WebAnno Developer Guide

The WebAnno Team

Version 3.6.2

Table of Contents

This document targets developers working on WebAnno.

Introduction

Setup

GIT

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.

Setting up the for 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 WebAnno into the Workspace

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

```
$ git clone https://github.com/webanno/webanno.git
```

In Eclipse, go to **File → Import**, choose **Existing Maven projects**, and select the folder to which you have cloned WebAnno. 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 `-Dwebanno.home="/srv/webanno"` 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...**:

WebAnno 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 WebAnno 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.

Checkstyle and Formatting

We use a style for formatting the source code in WebAnno. 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)`)

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.

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 WebAnno projects, right click and do a **Maven → Update project**

Should the steps mentioned above not have been sufficient, close all the WebAnno projects in Eclipse, then remove them form the workspace (not from the disk), delete any `.checkstyle` files in the WebAnno 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.

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.

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.

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.

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.

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.

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.

Annotation Editor

Paging

Typically, an annotation editor only shows a a part of a document on screen to allow working with large files without running into performance problems during rendering. The paging strategy defines how a document is divided into the units (e.g. lines, sentences, paragraphs, pages, etc.) from which the visible part of the document is constructed. The `AnnotationEditorFactory.initState()` method is intended to set up the editor state to be compatible with the editor. In particular, it is the place where the editor can inject its paging strategy. Contrary to many other modularized parts of the code, paging strategies are not Spring beans but simple stateless and serializable Java classes.

There are presently two paging strategies:

- `SentenceOrientedPagingStrategy` - divides the document into sentences using the `Sentence` annotations present in the CAS.
- `LineOrientedPagingStrategy` - divides the document into lines considering using the newline character (LF, dec: 10, hex: 0A) as a line separator.

A `PagingStrategy` has to implement three methods:

- `List<Unit> units(CAS, n, m)` to extract units **n** through **m** from the given CAS. If the value provided for **m** is greater than the number of units in the document, the returned list simply contains the maximum available units. If **n** is greater than the number of available units, an empty list is returned.
- `createPositionLabel(⋯)` creates a UI element displaying the current position within the document as well as which document is currently opened.
- `createPageNavigator(⋯)` creates a UI element to navigate through the pages. For most cases, the provided `DefaultPagingNavigator` should be fine.
- (optional) `unitCount(⋯)` fetches the number of units in the given CAS. It is usually faster to provide a special implementation for this method than to rely on the default implementation which resorts to calling `units(⋯).size()`.

If an annotation editor needs access to the visible units (e.g. to split spans overlapping units into multiple ranges), it can call `AnnotatorState.getVisibleUnits()`.

Appendices

System Properties

Setting	Description	Default	Example
wicket.core.settings.general.configuration-type	Enable Wicket debug mode	deployment	development