Advanced GATE Embedded

Module 5

Seventh GATE Training Course
June 2014

Outline

Outline

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).

```
1  public void contextInitialized(ServletContextEvent e) {
2    ServletContext ctx = e.getServletContext();
3    File gateHome = new File(
4        ctx.getRealPath("/WEB-INF"));
5    Gate.setGateHome(gateHome);
6    File userConfig = new File(
7        ctx.getRealPath("/WEB-INF/user.xml"));
8    Gate.setUserConfigFile(userConfig);
9    // default site config is gateHome/gate.xml
10   // default plugins dir is gateHome/plugins
11   Gate.init();
12 }
```

Initializing GATE using a ServletContextListener

You must register the listener in `web.xml`

```
1 <listener>
2   <listener-class>
3     gate.web.example.GateInitListener
4   </listener-class>
5 </listener>
```

Handling Concurrent Requests

Naïve approach – new PRs for every request

```
1  public void doPost(request, response) {
2    ProcessingResource pr = Factory.createResource(...);
3    try {
4      Document doc = Factory.newDocument(
5          getTextFromRequest(request));
6      try {
7        // do some stuff
8      }
9      finally {
10       Factory.deleteResource(doc);
11     }
12   }
13   finally {
14     Factory.deleteResource(pr);
15   }
16 }
```

Handling Concurrent Requests

Naïve approach – new PRs for every request

```
1  public void doPost(request, response) {
2    ProcessingResource pr = Factory.createResource(...);
3    try {
4      Document doc = Factory.newDocument(
5          getTextFromRequest(request));
6      try {
7        // do some stuff
8      }
9      finally {
10       Factory.deleteResource(doc);
11     }
12   }
13   finally {
14     Factory.deleteResource(pr);
15   }
16 }
```

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

```
1  private ThreadLocal<CorpusController> controller =
2       new ThreadLocal<CorpusController>() {
3
4    protected CorpusController initialValue() {
5      return loadController();
6    }
7  };
8
9  private CorpusController loadController() { ... }
10
11 public void doPost(request, response) {
12   CorpusController c = controller.get();
13   // do stuff with the controller
14 }
```

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:

```
1  private BlockingQueue<CorpusController> pool;
2
3  public void init() {
4    pool = new LinkedBlockingQueue<CorpusController>();
5    for(int i = 0; i < POOL_SIZE; i++) {
6      pool.add(loadController());
7    }
8  }
9
10 public void destroy() {
11   for(CorpusController c : pool) {
12     Factory.deleteResource(c);
13   }
14 }
```

Simple Example of Pooling

Processing requests:

```
15 public void doPost(request, response) {
16   CorpusController c = pool.take();
17   try {
18     // do stuff
19   }
20   finally {
21     pool.add(c);
22   }
23 }
```

Simple Example of Pooling

Processing requests:

```
15 public void doPost(request, response) {
16   CorpusController c = pool.take();
17   try {
18     // do stuff
19   }
20   finally {
21     pool.add(c);
22   }
23 }
```

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.

```
1  public class MyHandler
2          implements HttpRequestHandler {
3    // controller reference will be injected by Spring
4    public void setApplication(
5              CorpusController app) { ... }
6
7    // good manners to clean it up ourselves though this isn't
8    // necessary when using <gate:duplicate>
9    public void destroy() throws Exception {
10     Factory.deleteResource(app);
11   }
```

Spring Servlet Example

```
13    public void handleRequest(request, response) {
14      Document doc = Factory.newDocument(
15          getTextFromRequest(request));
16      try {
17        // do some stuff with the app
18      }
19      finally {
20        Factory.deleteResource(doc);
21      }
22    }
23  }
```

Tying it together

In `applicationContext.xml`

```
1  <gate:init ... />
2  <gate:duplicate id="myApp" return-template="true">
3    <gate:saved-application
4         location="/WEB-INF/application.xgapp" />
5  </gate:duplicate>
6
7  <!-- Define the handler bean, inject the controller -->
8  <bean id="mainHandler"
9        class="my.pkg.MyHandler"
10       destroy-method="destroy">
11   <property name="application" ref="myApp" />
12   <gate:pooled-proxy max-size="3"
13                      initial-size="3" />
14 </bean>
```

Tying it together: Spring Pooling

```
12  <gate:pooled-proxy max-size="3"
13                     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:

```
1 <listener>
2   <listener-class>
3     org.springframework.web.context.
          ContextLoaderListener
4   </listener-class>
5 </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:

```
7  <servlet>
8    <servlet-name>mainHandler</servlet-name>
9    <servlet-class>
10     org.springframework.web.context.support.
           HttpRequestHandlerServlet
11   </servlet-class>
12 </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

```
1 // load an application context from definitions in a file
2 ApplicationContext ctx =
3     new FileSystemXmlApplicationContext("beans.xml");
4
5 DocumentProcessor proc = ctx.getBean(
6     "documentProcessor", DocumentProcessor.class);
7
8 // in worker threads...
9 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`)

```
1 private Map dataTable;
2
3 public Map getDataTable() { return dataTable; }
4
5 @Sharable
6 public void setDataTable(Map m) {
7   dataTable = m;
8 }
```

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.

```
1  public Resource init() throws /* ... */ {
2    if(dataTable == null) {
3      // only need to build the data table if we weren't given a shared one
4      buildDataTable();
5    }
6  }
7
8  public void reInit() throws /* ... */ {
9    // clear sharables on reInit
10   dataTable = null;
11   super.reInit();
12 }
```

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:

```
1  private AtomicInteger totalCount;
2
3  public AtomicInteger getTotalCount() {
4    return totalCount;
5  }
6
7  @Sharable
8  public void setTotalCount(AtomicInteger tc) {
9    this.totalCount = tc;
10 }
```

Exercise 2: Solution

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

```
1  public Resource init() throws
2          ResourceInstantiationException {
3    if(totalCount == null) {
4      totalCount = new AtomicInteger(0);
5    }
6    return this;
7  }
8
9  public void reInit() throws
10         ResourceInstantiationException {
11   totalCount = null;
12   super.reInit();
13 }
```

`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.

```
1 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)

```
1  public Resource duplicate(DuplicationContext ctx)
2          throws ResourceInstantiationException {
3    // duplicate this controller in the default way - this handles subclasses nicely
4    Controller c = (Controller)Factory.defaultDuplicate(
5                    this, ctx);
6
7    // duplicate each of our PRs
8    List<ProcessingResource> newPRs =
9          new ArrayList<ProcessingResource>();
10   for(ProcessingResource pr : prList) {
11     newPRs.add((ProcessingResource)Factory.duplicate(
12           pr, ctx));
13   }
14   // and set this duplicated list as the PRs of the copy
15   c.setPRs(newPRs);
16
17   return c;
18 }
```

Outline

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

```
1  import gate.*;
2  import gate.creole.metadata.*;
3  import gate.corpora.*;
4
5  @CreoleResource(name = "Example DocumentFormat",
6      autoinstances = {@AutoInstance})
7  public class MyDocumentFormat
8            extends TextualDocumentFormat {
9    // ...
10 }
```

- `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

```
1 public Boolean supportsRepositioning() {
2   return false;
3 }
```

DocumentFormat methods

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

```
1 public void unpackMarkup(Document doc)
2       throws DocumentFormatException {
3   AnnotationSet om = doc.getAnnotations(
4       GateConstants.ORIGINAL_MARKUPS_ANNOT_SET_NAME);
5   // Make changes to the document content, add annotations to om
6 }
```

- 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`

```
1  public Resource init() throws
       ResourceInstantiationException {
2    MimeType mime = new MimeType("text", "x-special");
3    mimeString2ClassHandlerMap.put(
4        mime.getType()+ "/" + mime.getSubtype(), this);
5    mimeString2mimeTypeMap.put(
6        mime.getType() + "/" + mime.getSubtype(), mime);
7    suffixes2mimeTypeMap.put("spec", mime);
8    magic2mimeTypeMap.put("==special==", mime);
9
10   setMimeType(mime);
11   return this;
12 }
```

Registering a document format

```
2   MimeType mime = new MimeType("text", "x-special");
3   mimeString2ClassHandlerMap.put(
4       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.

```
5   mimeString2mimeTypeMap.put(
6       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

```
7   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.)

```
8   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

```
10    setMimeType(mime);
11    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

```
1  @Override
2  public Resource init() throws
       ResourceInstantiationException {
3    // create the primary MIME type
4    MimeType mime = new MimeType("text", "x-yam");
5    // usual boilerplate
6    mimeString2ClassHandlerMap.put(
7        mime.getType()+ "/" + mime.getSubtype(), this);
8    mimeString2mimeTypeMap.put(
9        mime.getType() + "/" + mime.getSubtype(), mime);
10   // file suffixes
11   suffixes2mimeTypeMap.put("yam", mime);
12   suffixes2mimeTypeMap.put("gate", mime);
13   // more boilerplate
14   setMimeType(mime);
15   return this;
16 }
```