The GATE Embedded API

Track II, Module 5

Sixth GATE Training Course
June 2013

© 2013 The University of Sheffield
This material is licenced under the Creative Commons
Attribution-NonCommercial-ShareAlike Licence
(http://creativecommons.org/licenses/by-nc-sa/3.0/)

Outline

Before We Start

Prerequisites

- Java 6 or 7 JDK (OpenJDK or Oracle)
- latest GATE Developer/Embedded (version 7.1)
- Java Development Environment such as Eclipse (not compulsory but *highly* recommended!).

Hands-on resources

- Download hands-on resources from the participants' wiki
- Unzip it somewhere on your hard disk
- Have a look at the build.xml
- Create a Java project in your IDE (call it `module5`)
- Write a Hello World program to test your configurations

Your First GATE-Based Project

Libraries to include

- `<gate-install-dir>/bin/gate.jar`
- `<gate-install-dir>/lib/*.jar`

Documentation

GATE Documentation

Core documentation from Sheffield

- The Developer and Embedded User Guide
- GATE Developer screencams
- The mailing list
- The core GATE plugins list
- Papers / theory (and how to cite our work)
- Brochures: 4 pages or 2-pages
- Acres of API and software docs, (Javadocs) and example code
- A May 2010 summary presentation
- For technical help please see the support page (or below)
- GATE's public wiki (running on GATE wiki, of course)

For Eclipse users

- Open your Eclipse preferences (in the usual place on the application menu for Mac, under *Window* on other platforms)
- Java → Build Path → User Libraries
- Create a new library named *GATE*
- *Add JARs...*
  - add `bin/gate.jar` and all the JARs in `gate/lib`
- expand the `gate.jar` entry, edit the "Source attachment" and point it at the `src` directory in your GATE.
- OK
- Add this user library to the build path of your module5 project.
- Eclipse configurations for later modules assume this library.

Exercise 1: Loading a Document

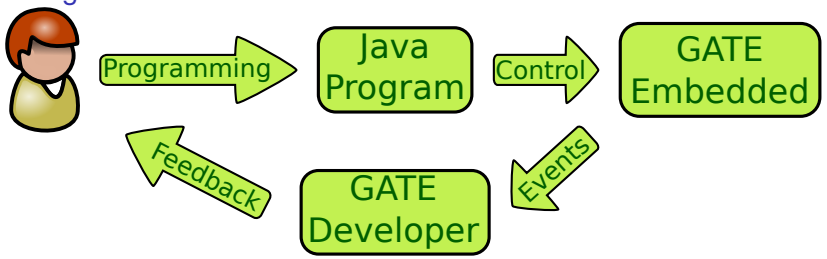
Try this:

```
1 import gate.*;
2 public class Main {
3   public static void main(String[] args)
4       throws Exception{
5     Gate.init(); // prepare the library
6     //create a new document
7     Factory.newDocument("This is a document");
8   }
9 }
```


Interacting with GATE

Using GATE Developer

Using GATE API



Loading a Document (take 2)

```
1  package gatetutorial;
2
3  import gate.*;
4  import gate.gui.*;
5
6  public class Main {
7    public static void main(String[] args)
8      throws Exception{
9      //prepare the library
10     Gate.init();
11     //show the main window
12     MainFrame.getInstance().setVisible(true);
13     //create a document
14     Factory.newDocument("This is a document");
15   }
16 }
```

Note for (at least) Mac Users

If the previous slide doesn't work for you, you may need:

```
3  import javax.swing.SwingUtilities;

11 // show the main window
12 SwingUtilities.invokeAndWait(new Runnable() {
13   public void run() {
14     MainFrame.getInstance().setVisible(true);
15   }
16 });
```


Outline

CREOLE

The GATE component model is called CREOLE (**C**ollection of **RE**usable **O**bjects for **L**anguage **E**ngineering).

CREOLE uses the following terminology:

- **CREOLE Plugins**: contain definitions for a set of resources.
- **CREOLE Resources**: Java objects with associated configuration.
- **CREOLE Configuration**: the metadata associated with Java classes that implement CREOLE resources.

CREOLE Plugins

CREOLE is organised as a set of plugins.

Each CREOLE plugin:

- is a directory on disk (or on a web server);
- is specified as a URL pointing to the **directory**;
- contains a special file called `creole.xml`;
- may contain one or more .jar files with compiled Java classes.
  - alternatively, the required Java classes may simply be placed on the application classpath.
- contains the definitions for a set of CREOLE resources.

CREOLE Resources

A CREOLE resource is a Java Bean with some additional metadata.

A CREOLE resource:

- must implement the `gate.Resource` interface;
- must provide accessor methods for its parameters;
- must have associated CREOLE metadata.

The CREOLE metadata associated with a resource:

- can be provided inside the `creole.xml` file for the plugin;
- can be provided as special Java annotations inside the source code (recommended).

More details about this in Module 7!

Outline

GATE Resource Types

There are three types of resources:

- **Language Resources (LRs)** used to encapsulate data (such as documents and corpora);
- **Processing Resources (PRs)** used to describe algorithms;
- **Visual Resources (VRs)** used to create user interfaces.

The different types of GATE resources relate to each other:

- PRs run over LRs,
- VRs display and edit LRs,
- VRs manage PRs, . . .

These associations are made via CREOLE configuration.

GATE Feature Maps

Feature Maps. . .

- are simply Java Maps, with added support for firing events.
- are used to provide parameter values when creating and configuring CREOLE resources.
- are used to store metadata on many GATE objects.

All GATE resources are feature bearers (they implement **`gate.util.FeatureBearer`**):

```
1 public interface FeatureBearer{
2   public FeatureMap getFeatures();
3
4   public void setFeatures(FeatureMap features);
5 }
```

FeatureMap Implementation

gate.FeatureMap

```
1 public interface FeatureMap extends Map<Object, Object>
2 {
3   public void removeFeatureMapListener(
4       FeatureMapListener l);
5   public void addFeatureMapListener(
6     FeatureMapListener l);
7 }
```

Events: gate.event.FeatureMapListener

```
1 public interface FeatureMapListener
2     extends EventListener
3 {
4   public void featureMapUpdated();
5 }
```


Resource Parameters

The behaviour of GATE resources can be affected by the use of parameters.

Parameter values:

- are provided as populated feature maps.
- can be any Java Object;
- This includes GATE resources!

Parameter Types

There are two types of parameters:

Init-time Parameters

- Are used during the instantiating resources.
- Are available for all resource types.
- Once set, they cannot be changed.

Run-time Parameters

- are only avaialable for Processing Resources.
- are set before executing the resource, and are used to affect the behaviour of the PR.
- can be changed between consecutive runs.

Creating a GATE Resource

Always use the GATE Factory to create and delete GATE resources!

gate.Factory

```
1 public static Resource createResource(
2   String resourceClassName,
3   FeatureMap parameterValues,
4   FeatureMap features,
5   String resourceName){
6   ...
7 }
```

Only the first parameter is required; other variants of this method are available, which require fewer parameters.

Creating a GATE Resource

You will need the following values:

- **`String resourceClassName`**: the class name for the resource you are trying to create. This should be a string with the fully-qualified class name, e.g. `"gate.corpora.DocumentImpl"`.
- **`FeatureMap parameterValues`**: the values for the init-time parameters. Parameters the are not specified will get their default values (as described in the CREOLE configuration). It is an error for a required parameter not to receive a value (either explicit or default)!
- **`FeatureMap features`**: the initial values for the new resource's features.
- **`String resourceName`**: the name for the new resource.

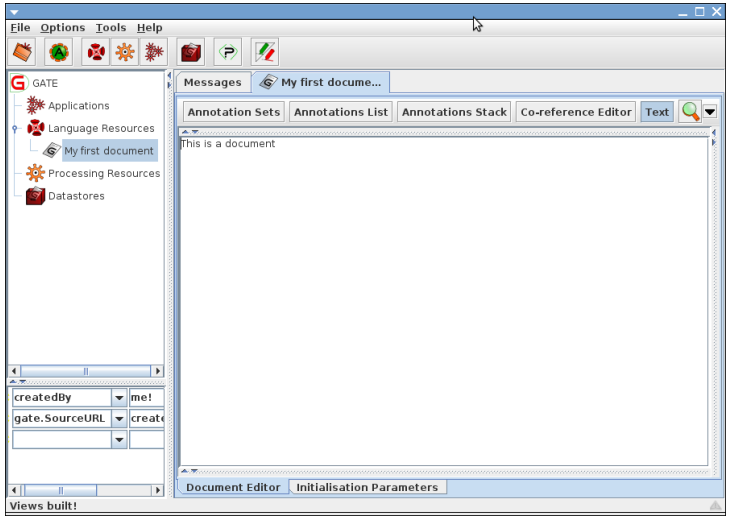
Example: Load a Document (take 3)

```
1 FeatureMap params = Factory.newFeatureMap();
2 params.put(
3     Document.DOCUMENT_STRING_CONTENT_PARAMETER_NAME,
4     "This is a document!");
5 FeatureMap feats = Factory.newFeatureMap();
6 feats.put("createdBy", "me!");
7 Factory.createResource("gate.corpora.DocumentImpl",
8     params, feats, "My first document");
```

TIP: Resource Parameters

The easiest way to find out what parameters resources take (and which ones are required, and what types of values they accept) is to use the GATE Developer UI and try to create the desired type of resource in the GUI!

Example: Load a Document (take 3)



Shortcuts for Loading GATE Resources

Loading a GATE **document**

```
1 import gate.*;
2 //create a document from a String content
3 Document doc = Factory.newDocument("Document text");
4 //...or a URL
5 doc = Factory.newDocument(new URL("http://gate.ac.uk"));
6 //...or a URL and a specified encoding
7 doc = Factory.newDocument(new URL("http://gate.ac.uk"),
8     "UTF-8");
```

Loading a GATE **corpus**

```
1 Corpus corpus = Factory.newCorpus("Corpus Name");
```

Exercise 2: Loading a Document (again!)

Load a document:

- using the GATE home page as a source;
- using the UTF-8 encoding;
- having the name "This is home";
- having a feature named "`date`", with the value the current date.

TIP: Make sure the GATE Developer main window is shown to test the results!

Outline

GATE Documents

A GATE Document comprises:

- a DocumentContent object;
- a Default annotation set (which has no name);
- zero or more named annotation sets;

A Document is also a type of Resource, so it also has:

- a name;
- features.

Main Document API Calls

```
1  // Obtain the document content
2  public DocumentContent getContent();
3  // Get the default annotation set.
4  public AnnotationSet getAnnotations();
5  // Get a named annotation set.
6  public AnnotationSet getAnnotations(String name);
7  // Get the names for the annotation sets.
8  public Set<String> getAnnotationSetNames();
9  // Get all named annotation sets.
10 public Map<String, AnnotationSet>
11   getNamedAnnotationSets();
12 // Convert to GATE stand-off XML
13 public String toXml();
14 // Convert some annotations to inline XML.
15 public String toXml(Set aSourceAnnotationSet,
16   boolean includeFeatures);
```

Annotation Sets

GATE Annotation Sets...

- maintain a set of **`Node`** objects (which are associated with offsets in the document content);
- and a set of annotations (which have a start and an end node).
- implement the **`gate.AnnotationSet`** interface;
- ...which extends **`Set<Annotation>`**.
- implement several **`get()`** methods for obtaining the included annotations according to various constraints.
- are created, deleted, and managed by the Document they belong to.

TIP: always use a Document object to create a new annotation set! Do not use the constructor!

Main AnnotationSet API Calls

Nodes

```
1 // Get the node with the smallest offset.
2 public Node firstNode();
3 // Get the node with the largest offset.
4 public Node lastNode();
```

Creating new Annotations

```
1 // Create (and add) a new annotation
2 public Integer add(Long start, Long end,
3     String type, FeatureMap features);
4 // Create (and add) a new annotation
5 public Integer add(Node start, Node end,
6     String type, FeatureMap features)
```


AnnotationSet API (continued)

Getting Annotations by ID, or type

```
1 // Get annotation by ID
2 public Annotation get(Integer id);
3 // Get all annotations of one type
4 public AnnotationSet get(String type)
5 // Get all annotation types present
6 public Set<String> getAllTypes()
7 // Get all annotations of specified types
8 public AnnotationSet get(Set<String> types)
```


AnnotationSet API (continued)

Getting Annotations by position

```
1 // Get all annotations starting at a given
2 // location, or right after.
3 public AnnotationSet get(Long offset)
4 // Get all annotations that overlap an interval
5 public AnnotationSet get(Long startOffset,
6     Long endOffset)
7 // Get all annotations within an interval.
8 public AnnotationSet getContained(Long startOffset,
9     Long endOffset)
10 // Get all annotations covering an interval.
11 public AnnotationSet getCovering(String neededType,
12     Long startOffset, Long endOffset)
```

AnnotationSet API (continued)

Combined get methods

```
// Get by type and feature constraints.
public AnnotationSet get(String type,
    FeatureMap constraints)
// Get by type, constraints and start position.
public AnnotationSet get(String type,
    FeatureMap constraints, Long offset)
// Get by type, and interval overlap.
public AnnotationSet get(String type,
    Long startOffset, Long endOffset)
// Get by type and feature presence
public AnnotationSet get(String type,
    Set featureNames)
```


Exercise 3: The AnnotationSet API

For the document loaded in excercise 2:

- find out how many named annotation sets it has;
- find out how many annotations each set contains;
- for each annotation set, for each annotation type, find out how many annotations are present.

TIP: Make sure the GATE Developer main window is shown to test the results!

Annotations

GATE Annotations...

- are metadata associated with a document segment;
- have a type (**`String`**);
- have a start and an end Node (**`gate.Node`**);
- have features;
- are created, deleted and managed by annotation sets.

TIP: always use an annotation set to create a new annotation! Do not use the constructor.

Annotation API

Main Annotation methods:

```
public String getType();
public Node getStartNode();
public Node getEndNode();
public FeatureMap getFeatures();
```

`gate.Node`

```
public Long getOffset();
```

Exercise 4: Annotation API

Implement the following:

- Use the document created in exercise 3;
- Use the annotation set **`Original markups`** and obtain annotations of type **`a`** (anchor).
- Iterate over each annotation, obtain its features and print the value of **`href`** feature.

TIP: Before printing the value of **`href`** feature, use the **`new URL(URL context, String spec)`** constuctor such that the value of the **`href`** feature is parsed within the context of the document's source url.

Outline

GATE Processing Resources

Processing Resources (**PRs**) are java classes that can be executed.

`gate.Executable`

```
1 public interface Executable {
2   public void execute() throws ExecutionException;
3   public void interrupt();
4   public boolean isInterrupted();
5 }
```

`gate.ProcessingResource`

```
1 public interface ProcessingResource
2   extends Resource, Executable
3 {
4   public void reInit()
5     throws ResourceInstantiationException;
6 }
```


Language Analysers

Analysers are PRs that are designed to run over the documents in a corpus.

```
1  public interface LanguageAnalyser
2      extends ProcessingResource {
3
4    // Set the document property for this analyser.
5    public void setDocument(Document document);
6
7    // Get the document property for this analyser.
8    public Document getDocument();
9
10   // Set the corpus property for this analyser.
11   public void setCorpus(Corpus corpus);
12
13   // Get the corpus property for this analyser.
14   public Corpus getCorpus();
```

Loading a CREOLE Plugin

- Documents and corpora are built in resource types.
- All other CREOLE resources are defined as plugins.
- Before instantiating a resource, you need to load its CREOLE plugin first!

Loading a CREOLE plugin

```
1 //get the root plugins dir
2 File pluginsDir = Gate.getPluginsHome();
3 //Let's load the Tools plugin
4 File aPluginDir = new File(pluginsDir, "Tools");
5 //load the plugin.
6 Gate.getCreoleRegister().registerDirectories(
7   aPluginDir.toURI().toURL());
```


Exercise 5: Run a Tokeniser

Implement the following:

- Load the plugin named "ANNIE";
- Instantiate a Language Analyser of type **`gate.creole.tokeniser.DefaultTokeniser`** (using the default values for all parameters);
- set the **`document`** of the tokeniser to the document created in exercise 2;
- set the **`corpus`** of the tokeniser to **`null`**;
- call the **`execute()`** method of the tokeniser;
- inspect the document and see what the results were.

Outline

GATE Controllers

- Controllers provide the implementation for execution control in GATE.
- They are called *applications* in GATE Developer.
- The implementations provided by default implement a *pipeline* architecture (they run a set of PRs one after another).
- Other kind of implementations are also possible.
  - e.g. the Groovy plugin provides a *scriptable* controller implementation (more details in module 8).
- A controller is a class that implements **`gate.Controller`**.

Implementation

gate.Controller

```
1 public interface Controller extends Resource,
2     Executable, NameBearer, FeatureBearer {
3   public Collection getPRs();
4   public void setPRs(Collection PRs);
5   public void execute() throws ExecutionException;
6 }
```

- all default controller implementations also implement **gate.ProcessingResource** (so you can include controllers inside other controllers!);
- like all GATE resources, controllers are created using the **Factory** class;
- controllers have names, and features.

Default Controller Types

The following default controller implementations are provided (all in the **gate.creole** package):

- **SerialController**: a pipeline of PRs.
- **ConditionalSerialController**: a pipeline of PRs. Each PR has an associated **RunningStrategy** value which can be used to decide **at runtime** whether or not to run the PR.
- **SerialAnalyserController**: a pipeline of **LanguageAnalyser**s, which runs all the PRs over all the documents in a **Corpus**. The **corpus** and **document** parameters for each PR are set by the controller.
- **RealtimeCorpusController**: a version of **SerialAnalyserController** that interrupts the execution over a document when a specified timeout has lapsed.

SerialAnalyserController API

SerialAnalyserController is the most used type of Controller. Its most important methods are:

```
1  // Adds a new PR at a given position
2  public void add(int index, ProcessingResource pr);
3  // Adds a new PR at the end
4  public void add(ProcessingResource pr);
5  // Replaces the PR at a given position
6  public ProcessingResource set(int index,
7      ProcessingResource pr);
8  // Remove a PRs by position
9  public ProcessingResource remove(int index);
10 // Remove a specified PR
11 public boolean remove(ProcessingResource pr);
12 // Sets the corpus to be processed
13 public void setCorpus(gate.Corpus corpus);
14 // Runs the controller
15 public void execute() throws ExecutionException;
```


Exercise 6: Run a Tokeniser (again!)

Implement the following:

- Create a SerialAnalyserController, and add the tokeniser from exercise 5 to it;
- Create a corpus, and add the document from exercise 2 to it;
- Set the **corpus** value of the controller to the newly created corpus;
- Execute the controller;
- Inspect the results.

Controller Persistency (or *Saving Applications*)

- The configuration of a controller (i.e. the list of PRs included, as well as the features and parameter values for the controller and its PRs) can be saved using a special type of XML serialisation.
- This is done using the **`gate.util.persistence.PersistenceManager`** class.
- This is what *GATE Developer* does when saving and loading applications.

Implementation

`gate.util.persistence.PersistenceManager`

```
1  // Serialises the configuration of a GATE object
2  // to a special XML format.
3  public static void saveObjectToFile(Object obj,
4      File file) throws PersistenceException,
5      IOException ;
6
7  // Re-creates the serialised GATE object from the saved
8  // configuration data.
9  public static Object loadObjectFromFile(File file)
10     throws PersistenceException, IOException,
11     ResourceInstantiationException;
12 // Loads a GATE object from a [remote] location.
13 public static Object loadObjectFromUrl(URL url)
14     throws PersistenceException, IOException,
15     ResourceInstantiationException;
```


Thank you!

Questions?

More answers at:

- `http://gate.ac.uk` (Our website)
- `http://gate.ac.uk/mail/` (Our mailing list)