master` branch to have the fix both in maintenance and `master`. From time to time, a new bugfix release is made. Here, the third digit of the version is increased (e.g. from `0.8.0` to `0.8.1`).

The first digit of the version can be increased to indicate e.g. major improvements, major technological changes or incompatible changes. This is done only very rarely (maybe once every 1-3 years).

Release steps

For these steps, we assume push permissions to the master branch. If these are not given, pull requests have to be used for the relevant steps. For the sake of this guide, we assume that `0.8.x` is the most recent maintenance branch.

Bugfix release preparations

Do this when you want to do a bugfix release.

1. Create a bugfix release issue in the INCEpTION Github issue tracker
2. Checkout the `master` branch: e.g. `git checkout master`
3. Merge the maintenance into the `master` branch; this makes sure that all bugfixes are also applied to master: e.g. `git merge 0.8.x` and then `git push`
4. Checkout the maintenance branch of which you want to make a bugfix release, by e.g. `git checkout 0.8.x`

Feature release preparations

Do this when you want to do a feature release.

1. Checkout `master` and make sure that you pulled the most recent version of it
2. Create a new branch from `master` and name it according to the new version with the last digit (the bugfix part of the version) being `x`, by e.g. `git checkout -b 0.9.x`

Running the release process

1. Run `mvn release:prepare`; this step will ask you for the release version (e.g. `0.9.0`) for the tag to be created (e.g. `inception-app-0.9.0`), and for the new development version (e.g. `0.10.0-SNAPSHOT`). After that, it will update the version (push), build the code, and create a tag (push).
2. Run `mvn release:perform`; this will check out the created tag into the folder `target/checkout` and build the release artifacts there.
3. Immediately after the release, again merge the released branch into the master branch, using e.g. `git checkout master` and `git merge -s ours 0.9.x` and `git push`. Doing this second merge does not actually change anything in the master. However, it ensures that the updates to the version number in the released_branch (e.g. `0.9.x`) will not make it to the master branch.
4. The released jar is now in `target/checkout/inception-app-webapp/target/` (**not** in `inception-app-webapp/target/`), look for the standalone jar.

Making the release available

1. Sign the standalone jar using `gpg --detach-sign --armor`. This creates the signature in a separate file. Attach both files to the GitHub release that you are currently preparing. GitHub automatically creates a stub release for every tag and the Maven release process has automatically created such a tag for your release. The issues that were solved with this release can be found by filtering issues by milestones.

Update Github milestones

After the release you will need to close the respective milestone on Github to prevent further issues or pull requests being assigned to it.

Deploy the new release

In order to deploy the new version, see the upgrade process in the Administrator Guide.

Updating documentation

Checkout or go into the `inception-project.github.io` repository and create a new branch. In order to publish the documentation for this new release, copy the data from the INCEpTION repository that was just released under `target/checkout/inception-app-webapp/target/generated-docs` into a version folder under `releases/{version}/docs` in the documentation repository. Also update the `_data/releases.yml` file. Create a pull request for this as well.

Publish to docker

Make sure that you configured your docker user in your maven settings:

```
<server>
  <id>docker.io</id>
  <username>user</username>
  <password>password</password>
</server>
```

Also, your user needs to have push rights to the `inceptionproject` Dockerhub group. Check out the release tag, make sure that in the master pom, the version is set to a release version (no snapshot suffix). Then run

```
mvn -Pdocker clean install docker:build
-Ddocker.image.name="inceptionproject/inception"
mvn -Pdocker clean install docker:push
-Ddocker.image.name="inceptionproject/inception"
```

Aborting and re-running a release

If for some reason the release process failed, re-run the maven command during which the process was aborted (i.e. `mvn release:prepare` or `mvn release:perform`). Maven should repeat any failed steps of the respective process.

If you need to abort the release process use `mvn release:rollback`. However, in this case, you also need to check if the release tag was already created. You then might need to manually remove it. You might also need to revert the commits that were created during the release to re-set the version to the previous state. This can be done with `git revert <commit>`.

Appendices

Appendix A: System Properties

Setting	Description	Default	Example
wicket.configuration	Enable Wicket debug mode	deployment	development