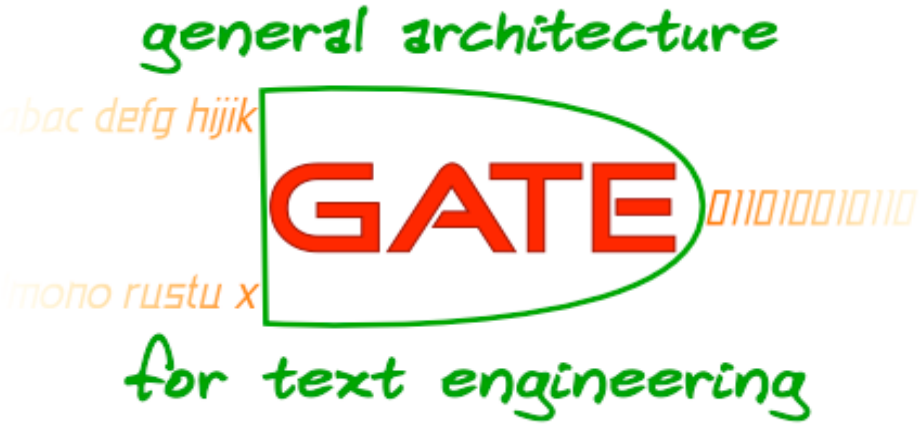

GATE Applications as Web Services

Ian Roberts

Introduction

- Scenario:
  - Implementing a web service (or other 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
  - Call Gate.init() once (and only once) before using any other GATE APIs

Initialisation using a ServletContextListener

- ServletContextListener is registered in web.xml

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

- Called when the application starts up

```
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);
  // site config is gateHome/gate.xml
  // plugins dir is gateHome/plugins
  Gate.init();
}
```

GATE in a multithreaded environment

- GATE PRs are *not* thread-safe
  - Due to design of parameter-passing as JavaBean properties
- Must ensure that a given PR/Controller instance is only used by one thread at a time

First attempt: one instance per request

- Naïve approach - create new PRs for each 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 nested try/finally: ugly but necessary to make sure we clean up even when errors occur. You will get very used to these…

Problems with this 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)

Second attempt: using ThreadLocals

- Store the PR/Controller in a thread local variable

```
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
}
```

Better than attempt 1…

- Only initialise resources once per thread
- Interacts nicely with typical web server thread pooling
- But if a thread dies, no way to clean up its controller
  - Possibility of memory leaks

A 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

```
private BlockingQueue<CorpusController> pool;

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

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

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

Blocks if the pool is empty: use `poll()` if you want to handle empty pool yourself

Further reading

- Spring Framework
  - http://www.springsource.org/
  - Handles application startup and shutdown
  - Configure your business objects and connections between them using XML
  - 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, …)

Conclusions

- Only use GATE Resources in one thread at a time
- Make sure to clean up after yourself, even when things go wrong
  - try/finally
  - Whenever you createResource, be sure to deleteResource