Advanced GATE Embedded

Module 8, part 3

Eleventh GATE Training Course
June 2018

Outline

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

2. Extending GATE
   - Adding new document formats

Outline

1. GATE in Multi-threaded/Web Applications
   - Multi-threading and GATE
   - Servlet Example
   - The Spring Framework
   - Making your own PRs duplication-friendly
2. 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, Grails. . .)

Setting up

- GATE libraries in `WEB-INF/lib`
  - dependencies from central via your build tool, or copy the `lib` folder from GATE installation
- Optional:
  - user config file if you need to configure things in there (e.g. “add space on markup unpack”)

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) {
  // if you want a user config
  Gate.runInSandbox(false);
  ServletContext ctx = e.getServletContext();
  File userConfig = new File(
      ctx.getRealPath("/WEB-INF/user.xml"));
  Gate.setUserConfigFile(userConfig);
  // otherwise ignore this and use the default (sandbox=true)

  // initialise GATE
  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

```java
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

```java
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, JAPE grammars, etc.)

Take Two: using ThreadLocal

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
}
```

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:

```
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:

```
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 learning framework
  - 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, Groovy or Java annotations.
- Handles application startup and shutdown
- GATE provides helper library to initialise GATE, load saved applications, etc.
- Built-in support for object pooling
- Web application framework (Spring MVC)
- Used by other frameworks (Grails, etc.)

Initializing GATE via Spring XML

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:gate="http://gate.ac.uk/ns/spring">
  <gate:init run-in-sandbox="false"
      user-config-file="gate-files/user-gate.xml" />

  <gate:extra-plugin group-id="uk.ac.gate.plugins"
                     artifact-id="annie"
                     version="8.5" />
</beans>
```

- Paths can be full URLs (`file:/...`) or *resource* paths that are resolved appropriately by Spring

Loading a Saved Application

To load an application state saved from GATE Developer:

```
<gate:saved-application
        id="myApp"
        location="gate-files/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.

Worked example – a Spring Boot webapp

- *Spring Boot* is a framework to get a Spring-based application up and running with a few lines of code.
- "Convention over configuration" approach, providing sensible defaults that you can override.
- Builds using Maven 3 (or Gradle)
- Example is a Spring MVC web application
  - see hands-on materials for the code.

Setting up a Spring Boot app

- One “parent” and one “plugin” to add to Maven POM
- Add dependencies on the relevant “starter” modules, which themselves depend on the required libraries
    - `spring-boot-starter-web` for a basic Spring MVC application
    - your preferred view technology, in our case `spring-boot-starter-thymeleaf`
- And whatever other libraries you require
    - in this case we need `gate-spring`, which in turn depends on `gate-core`

Setting up a Spring Boot app

- Entry point is a simple boilerplate class with annotations.
- Automatically scans sub-packages for other annotated classes (controllers, etc.).

```java
package gatetutorial;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.
    SpringBootApplication;

@SpringBootApplication
public class TutorialApp {

  public static void main(String... args) {
    SpringApplication.run(TutorialApp.class, args);
  }
}
```

A simple example controller

- Our example is a single controller that presents an HTML form to enter text.
- When form is submitted, process the document with a GATE application and show the document features as a table.

Spring pooling support

- Spring's built-in AOP features offer support for object pooling.
- Given a bean definition, we can expose a *proxy* object with the same behaviour backed by a pool of instances
- *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, can configure further requests to block or fail.

The `gate:pooled-proxy` helper

- The machinery is all Spring but complex to configure.
- `gate-spring` provides a helper to automate this in Spring XML configuration.
- Don't pool GATE applications directly, instead pool a helper class that calls GATE.

```
<bean id="gateService"
      class="gatetutorial.service.GateService">
  <gate:pooled-proxy max-size="3"
                     initial-size="3" />
</bean>
```

More advanced pooling options

- 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.

The GateService

- The GateService is written assuming single-threaded access.

```java
public class GateService {
  // will be injected automatically
  @Autowired private CorpusController application;

  private Corpus corpus;

  @PostConstruct
  public void init() throws GateException {
    corpus = Factory.newCorpus("GateService");
    application.setCorpus(corpus);
  }
```

The GateService

```
  @PreDestroy
  public void destroy() {
    Factory.deleteResource(corpus);
    // not strictly necessary as gate:duplicate will handle this
    Factory.deleteResource(application);
  }
```

The GateService

```
  public FeatureMap processWithGate(Document doc)
                            throws GateException {
    try {
      corpus.add(doc);
      application.execute();
      return doc.getFeatures();
    } finally {
      corpus.clear();
    }
  }
}
```

Tying it together

- Initialize GATE and configure service pool in XML (in `src/main/resources`)

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

<bean id="gateService"
      class="gatetutorial.service.GateService">
  <gate:pooled-proxy max-size="3"
              initial-size="3" />
</bean>
```

Tying it together

- Add an annotation to `TutorialApp` to load the XML.

```
import org.springframework.context.annotation.ImportResource;
// ...

@SpringBootApplication
@ImportResource("/gate-beans.xml")
public class TutorialApp {
```

Tying it together

- And finally, autowire the `GateService` into controller, and call its methods without having to worry about threading.

```
// imports as required

@Controller
public class GateController {

  @Autowired
  private GateService gateService;
```

Tying it together

```
  @RequestMapping(value="/",
                  method = RequestMethod.POST)
  public String process(
      @ModelAttribute("params")
      AnnotationRequest params,
      Map<String, Object> model) throws GateException {
    Document doc = // extract text from request
    try {
      FeatureMap features =
            gateService.processWithGate(doc);
      model.put("features", features);
      return "index";
    } finally { Factory.deleteResource(doc); }
  }
}
```

Exercise 1: The Spring Boot example

- In `hands-on/webapps` you have the source code for the Spring Boot example we've been discussing.
- What's provided:
  - the `pom.xml` with the necessary dependencies,
  - source code for the controller and `GateService` (in `src/main/java`),
  - the Thymeleaf view with the text entry form and results table,
  - configuration in `src/main/resources`, including GATE config files and the bean definition XML.

Exercise 1: The Spring Boot example

- What's missing:
  - the GATE application iself. . .
- Use the document statistics PR from earlier.
- In GATE Developer, create a "corpus pipeline" application containing a tokeniser and your statistics PR.
- Right-click on the application and "Export for GATE Cloud".
  - This will save the application state along with all the plugins it depends on in a single zip file.
- Unpack the zip file under `src/main/resources/gate-files`
  - don't create any extra directories – you need `application.xgapp` and `maven-cache.gate` to end up in `gate-files`.

Exercise 1: The Spring Boot example

- Now you can run the application – in `hands-on/webapps` run `mvn spring-boot:run`
- Browse to `http://localhost:8080/`, enter some text and submit
- Watch the log messages. . .
- Notice the result page includes a feature “handledBy” – each service instance in the pool has a unique ID.
- Multiple submissions go to different instances in the pool.
- Try editing `src/main/resources/gate-beans.xml` and change the pooling configuration.
- Test concurrent requests – the service has a built-in delay to simulate a slow application.

Not Just for Webapps

- Spring Boot (and Spring in general) 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, which exposes one `void` method `processDocument`

```
<bean id="processor"
   class="gate.util.LanguageAnalyserDocumentProcessor">
  <property name="analyser" ref="gateApplication"/>
  <gate:pooled-proxy max-size="3" initial-size="3" />
</bean>
```

A simple command-line app

```java
@Component
public class GateCommand implements CommandLineRunner {
  @Autowired private DocumentProcessor proc;

  public void run(String... args) throws Exception {
    Document doc = Factory.newDocument(args[0]);
    try {
      proc.processDocument(doc);
      // ...
    } finally { Factory.deleteResource(doc); }
  }
}
```

The main entry point `TutorialApp` is unchanged from the web example.

A JMS message consumer

```
@Component
public class Receiver {
  @Autowired private DocumentProcessor proc;

  @JmsListener(destination = "someQueue",
               concurrency = "3")
  public void receive(String stringMessage) {
    Document doc = Factory.newDocument(stringMessage);
    try {
      proc.processDocument(doc);
      doStuffWithResults(doc);
    } finally { Factory.deleteResource(doc); }
  }
}
```

In this case we need to add `@EnableJms` to the entry point class, and relevant dependencies to the POM.

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 the `DocStats` PR that keeps a running total of the number of Tokens it has seen.
- Build this (using the Maven pom), load the plugin, create an application containing a tokeniser and a “Shared document statistics” PR, export for GATE Cloud 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 JAR file to the appropriate place under `gate-files/maven-cache.gate`)

Exercise 2: Solution

Getter and setter:

```
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)

```java
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
- Multi-threading and GATE
- Servlet Example
- The Spring Framework
- Making your own PRs duplication-friendly

2 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

```java
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, `mvn install`, then load the `yam-format` plugin into 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

```java
@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;
}
```