Advanced GATE Embedded

Track II, Module 8

Fifth GATE Training Course
June 2012

Outline

1. GATE in Multi-threaded/Web Applications
   - Introduction
   - Multi-threading and GATE
   - Servlet Example
   - The Spring Framework
   - Making your own PRs duplication-friendly
2. GATE and Groovy
   - Introduction to Groovy
   - Scripting GATE Developer
   - Groovy Scripting for PRs and Controllers
   - Writing GATE Resource Classes in Groovy
3. Extending GATE
   - Adding new document formats

Outline

1 GATE in Multi-threaded/Web Applications
- Introduction
- Multi-threading and GATE
- Servlet Example
- The Spring Framework
- Making your own PRs duplication-friendly

2 GATE and Groovy
- Introduction to Groovy
- Scripting GATE Developer
- Groovy Scripting for PRs and Controllers
- Writing GATE Resource Classes in Groovy

3 Extending GATE
- Adding new document formats

Introduction

- Scenario:
  - Implementing a web application that uses GATE Embedded to process requests.
  - Want to support multiple concurrent requests
  - Long running process - need to be careful to avoid memory leaks, etc.
- Example used is a plain HttpServlet
  - Principles apply to other frameworks (struts, Spring MVC, Metro/CXF, Grails. . .)

Setting up

- GATE libraries in `WEB-INF/lib`
    - gate.jar + JARs from lib
- Usual GATE Embedded requirements:
    - A directory to be "gate.home"
    - Site and user config files
    - Plugins directory
- Alternatively use `Gate.runInSandbox`, but certain things can only be configured from the gate.xml files.

GATE in a Multi-threaded Environment

- GATE initialization needs to happen once (and only once) before any other GATE APIs are used.
- The `Factory` is synchronized internally, so safe for use in multiple threads.
- Individual PRs/controllers are *not* safe – must not use the same PR instance concurrently in different threads
  - this is due to the design of runtime parameters as Java Beans properties.
- Individual LRs (documents, ontologies, etc.) are only thread-safe when accessed read-only by *all* threads.
  - if you need to share an LR between threads, be sure to synchronize (e.g. using `ReentrantReadWriteLock`)

Initializing GATE using a ServletContextListener

`ServletContextListener` called by container at startup and shutdown (only startup method shown).

```java
public void contextInitialized(ServletContextEvent e) {
  ServletContext ctx = e.getServletContext();
  File gateHome = new File(
      ctx.getRealPath("/WEB-INF"));
  Gate.setGateHome(gateHome);
  File userConfig = new File(
      ctx.getRealPath("/WEB-INF/user.xml"));
  Gate.setUserConfigFile(userConfig);
  // default site config is gateHome/gate.xml
  // default plugins dir is gateHome/plugins
  Gate.init();
}
```

Initializing GATE using a ServletContextListener

You must register the listener in `web.xml`

```
<listener>
  <listener-class>
    gate.web.example.GateInitListener
  </listener-class>
</listener>
```

Handling Concurrent Requests

Naïve approach – new PRs for every request

```
public void doPost(request, response) {
  ProcessingResource pr = Factory.createResource(...);
  try {
    Document doc = Factory.newDocument(
        getTextFromRequest(request));
    try {
      // do some stuff
    }
    finally {
      Factory.deleteResource(doc);
    }
  }
  finally {
    Factory.deleteResource(pr);
  }
}
```

Handling Concurrent Requests

Naïve approach – new PRs for every request

```
public void doPost(request, response) {
  ProcessingResource pr = Factory.createResource(...);
  try {
    Document doc = Factory.newDocument(
        getTextFromRequest(request));
    try {
      // do some stuff
    }
    finally {
      Factory.deleteResource(doc);
    }
  }
  finally {
    Factory.deleteResource(pr);
  }
}
```

Many levels of try/finally – make sure you clean up even when errors occur

Problems with Naïve Approach

- Guarantees no interference between threads
- But inefficient, particularly with complex PRs (large gazetteers, etc.)
- Hidden problem with JAPE:
  - Parsing a JAPE grammar creates and compiles Java classes
  - Once created, classes are never unloaded
  - Even with simple grammars, eventually OutOfMemoryError (PermGen space)

Take Two: using ThreadLocal

Store the PR/Controller in a thread-local variable

```java
private ThreadLocal<CorpusController> controller =
     new ThreadLocal<CorpusController>() {

  protected CorpusController initialValue() {
    return loadController();
  }
};

private CorpusController loadController() { ... }

public void doPost(request, response) {
  CorpusController c = controller.get();
  // do stuff with the controller
}
```

An Improvement. . .

- Only initialise resources once per thread
- Interacts nicely with typical web server thread pooling
- But if a thread dies (e.g. with an exception), no way to clean up its controller

One Solution: Object Pooling

- Manage your own pool of Controller instances
- Take a controller from the pool at the start of a request, return it (in a finally!) at the end
- Number of instances in the pool determines maximum concurrency level

Simple Example of Pooling

Setting up and cleaning up:

```java
private BlockingQueue<CorpusController> pool;

public void init() {
  pool = new LinkedBlockingQueue<CorpusController>();
  for(int i = 0; i < POOL_SIZE; i++) {
    pool.add(loadController());
  }
}

public void destroy() {
  for(CorpusController c : pool) {
    Factory.deleteResource(c);
  }
}
```

Simple Example of Pooling

Processing requests:

```
public void doPost(request, response) {
  CorpusController c = pool.take();
  try {
    // do stuff
  }
  finally {
    pool.add(c);
  }
}
```

Simple Example of Pooling

Processing requests:

```java
public void doPost(request, response) {
  CorpusController c = pool.take();
  try {
    // do stuff
  }
  finally {
    pool.add(c);
  }
}
```

This blocks when the pool is empty. Use `poll` for non-blocking check.

Creating the pool

- Typically to create the pool you would use `PersistenceManager` to load a saved application several times.
- But this is not always optimal, e.g. large gazetteers consume lots of memory.
- GATE provides API to *duplicate* an existing instance of a resource: `Factory.duplicate(existingResource)`.
- By default, this simply calls `Factory.createResource` with the same class name, parameters, features and name.
- But individual Resource classes can override this by implementing the `CustomDuplication` interface (more later).
    - e.g. `DefaultGazetteer` uses a `SharedDefaultGazetteer` — same behaviour, but shares the in-memory representation of the lists.

Other Caveats

- With most PRs it is safe to create lots of identical instances
- But *not all*!
  - e.g. training a machine learning model with the batch learning PR (in the `Learning` plugin)
  - but it is safe to have several instances *applying* an existing model.
- When using `Factory.duplicate`, be careful not to duplicate a PR that is being used by another thread
  - i.e. either create all your duplicates up-front or else keep the original prototype “pristine”.

Exporting the Grunt Work: Spring

- http://www.springsource.org/
- “Inversion of Control”
- Configure your business objects and connections between them using XML or Java annotations
- Handles application startup and shutdown
- GATE provides helpers to initialise GATE, load saved applications, etc.
- Built-in support for object pooling
- Web application framework (Spring MVC)
- Used by other frameworks (Grails, CXF, . . .)

Using Spring in Web Applications

- Spring provides a `ServletContextListener` to create a single *application context* at startup.
- Takes configuration by default from `WEB-INF/applicationContext.xml`
- Context made available through the `ServletContext`
- For our running example we use Spring's `HttpRequestHandler` interface which abstracts from servlet API
- Configure an `HttpRequestHandler` implementation as a Spring bean, make it available as a servlet.
    - allows us to configure dependencies and pooling using Spring

Initializing GATE via Spring

applicationContext.xml:

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:gate="http://gate.ac.uk/ns/spring">
  <gate:init gate-home="/WEB-INF"
             plugins-home="/WEB-INF/plugins"
             site-config-file="/WEB-INF/gate.xml"
             user-config-file="/WEB-INF/user-gate.xml">
    <gate:preload-plugins>
      <value>/WEB-INF/plugins/ANNIE</value>
    </gate:preload-plugins>
  </gate:init>
</beans>
```

Loading a Saved Application

To load an application state saved from GATE Developer:

```
<gate:saved-application
        id="myApp"
        location="/WEB-INF/application.xgapp"
        scope="prototype" />
```

- `scope="prototype"` means create a new instance each time we ask for it
- Default scope is “singleton” — one instance is created at startup and shared.

Duplicating an Application

- Alternatively, load the application once and then duplicate it

```
<gate:duplicate id="myApp" return-template="true">
  <gate:saved-application location="..." />
</gate:duplicate>
```

- `<gate:duplicate>` creates a new duplicate each time we ask for the bean.
- `return-template` means the original controller (from the `saved-application`) will be returned the first time, then duplicates thereafter.
- Without this the original is kept pristine and only used as a source for duplicates.

Spring Servlet Example

Write the `HttpRequestHandler` assuming single-threaded access, we will let Spring deal with the pooling for us.

```
public class MyHandler
        implements HttpRequestHandler {
  // controller reference will be injected by Spring
  public void setApplication(
            CorpusController app) { ... }

  // good manners to clean it up ourselves though this isn’t
  // necessary when using <gate:duplicate>
  public void destroy() throws Exception {
    Factory.deleteResource(app);
  }
```

Spring Servlet Example

```
  public void handleRequest(request, response) {
    Document doc = Factory.newDocument(
        getTextFromRequest(request));
    try {
      // do some stuff with the app
    }
    finally {
      Factory.deleteResource(doc);
    }
  }
}
```

Tying it together

In `applicationContext.xml`

```
<gate:init ... />
<gate:duplicate id="myApp" return-template="true">
  <gate:saved-application
       location="/WEB-INF/application.xgapp" />
</gate:duplicate>

<!-- Define the handler bean, inject the controller -->
<bean id="mainHandler"
      class="my.pkg.MyHandler"
      destroy-method="destroy">
  <property name="application" ref="myApp" />
  <gate:pooled-proxy max-size="3"
                     initial-size="3" />
</bean>
```

Tying it together: Spring Pooling

```
  <gate:pooled-proxy max-size="3"
                     initial-size="3" />
```

- A *bean definition decorator* that tells Spring that instead of a singleton `mainHandler` bean, we want
  - a pool of 3 instances of `MyHandler`
  - exposed as a single *proxy* object implementing the same interfaces
- *Each method call* on the proxy is dispatched to one of the objects in the pool.
- Each target bean is guaranteed to be accessed by no more than one thread at a time.
- When the pool is empty (i.e. more than 3 concurrent requests) further requests will block.

Tying it together: Spring Pooling

- Many more options to control the pool, e.g. for a pool that grows as required and shuts down instances that have been idle for too long, and where excess requests fail rather than blocking:

```
<gate:pooled-proxy
    max-size="10"
    max-idle="3"
    time-between-eviction-runs-millis="180000"
    min-evictable-idle-time-millis="90000"
    when-exhausted-action-name="WHEN_EXHAUSTED_FAIL"
/>
```

- Under the covers, `<gate:pooled-proxy>` creates a Spring `CommonsPoolTargetSource`, attributes correspond to properties of this class.
- See the Spring documentation for full details.

Tying it together: web.xml

To set up the Spring context:

```
<listener>
  <listener-class>
    org.springframework.web.context.
        ContextLoaderListener
  </listener-class>
</listener>
```

Tying it together: web.xml

To make the HttpRequestHandler available as a servlet, create a servlet entry in web.xml with the same name as the (pooled) handler bean:

```
<servlet>
  <servlet-name>mainHandler</servlet-name>
  <servlet-class>
    org.springframework.web.context.support.
        HttpRequestHandlerServlet
  </servlet-class>
</servlet>
```

Exercise 1: A simple web application

- In `hands-on/webapps` you have an implementation of the `HttpRequestHandler` example.
- `hands-on/webapps/gate` is a simple web application which provides
  - an HTML form where you can enter text to be processed by GATE
  - an `HttpRequestHandler` that processes the form submission using a GATE application and displays the document's features in an HTML table
  - the application and pooling of the handlers is configured using Spring.
- Embedded Jetty server to run the app.
- To keep the download small, most of the required JARs are not in the module-8.zip file – you already have them in GATE.

Exercise 1: A simple web application

- To run the example you need ant.
- Edit `webapps/gate/WEB-INF/build.xml` and set the `gate.home` property correctly.
- In `webapps/gate/WEB-INF`, run `ant`.
  - this copies the remaining dependencies from GATE and compiles the `HttpRequestHandler` Java code from `WEB-INF/src`.
- `WEB-INF/gate-files` contains the site and user configuration files.
- This is also where the webapp expects to find the `.xgapp`.
- No `.xgapp` provided by default – you need to provide one.

Exercise 1: A simple web application

- Use the statistics application you wrote yesterday.
- In GATE Developer, create a "corpus pipeline" application containing a tokeniser and your statistics PR.
- Right-click on the application and "Export for GATECloud.net".
    - This will save the application state along with all the plugins it depends on in a single zip file.
- Unpack the zip file under `WEB-INF/gate-files`
    - don't create any extra directories – you need `application.xgapp` to end up in `gate-files`.

Exercise 1: A simple web application

- You can now run the server – in `hands-on/webapps` run `ant -emacs`
- Browse to `http://localhost:8080/gate/`, enter some text and submit
- Watch the log messages. . .
- Notice the result page includes "GATE handler *N*" – each handler in the pool has a unique ID.
- Multiple submissions go to different handler instances in the pool.
- `http://localhost:8080/stop` to shut down the server gracefully
- Try editing `gate/WEB-INF/applicationContext.xml` and change the pooling configuration.
- Try opening several browser windows and using a longer "delay" to test concurrent requests.

Not Just for Webapps

- Spring isn't just for web applications
- You can use the same tricks in other embedded apps
- GATE provides a `DocumentProcessor` interface suitable for use with Spring pooling

```
// load an application context from definitions in a file
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("beans.xml");

DocumentProcessor proc = ctx.getBean(
    "documentProcessor", DocumentProcessor.class);

// in worker threads…
proc.processDocument(myDocument);
```

Not Just for Webapps

The `beans.xml` file:

```
<gate:init ... />
<gate:duplicate id="myApp">
  <gate:saved-application
         location="resources/application.xgapp" />
</gate:duplicate>

<!-- Define the processor bean to be pooled -->
<bean id="documentProcessor"
      class="gate.util.
         LanguageAnalyserDocumentProcessor"
      destroy-method="cleanup">
  <property name="analyser" ref="myApp" />
  <gate:pooled-proxy max-size="3" />
</bean>
```

Conclusions

Two golden rules:

- Only use a GATE Resource in one thread at a time
- Always clean up after yourself, even if things go wrong (`deleteResource` in a finally block).

Duplication and Custom PRs

- Recap: by default, `Factory.duplicate` calls `createResource` passing the same type, parameters, features and name
- This can be sub-optimal for resources that rely on large read-only data structures that could be shared
- If this applies to your custom PR you can take steps to make it handle duplication more intelligently
- For simple cases: *sharable properties*, for complex cases: *custom duplication*.

Sharable properties

- A way to share object references between a PR and its duplicates
- A JavaBean setter/getter pair with the setter annotated (same as for `@CreoleParameter`)

```java
private Map dataTable;

public Map getDataTable() { return dataTable; }

@Sharable
public void setDataTable(Map m) {
  dataTable = m;
}
```

Sharable properties

- Default duplication algorithm will get property value from original and set it on the duplicate before calling `init()`
- `init()` must detect when sharable properties have been set and react appropriately.

```
public Resource init() throws /* ... */ {
  if(dataTable == null) {
    // only need to build the data table if we weren't given a shared one
    buildDataTable();
  }
}

public void reInit() throws /* ... */ {
  // clear sharables on reInit
  dataTable = null;
  super.reInit();
}
```

Sharable properties – Caveats

- Anything shared between PRs *must* be thread-safe
  - use appropriate synchronization if any of the threads modifies the shared object (e.g. a `ReentrantReadWriteLock` which is itself `@Sharable`).
  - or (for the dataTable example), use an inherently safe class such as `ConcurrentHashMap`
  - for shared counter, use `AtomicInteger`
- If you use sharable properties, take care not to break `reInit`

Exercise 2: Multi-threaded cumulative statistics

- `hands-on/shared-stats` contains a variation on yesterday's `DocStats` PR that keeps a running total of the number of Tokens it has seen.
- Build this (using the Ant build file), load the plugin, create an application containing a tokeniser and a "Shared document statistics" PR, export for GATECloud.net and unzip into your webapp as before.
- Try posting some requests to the webapp.
- You will see a `running_total` feature, but this is per handler, not global across handlers.

Exercise 2: Multi-threaded cumulative statistics

- Your task: make the running total global.
- Make the `totalCount` field into a sharable property
  - it's already a thread-safe `AtomicInteger`
  - add a getter and setter, with the right annotation
  - `init()` logic to handle the shared/non-shared cases
  - implement a sensible `reInit()`
- You will need to re-build your PR and re-export (or just copy the compiled plugin to the right place in your webapp).

Exercise 2: Solution

Getter and setter:

```java
private AtomicInteger totalCount;

public AtomicInteger getTotalCount() {
  return totalCount;
}

@Sharable
public void setTotalCount(AtomicInteger tc) {
  this.totalCount = tc;
}
```

Exercise 2: Solution

`init()` and `reInit()`:

```
public Resource init() throws
        ResourceInstantiationException {
  if(totalCount == null) {
    totalCount = new AtomicInteger(0);
  }
  return this;
}

public void reInit() throws
        ResourceInstantiationException {
  totalCount = null;
  super.reInit();
}
```

`execute()` is unchanged.

Custom Duplication

- For more complex cases, a resource can take complete control of its own duplication by implementing `CustomDuplication`
- This tells `Factory.duplicate` to call the resource's own `duplicate` method instead of the default algorithm.

```
public Resource duplicate(DuplicationContext ctx)
    throws ResourceInstantiationException;
```

- `duplicate` should create and return a duplicate, which need not be the same concrete class but must "behave the same"
  - Defined in terms of implemented interfaces.
  - Exact specification can be found in the `Factory.duplicate` JavaDoc.

Custom Duplication

- If you need to duplicate other resources, use the two-argument `Factory.duplicate`, passing the `ctx` as the second parameter, to preserve object graph
  - two calls to `Factory.duplicate(r, ctx)` for the same resource `r` in the same context `ctx` will return the same duplicate.
  - calls to the single argument `Factory.duplicate(r)` or to the two-argument version with different contexts will return different duplicates.
- Can call the default duplicate algorithm (bypassing the `CustomDuplication` check) via `Factory.defaultDuplicate`
  - it is safe to call `defaultDuplicate(this, ctx)`, but calling `duplicate(this, ctx)` from within its own custom `duplicate` will cause infinite recursion!

Custom Duplication Example (SerialController)

```
public Resource duplicate(DuplicationContext ctx)
        throws ResourceInstantiationException {
  // duplicate this controller in the default way - this handles subclasses nicely
  Controller c = (Controller)Factory.defaultDuplicate(
                 this, ctx);

  // duplicate each of our PRs
  List<ProcessingResource> newPRs =
        new ArrayList<ProcessingResource>();
  for(ProcessingResource pr : prList) {
    newPRs.add((ProcessingResource)Factory.duplicate(
          pr, ctx));
  }
  // and set this duplicated list as the PRs of the copy
  c.setPRs(newPRs);

  return c;
}
```

Outline

1 GATE in Multi-threaded/Web Applications
- Introduction
- Multi-threading and GATE
- Servlet Example
- The Spring Framework
- Making your own PRs duplication-friendly

2 GATE and Groovy
- Introduction to Groovy
- Scripting GATE Developer
- Groovy Scripting for PRs and Controllers
- Writing GATE Resource Classes in Groovy

3 Extending GATE
- Adding new document formats

Groovy

- Dynamic language for the JVM
- Groovy scripts and classes compile to Java bytecode – fully interoperable with Java.
- Syntax very close to regular Java
- Explicit types optional, semicolons optional
- Dynamic dispatch – method calls dispatched based on runtime type rather than compile-time.
- Can add new methods to existing classes at runtime using *metaclass* mechanism
- Groovy adds useful extra methods to many standard classes in `java.io`, `java.lang`, etc.

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get('a').findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get('a').findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `def` keyword declares an untyped variable

Groovy example

Find the start offset of each absolute link in the document.

```groovy
def om = document.getAnnotations("Original markups")
om.get(’a’).findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `def` keyword declares an untyped variable
- but dynamic dispatch ensures the `get` call goes to the right class (`AnnotationSet`).

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get('a').findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `def` keyword declares an untyped variable
- but dynamic dispatch ensures the `get` call goes to the right class (`AnnotationSet`).
- `findAll` and `collect` are methods added to `Collection` by Groovy

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get(’a’).findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- **`def`** keyword declares an untyped variable
- but dynamic dispatch ensures the `get` call goes to the right class (`AnnotationSet`).
- **`findAll`** and **`collect`** are methods added to `Collection` by Groovy
  - `http://groovy.codehaus.org/groovy-jdk` has the details.

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get('a').findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `def` keyword declares an untyped variable
- but dynamic dispatch ensures the `get` call goes to the right class (`AnnotationSet`).
- **`findAll`** and **`collect`** are methods added to `Collection` by Groovy
    - `http://groovy.codehaus.org/groovy-jdk` has the details.
- `?.` is the *safe navigation* operator – if the left hand operand is **`null`** it returns **`null`** rather than throwing an exception

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get(’a’).findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get(’a’).findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `=~` for regular expression matching

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get(’a’).findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `=~` for regular expression matching
- unified access to JavaBean properties – `it.startNode` shorthand for `it.getStartNode()`

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get(’a’).findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `=~` for regular expression matching
- unified access to JavaBean properties – `it.startNode` shorthand for `it.getStartNode()`
- and `Map` entries – `anchor.features.href` shorthand for `anchor.getFeatures().get("href")`

Groovy example

Find the start offset of each absolute link in the document.

```
def om = document.getAnnotations("Original markups")
om.get('a').findAll { anchor ->
  anchor.features?.href =~ /^http:/
}.collect { it.startNode.offset }
```

- `=~` for regular expression matching
- unified access to JavaBean properties – `it.startNode` shorthand for `it.getStartNode()`
- and `Map` entries – `anchor.features.href` shorthand for `anchor.getFeatures().get("href")`
- `Map` entries can also be accessed like arrays, e.g. `features["href"]`

Closures

Parameter to `collect`, `findAll`, etc. is a *closure*

- like an anonymous function (JavaScript), a block of code that can be assigned to a variable and called repeatedly.
- Can declare parameters (typed or untyped) between the opening brace and the `->`
- If no explicit parameters, closure has an implicit parameter called `it`.
- Closures have access to the variables in their containing scope (unlike Java inner classes these do not have to be `final`).
- The return value of a closure is the value of its last expression (or an explicit `return`).
- Closures are used all over the place in Groovy

More Groovy Syntax

- Shorthand for lists: `["item1", "item2"]` declares an `ArrayList`
- Shorthand for maps: `[foo:"bar"]` creates a `HashMap` mapping the key `"foo"` to the value `"bar"`.
- Interpolation in *double-quoted* strings (like Perl): `"There are ${anns.size()} annotations of type ${annType}"`
- Parentheses for method calls are optional (where this is unambiguous): `myList.add 0, "someString"`
    - When you use parentheses, if the last parameter is a closure it can go outside them: this is a method call with two parameters `someList.inject(0) { last, cur -> last + cur }`
- "slashy string" syntax where backslashes don't need to be doubled: `/C:\Program Files\Gate/` equivalent to `'C:\\Program Files\\Gate'`

Operator Overloading

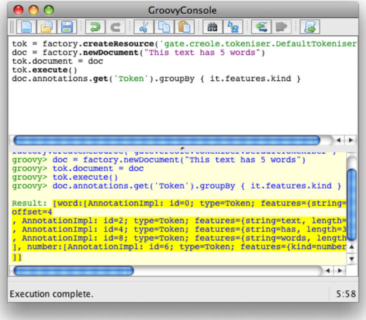
- Groovy supports operator overloading cleanly
- Every operator translates to a method call
  - `x == y` becomes `x.equals(y)` (for reference equality, use `x.is(y)`)
  - `x + y` becomes `x.plus(y)`
  - `x << y` becomes `x.leftShift(y)`
  - full list at `http://groovy.codehaus.org`
- To overload an operator for your own class, just implement the method.
- e.g. List implements `leftShift` to append items to the list:
  `['a', 'b'] << 'c'== ['a', 'b', 'c']`

Groovy in GATE

- Groovy support in GATE is provided by the `Groovy` plugin.
- Loading the plugin
  - enables the Groovy scripting console in GATE Developer
  - adds utility methods to various GATE classes and interfaces for use from Groovy code
  - provides a PR to run a Groovy script.
  - provides a *scriptable controller* whose execution strategy is determined by a Groovy script.

Scripting GATE Developer

- Groovy provides a Swing-based *console* to test out small snippets of code.
- The console is available in the GATE Developer GUI via the Tools menu. To enable, load the `Groovy` plugin.



```
tok = factory.createResource('gate.creole.tokeniser.DefaultTokeniser
doc = factory.newDocument("This text has 5 words")
tok.document = doc
tok.execute()
doc.annotations.get('Token').groupBy { it.features.kind }
```

```
groovy> doc = factory.newDocument("This text has 5 words")
groovy> tok.document = doc
groovy> tok.execute()
groovy> doc.annotations.get('Token').groupBy { it.features.kind }

Result: [word:[AnnotationImpl: id=0; type=Token; features={string=
offset=4
, AnnotationImpl: id=2; type=Token; features={string=text, length=
, AnnotationImpl: id=4; type=Token; features={string=has, length=3
, AnnotationImpl: id=8; type=Token; features={string=words, length
], number:[AnnotationImpl: id=6; type=Token; features={kind=number
]]
```

Execution complete.

Imports and Predefined Variables

The GATE Groovy console imports the same packages as JAPE RHS actions:

- `gate`, `gate.annotation`, `gate.util`, `gate.jape` and `gate.creole.ontology`

The following variables are implicitly defined:

corpora a list of loaded corpora LRs (`Corpus`)

docs a list of all loaded document LRs (`DocumentImpl`)

prs a list of all loaded PRs

apps a list of all loaded Applications (`AbstractController`)

Exercise 1: The Groovy Console

- Start the GATE Developer GUI
- Load the `Groovy` plugin
- Select Tools → Groovy Tools → Groovy Console
- Experiment with the console
- For example to tokenise a document and find how many “number” tokens it contains:

```
doc = Factory.newDocument(new URL(’http://gate.ac.uk’))
tokeniser = Factory.createResource(’gate.creole.tokeniser.
    DefaultTokeniser’)
tokeniser.document = doc
tokeniser.execute()
tokens = doc.annotations.get(’Token’)
tokens.findAll { it.features.kind == ’number’ }.size()
```

Exercise 1: The Groovy Console

- Variables you assign in the console (without a **`def`** or a type declaration) remain available to future scripts in the same console.
- So you can run the previous example, then try more things with the `doc` and `tokens` variables.
- Some things to try:
  - Find the names and sizes of all the annotation sets on the document (there will probably only be one named set).
  - List all the different `kind`s of token
  - Find the longest word in the document

Exercise 1: Solution

Some possible solutions (there are many. . .)

```
// Find the annotation set names and sizes
doc.namedAnnotationSets.each { name, set ->
  println "${name} has size ${set.size()}"
}

// List the different kinds of token
tokens.collect { it.features.kind }.unique()

// Find the longest word
tokens.findAll {
  it.features.kind == ’word’
}.max { it.features.length.toInteger() }
```

Groovy Categories

- In Groovy, a class declaring static methods can be used as a *category* to inject methods into existing types (including interfaces)
- A static method in the category class whose first parameter is a `Document`:

  ```
  public static SomeType foo(Document d, String arg)
  ```
- . . . becomes an instance method of the `Document` class:

  ```
  public SomeType foo(String arg)
  ```
- The `use` keyword activates a category for a single block
- To enable the category globally:

  ```
  TargetClass.mixin(CategoryClass)
  ```

Utility Methods

- The `gate.Utils` class (mentioned in the JAPE module) contains utility methods for documents, annotations, etc.
- Loading the `Groovy` plugin treats this class as a category and installs it as a global mixin.
- Enables syntax like:

```
tokens.findAll {
  it.features.kind == 'number'
}.each {
  println "${it.type}: length = ${it.length()}, "
  println "  string = ${doc.stringFor(it)}"
}
```

Utility Methods

- The `Groovy` plugin also mixes in the `GateGroovyMethods` class.
- This extends common Groovy idioms to GATE classes
    - e.g. implements **`each`**, **`eachWithIndex`** and **`collect`** for `Corpus` to do the right thing when the corpus is stored in a datastore
    - defines a `withResource` method on `Resource`, to call a closure with a given resource as a parameter, and ensure the resource is deleted when the closure returns:

```
Factory.newDocument(someURL).withResource { doc ->
  // do something with the document
}
```

Utility Methods

- Also overloads the subscript operator `[]` to allow:
  - `annSet["Token"]` and `annSet["Person", "Location"]`
  - `annSet[15..20]` to get annotations within given span
  - `doc.content[15..20]` to get the `DocumentContent` within a given span
- See `src/gate/groovy/GateGroovyMethods.java` in the `Groovy` plugin for details.

Exercise 2: Using a category

In the console, try using some of these new methods:

```
tokens = doc.annotations["Token"]
tokens.findAll {
  it.features.kind == ’number’
}.each {
  println "${it.type}: length = ${it.length()}, "
  println "  string = ${doc.stringFor(it)}"
}
```

The Groovy Script PR

- The `Groovy` plugin provides a PR to execute a Groovy script.
- Useful for quick prototyping, or tasks that can't be done by JAPE but don't warrant writing a custom PR.
- PR takes the following parameters:

scriptURL (init-time) The path to a valid Groovy script

inputASName an optional annotation set intended to be used as input by the PR

outputASName an optional annotation set intended to be used as output by the PR

scriptParams optional parameters for the script as a `FeatureMap`

Script Variables

The script has the following implicit variables available when it is run

doc the current document

corpus the corpus containing the current document

content the string content of the current document

inputAS the annotation set specified by inputASName in the PRs runtime parameters

outputAS the annotation set specified by outputASName in the PRs runtime parameters

scriptParams the parameters `FeatureMap` passed as a runtime parameter

and the same implicit imports as the console.

Corpus-level processing

- Any other variables are treated like instance variables in a PR – values set while processing one document are available while processing the next.
- So Groovy script is stateful, can e.g. collect statistics from all the documents in a corpus.
- Script can declare methods for pre- and post-processing:
  - `beforeCorpus` called before first document is processed.
  - `afterCorpus` called after last document is processed
  - `aborted` called if anything goes wrong
- All three take the corpus as a parameter
- `scriptParams` available within methods, other variables not.

Controller Callbacks Example

Count the number of annotations of a particular type across the corpus

```
void beforeCorpus(c) {
  println "Processing corpus ${c.name}"
  count = 0
}

count += doc.annotations[scriptParams.type].size()

void afterCorpus(c) {
  println "Total ${scriptParams.type} annotations " +
      "in corpus ${c.name}: ${count}"
}
```

Exercise 3: Using the Script PR

- Write a very simple Goldfish annotator as a Groovy script
  - Annotate all occurrences of the word “goldfish” (case-insensitive) in the input document as the annotation type “Goldfish”.
  - Add a “numFish” feature to each Sentence annotation giving the number of Goldfish annotations that the sentence contains.
- Put your script in the file `hands-on/groovy/goldfish.groovy`
- To test, load `hands-on/groovy/goldfish-app.xgapp` into GATE Developer (this application contains tokeniser, sentence splitter and goldfish script PR).
- You need to re-initialize the Groovy Script PR after each edit to `goldfish.groovy`

Exercise 3: Solution

One of many possible solutions:

```
def m = (content =~ /(?i)goldfish/)
while(m.find()) {
  outputAS.add((long)m.start(), (long)m.end(),
      ’Goldfish’, [:].toFeatureMap())
}

def allGoldfish = outputAS["Goldfish"]
inputAS["Sentence"].each { sent ->
  sent.features.numFish =
      allGoldfish[sent.start()..sent.end()].size()
}
```

The Scriptable Controller

- `ConditionalSerialAnalyserController` can run PRs conditionally based on the value of a document feature.
- This is useful but limited; Groovy plugin's scriptable controller provides more flexibility.
- Uses Groovy DSL to define the execution strategy.

The ScriptableController DSL

- Run a single PR by using its *name* as a method call
  - So good idea to give your PRs identifier-friendly names.
- Iterate over the documents in the corpus using `eachDocument`
- Within an `eachDocument` closure, any PRs that implement `LanguageAnalyser` get their `document` and `corpus` parameters set appropriately.
- Override runtime parameters by passing named arguments to the PR method call.
- DSL is a Groovy script, so all Groovy language features available (conditionals, loops, method declarations, local variables, etc.).

`http://gate.ac.uk/userguide/sec:api:groovy:controller`

ScriptableController example

```
eachDocument {
  documentReset()
  tokeniser()
  gazetteer()
  splitter()
  posTagger()
  findLocations()
  // choose the appropriate classifier depending how many Locations were found
  if(doc.annotations["Location"].size() > 100) {
    fastLocationClassifier()
  }
  else {
    fullLocationClassifier()
  }
}
```

ScriptableController example

```
eachDocument {
  // find all the annotatorN sets on this document
  def annotators =
    doc.annotationSetNames.findAll {
      it ==~ /annotator\d+/
    }

  // run the post-processing JAPE grammar on each one
  annotators.each { asName ->
    postProcessingGrammar(
        inputASName: asName,
        outputASName: asName)
  }

  // merge them to form a consensus set
  mergingPR(annSetsForMerging: annotators.join(';'))
}
```

Robustness and Realtime Features

- When processing large corpora, applications need to be robust.
  - If processing of a single document fails it should not abort processing of the whole corpus.
- When processing mixed corpora or using complex grammars, most documents process quickly but a few may take much longer.
  - Option to interrupt/terminate processing of a document when it takes too long.
  - Particularly useful with pay-per-hour processing such as GATECloud.net

Ignoring Errors

- Use an `ignoringErrors` block to ignore any exceptions thrown in the block.

```
eachDocument {
  ignoringErrors {
    myTransducer()
  }
}
```

- Exceptions thrown will be logged but will not terminate execution.
- Note nesting
  - `ignoringErrors` inside `eachDocument` – exception means move to next document.
  - `eachDocument` inside `ignoringErrors` – exception would terminate processing of corpus.

Limiting Execution Time

- Use a `timeLimit` block to place a limit on the running time of the given block.

```
eachDocument {
  annotateLocations()
  timeLimit(soft:30.seconds, hard:30.seconds) {
    classifyLocations()
  }
}
```

- *soft* limit – `interrupt` the running thread and PR
- *hard* limit – `Thread.stop()`
- Limits are cumulative – hard limit starts counting from the expiry of the soft limit.

Limiting Execution Time (2)

- When a block is terminated due to reaching a hard time limit, this generates an exception.
    - So in GATE Developer you probably want to wrap the `timeLimit` block in an `ignoringErrors` so it doesn't fail the corpus.
    - But on GATECloud.net each document is processed separately, so you *do* want the exception thrown to mark the offending document as failed.
- Treat `timeLimit` as a last resort – use heuristics to try and avoid long-running PRs (see the "fast" vs. "full" location classifier example).

Writing Resources in Groovy

- Groovy is more than a scripting language – you can write classes (including GATE resources such as `ScriptableController`) in Groovy and compile them to Java bytecode.
- Compiler available via `<groovyc>` Ant task in `groovy-all` JAR.
- In order to use GATE resources written in Groovy (other than those that are part of the Groovy plugin), `groovy-all` JAR file must go into `gate/lib`.

Groovy Beans

- Recall unified Java Bean property access in Groovy
  - `x = it.someProp` means `x = it.getSomeProp()`
  - `it.someProp = x` means `it.setSomeProp(x)`
- Declarations have a similar shorthand: a field declaration with no **`public`**, **`protected`** or **`private`** modifier becomes a private field plus an auto-generated public getter/setter pair.
- But you can provide explicit setter or getter, which will be used instead of the automatic one.
  - Need to do this if you need to annotate the setter (e.g. as a `CreoleParameter`).
  - Declare the setter **`private`** to get a read-only property (but not if it's a creole parameter).

Example: a Groovy Regex PR

```groovy
package gate.groovy.example

import gate.*
import gate.creole.*

public class RegexPR extends AbstractLanguageAnalyser {
  String regex
  String annType
  String annotationSetName

  public void execute() {
    def aSet = document.getAnnotations(annotationSetName)
    def matcher = (document.content.toString() =~ regex)
    while(matcher.find()) {
      aSet.add(matcher.start(), matcher.end(),
          annType, [:].toFeatureMap())
    }
  }
}
```

Outline

1. GATE in Multi-threaded/Web Applications
   - Introduction
   - Multi-threading and GATE
   - Servlet Example
   - The Spring Framework
   - Making your own PRs duplication-friendly
2. GATE and Groovy
   - Introduction to Groovy
   - Scripting GATE Developer
   - Groovy Scripting for PRs and Controllers
   - Writing GATE Resource Classes in Groovy
3. Extending GATE
   - Adding new document formats

Adding new document formats

- GATE provides default support for reading many source document formats, including plain text, HTML, XML, PDF, DOC, ...
- The mechanism is extensible – the format parsers are themselves resources, which can be provided via CREOLE plugins.
- GATE chooses the format to use for a document based on *MIME type*, deduced from
  - explicit `mimeType` parameter
  - file extension (for documents loaded from a URL)
  - web server supplied Content-Type (for documents loaded from an `http:` URL)
  - “magic numbers”, i.e. signature content at or near the beginning of the document

The DocumentFormat resource type

- A GATE document format parser is a resource that extends the `DocumentFormat` abstract class or one of its subclasses.
- Override `unpackMarkup` method to do the actual format parsing, creating annotations in the `Original markups` annotation set and optionally modifying the document content.
- Override `init` to register with the format detection mechanism.
- In theory, can take parameters like any other resource . . .
- . . . but in practice most formats are singletons, created as *autoinstances* when their defining plugin is loaded.

Repositioning info

- Some formats are able to record *repositioning info*
- Associates the offsets in the extracted text with their corresponding offsets in the original content.
- Allows you to save annotations as markup inserted into the original content.
- Of the default formats, only HTML can do this reliably.
    - If you're interested, see the `NekoHtmlDocumentFormat`

Implementing a `DocumentFormat`

- Define a class that extends `DocumentFormat`, with CREOLE metadata

```java
import gate.*;
import gate.creole.metadata.*;
import gate.corpora.*;

@CreoleResource(name = "Example DocumentFormat",
    autoinstances = {@AutoInstance})
public class MyDocumentFormat
         extends TextualDocumentFormat {
  // ...
}
```

- `autoinstances` causes GATE to create an instance of this resource automatically when the plugin is loaded.

DocumentFormat methods

- Most formats need to override three or four methods.
- supportsRepositioning to specify whether or not the format is capable of collecting repositioning info – most aren't

```
public Boolean supportsRepositioning() {
  return false;
}
```

`DocumentFormat` methods

- Two variants of `unpackMarkup`
- If you don't support repositioning then best to extend `TextualDocumentFormat` and just override the simple one:

```java
public void unpackMarkup(Document doc)
      throws DocumentFormatException {
  AnnotationSet om = doc.getAnnotations(
      GateConstants.ORIGINAL_MARKUPS_ANNOT_SET_NAME);
  // Make changes to the document content, add annotations to om
}
```

- Other variant (for repositioning formats) is implemented in terms of this one by `TextualDocumentFormat`

`DocumentFormat` methods

- Finally, `init` to register the format with GATE
- Mostly boilerplate, using protected `Map` fields defined in `DocumentFormat`

```java
public Resource init() throws
    ResourceInstantiationException {
  MimeType mime = new MimeType("text", "x-special");
  mimeString2ClassHandlerMap.put(
      mime.getType()+ "/" + mime.getSubtype(), this);
  mimeString2mimeTypeMap.put(
      mime.getType() + "/" + mime.getSubtype(), mime);
  suffixes2mimeTypeMap.put("spec", mime);
  magic2mimeTypeMap.put("==special==", mime);

  setMimeType(mime);
  return this;
}
```

Registering a document format

```
MimeType mime = new MimeType("text", "x-special");
mimeString2ClassHandlerMap.put(
    mime.getType()+ "/" + mime.getSubtype(), this);
```

- Create a `MimeType` object representing the "primary" MIME type for this format.
- Register this object as the handler for this MIME type.

```
mimeString2mimeTypeMap.put(
    mime.getType() + "/" + mime.getSubtype(), mime);
```

- Establish a mapping between the MIME string "text/x-special" and the primary `MimeType` object.
- To register a format against several different MIME types (e.g. text/json and application/json), add them to the `mimeString2mimeTypeMap`

Registering a document format

```
suffixes2mimeTypeMap.put("spec", mime);
```

- Register the file suffixes (not including the leading dot) that the format will handle, by mapping them to the primary `MimeType`
- Can add several different suffixes for the same type (txt, text, etc.)

```
magic2mimeTypeMap.put("==special==", mime);
```

- Add “magic numbers” – strings whose presence within the first 2kB of content will select the format
- E.g. “<?xml” is a strong predictor of XML documents.

Registering a document format

```
  setMimeType(mime);
  return this;
```

- Boilerplate.
- Suffixes and magic numbers are optional – don't use them if they don't make sense for your particular format.
- . . . but if neither are specified then only documents created with an explicit `mimeType` parameter will use the format.

Exercise: Document format registration

- `hands-on/yam-format` contains a simple document format implementation.
- Processes text files in the “YAM” format (the Wiki markup syntax used on `http://gate.ac.uk`).
- `unpackMarkup` has been written for you.
- Annotates `*bold*`, `_italic_` and `^teletype^` text, and section headings (lines starting `%1`, `%2`, etc.).
- For simplicity, does not modify the text or do repositioning, only adds Original markups annotations.

Exercise: Document format registration

- Your task – write the `init` method registration code
  - Primary MIME type “text/x-yam”
  - File suffixes “.yam” and “.gate”
  - No magic numbers
- To test, `ant jar` to build the JAR file, then load the `yam-format` directory as a plugin in GATE Developer.
  - Note the auto-instance created when the plugin loads
- Create a document from the `overview.yam` file and inspect the Original markups.

Solution

```
@Override
public Resource init() throws
    ResourceInstantiationException {
  // create the primary MIME type
  MimeType mime = new MimeType("text", "x-yam");
  // usual boilerplate
  mimeString2ClassHandlerMap.put(
      mime.getType()+ "/" + mime.getSubtype(), this);
  mimeString2mimeTypeMap.put(
      mime.getType() + "/" + mime.getSubtype(), mime);
  // file suffixes
  suffixes2mimeTypeMap.put("yam", mime);
  suffixes2mimeTypeMap.put("gate", mime);
  // more boilerplate
  setMimeType(mime);
  return this;
}
```

Further Reading

- Spring: `http://www.springsource.org`
- Groovy: `http://groovy.codehaus.org`
  - `http://gate.ac.uk/userguide/sec:api:groovy` for GATE details.
  - Also worth a look: Grails: `http://grails.org`. A Groovy- and Spring-based rapid development framework for web applications. We use Grails for Mímir, GATE Wiki (which runs gate.ac.uk) and the front end of GATECloud.net.