A.1 Guía de uso de una SLA-Driven API

A.1.1 Introducción

En esta sección se pretende realizar una práctica guía de uso de API dirigida por Acuerdos a Nivel de Servicio. Supondremos que el lector conoce los conceptos básicos del concepto de SLA y de su formalización en el lenguaje iAgree, así como unas nociones básicas de las clases Java de JAVAX.SERVLET. Antes de todo veamos cuál es es problemas que esperamos resolver y cómo es la solución propuesta mediante una biblioteca de gestión de acuerdos.

Supongamos que hemos diseñado una API para nuestro proyecto u organización y deseamos establecer diversos planes, cada uno de ellos con diferentes características o garantías. Estos planes, a efectos prácticos, impondrán una serie de restricciones y prioridades sobre los diversos servicios que ofrecemos. Nuestro sistema debe ser consciente de estos planes y, según el usuario que realice la petición, debe comportarse de cierta forma determinada.

Para hacer esto, lo habitual es pensar en tener los diversos planes en la lógica de nuestro servicio[1]. El plan está fuertemente acoplado con nuestra aplicación que presta servicio. ¿Y si usamos el concepto de SLA? Y, en tal caso, ¿cómo podemos aprovechar su potencial?

Por suerte, existen numerosas líneas de investigación enfocadas a la computación orientada a servicios y, en particular, a la formalización, gestión y razonamiento de acuerdos a nivel de servicio. Concretamente, el Grupo ISA, del Dpto. Lenguajes y Sistemas Informáticos de la Universidad de Sevilla, está desarrollando una biblioteca en Java para tal tarea. Durante esta guía la denominaremos *Agreement Management Library (AML)*.

[1]Se están popularizando también los API Proxies, que sirven de fachada de nuestra API y nos facilita la gestión del pricing y seguridad. Algunos ejemplos son: Mashape y 3Scale.

A.1.2 Entorno de ejemplo

Para que sirva de referencia a la hora de realizar un proyecto similar, vamos a partir de una API ya realizada, concretamente concretamente para la asignatura *Sistemas Orientados a Servicios*. Además, aprovechando la *free tier* que Google AppEngine ofrece, se ha desplegado el proyecto de ejemplo en la nube[2].

El objetivo del ejemplo es poder contabilizar y limitar las llamadas a la API realizadas por cada usuario en función de un acuerdo a nivel de servicio formalizado.

En líneas generales, el proyecto tiene presenta una estructura como se muestra en la figura §A.1.

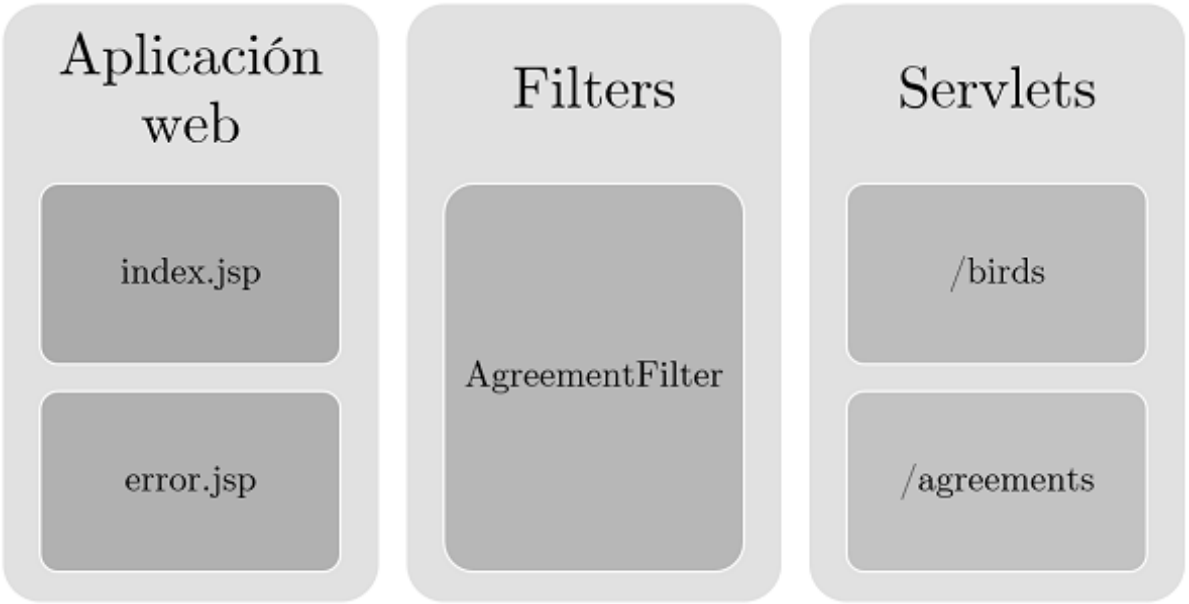

Figura A.1: Estructura general del ejemplo de una SLA-Driven API.

- La API está constituida sin usar tecnologías específicas como JAX-WS y es muy sencilla, pues sólo almacena objectos de tipo *Bird* y hace un CRUD REST con ellos. La documentación completa se puede ver en el sitio wen de Mashape, aunque, en este ejemplo usaremos únicamente los endpoints de tipo GET.
- Supondremos que cada llamada a la API supone el consumo de un crédito. Esto podría refinarse para contabilizar sólo las peticiones exitosas o que incluyan

[2] A modo de curiosidad, un papamoscas (*Muscicapidae*) es un tipo de ave insectívora que se puede encontrar por ejemplo, en el Parque Natural de Doñana.

ciertos parámetros, pero para este ejemplo, es mejor mantener la simplicidad.

- Cada usuario hace la llamada desde un endpoint que tiene una forma como esta: /API/V<VERSION>/BIRDS/{ID}?USER=<TOKEN>. Donde <VERSION> es la versión de API que se indique en la documentación y <TOKEN> es, en el caso más simple, el identificador del usuario.

Una vez definida toda la arquitectura, interesa ir al tema fundamental: el acuerdo que modela los diferentes planes de la API. Para ello partimos de una plantilla de acuerdos por cada plan que se ofrezca a los clientes de la API. En este sencillo ejemplo, tenemos los planes *basic plan*, *medium plan* y *pro plan*. En el fragmento A.1 se puede ver un ejemplo para el plan básico formalizado en el lenguaje *iAgree*.

```
1  Template basicPlanT version 1.0
2   Initiator: "Papamoscas SL";
3   Provider "Papamoscas SL" as Responder;
4   Metrics:
5   price: integer [0..500];
6   int: integer[0..10000];
7   AgreementTerms
8   Service BirdAPI availableAt
      ↪ "http://papamoscas-tfg.appspot.com/api/v3/birds"
9    GlobalDescription
10   PlanPrice: price = 9;
11   MonitorableProperties
12   global:
13    Requests: int = 0;
14   GuaranteeTerms
15   RequestTerm: Consumer guarantees Requests <=10;
16  CreationConstraints
17 EndTemplate
```

Código A.1: Plantilla del acuerdo para el *basic plan*.

Hemos hablado de plantillas de acuerdo, pero éstas no llegan a ser un acuerdo completo todavía. El siguiente paso es que cada cliente presente el acuerdo definitivo con las variaciones que permitan los *CreationConstraints* de la plantilla. En este sencillo ejemplo, al no existir restricciones sobre la creación de ofertas, se puede afirmar que la plantilla ya es, en sí, un acuerdo. Lo único que habrá que hacer será registrar una

copia de la plantilla por cada usuario dado de alta, siempre según el plan que hubiese contratado.

Salvando estos detalles formales, conviene resaltar la esencia del acuerdo, el *Guarantee Term* donde el consumidor de la API garantiza que sus peticiones serán menores que un determinado valor (*RequestTerm: Consumer guarantees Requests <=10;*). Es fácil ver que si actualizamos el número de peticiones cada vez, para decidir si una petición entrante es servida o no, basta con evaluar este *Guarantee Term*.

Para facilitar la lectura de los acuerdos y de la evolución de los mismos, se ha diseñado una simple API que se encuentra en el siguiente endpoint: /API/V<VERSION>/AGREEMENTS/{ID}. Donde <VERSION> es la versión de API que se indique en la documentación.

A.1.3 Poniendo en marcha el ejemplo

Supondremos que el lector tiene instalado un entorno de desarrollo con algún plugin para desplegar con *AppEngine* de Google. Es muy frecuente usar *Eclipse*, no obstante, otros IDE como *Netbeans* o *IntelliJ Ideas* funcionarán a la perfección.

- En primer lugar debemos clonar el repositorio de AML: . En la carpeta *samples/-papamoscas* se encuentra el proyecto de ejemplo.
    - Para usar AML en cualquier otro proyecto, debemos importar el JAR que se proporciona.
    - Si todo ha ido bien, tendremos un proyecto como el de la figura §A.2.

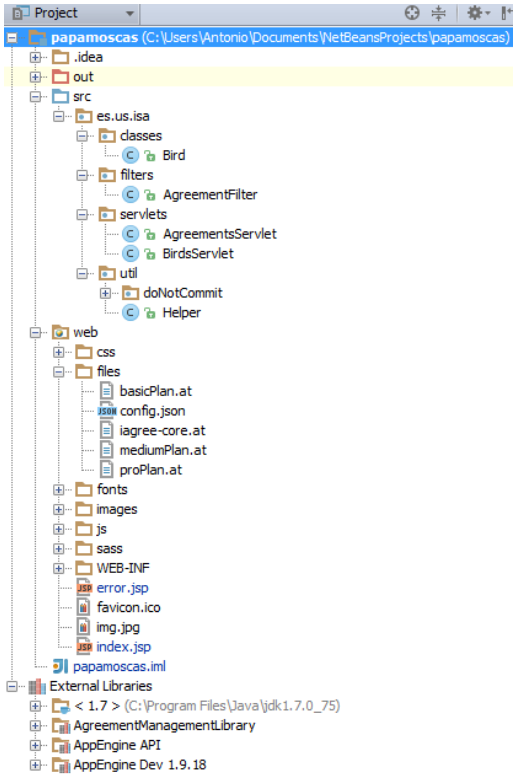

Figura A.2: Estructura del proyecto de ejemplo.

- La estructura de nuestro proyecto queda así:
    - En *web/files* se encuentran las plantillas, como la que vimos en A.1.
    - La clase *AgreementFilter* (A.3) contiene lo fundamental del proyecto: aprobar o denegar peticiones en función de un determinado SLA.

- La clase *AgreementServlet* (A.4) es una API REST de acuerdos. De esta forma, podemos ver la evolución del modelo de acuerdo según las peticiones realizadas.
- La clase *BirdsServlet* es una simple API REST que no tiene información alguna sobre acuerdos ni toma decisiones sobre servir o no la petición.
- La clase *Helper* (A.2) es un *singleton* que usaremos para instanciar una única vez la biblioteca y otros recursos.

- A continuación convendría compilar y desplegar el código en un servidor local que nos proporciona el SDK de Google *AppEngine*. Para probar que todo es correcto, realizaremos una secuencia como la que sigue:

  1. Comprobar que el servidor arranca sin excepciones.
  2. Hacemos una petición de tipo GET sobre el recurso LOCALHOST:PORT/API/V3/BIRDS?USER=PROUSER1. Debe devolver una lista de cuatro objetos.
  3. Hacemos lo mismo, pero cambiando el token de usuario por *basicUser1*. Realizamos diez peticiones con este usuario. Verificar que a la undécima vez, un código de error nos es enviado.

- Si además quisiéramos probar con más usuarios, las cuotas por defecto de cada usuario son las mostradas en la figura §A.3.

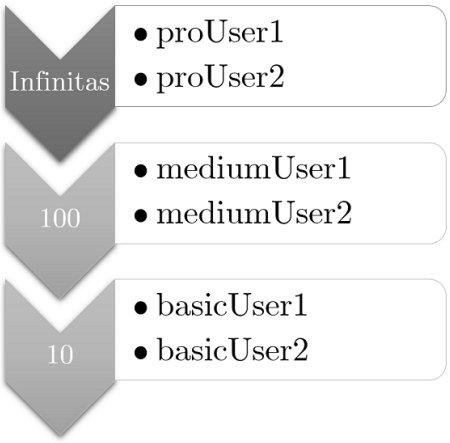

Figura A.3: Usuarios y cuotas por defecto.

- Habiendo realizado lo anterior de forma satisfactoria, veamos qué partes podemos cambiar con facilidad para obtener un funcionamiento distinto.

- En primer lugar podríamos editar las plantillas de acuerdos. Actualmente hay tres planes que se pueden alterar como se desee antes de desplegar la aplicación. En concreto, podemos cambiar el *Guarantee Term RequestTerm: Consumer guarantees Requests <=10;* y así controlar el número de peticiones deseado.
- Otra aspecto fácil de modificar es la inicialización de usuarios. Desde el método *loadTestAgreements()* de la clase *Helper* podemos cambiarlo.
- En caso de querer añadir una plantilla de acuerdo más, basta con añadirlo a la carpeta */web/files* (el identificador de la plantilla es que el aparece en ella, no el nombre del archivo) y luego añadir un usuario como arriba se comenta.
- Finalmente, si deseamos modificar la lógica de nuestro filtro, basta con editar el método *authorizeRequest(HttpServletRequest req)* y determinar el funcionamiento en función de los acuerdos definidos. Si cambiamos la lógica aquí, posiblemente sea preciso modificar el método *requestDone (HttpServletRequest req)*.

En definitiva, con este sencillo ejemplo se ha puesto de manifiesto la facilidad con la que una API cualquiera puede pasar a ser SLA-Driven con sólo incluir un filtro y una biblioteca.

Para más información sobre el uso de AML es posible consultar el Anexo §A.2.

A.2 NOCIONES BÁSICAS SOBRE AML

La librería *AML* nos proporciona un modelo de datos para trabajar con acuerdos a nivel de servicio, tal y como vemos en al figura §A.4.

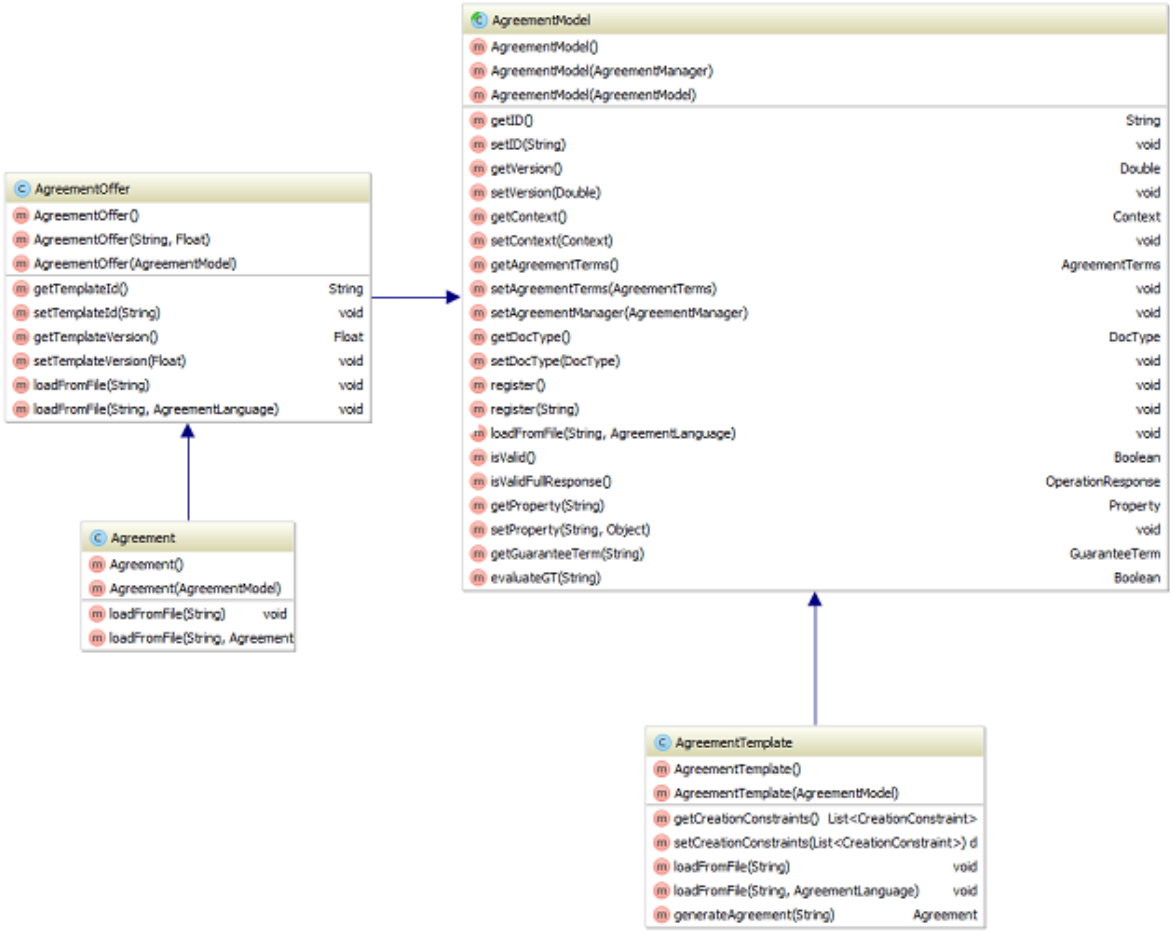

Figura A.4: Diagrama de clases del modelo de acuerdos.

Asimismo, la biblioteca nos ofrece un conjunto de métodos para trabajar con acuerdos. La figura §A.5 nos ilustra bien la funcionalidad que ofrece.

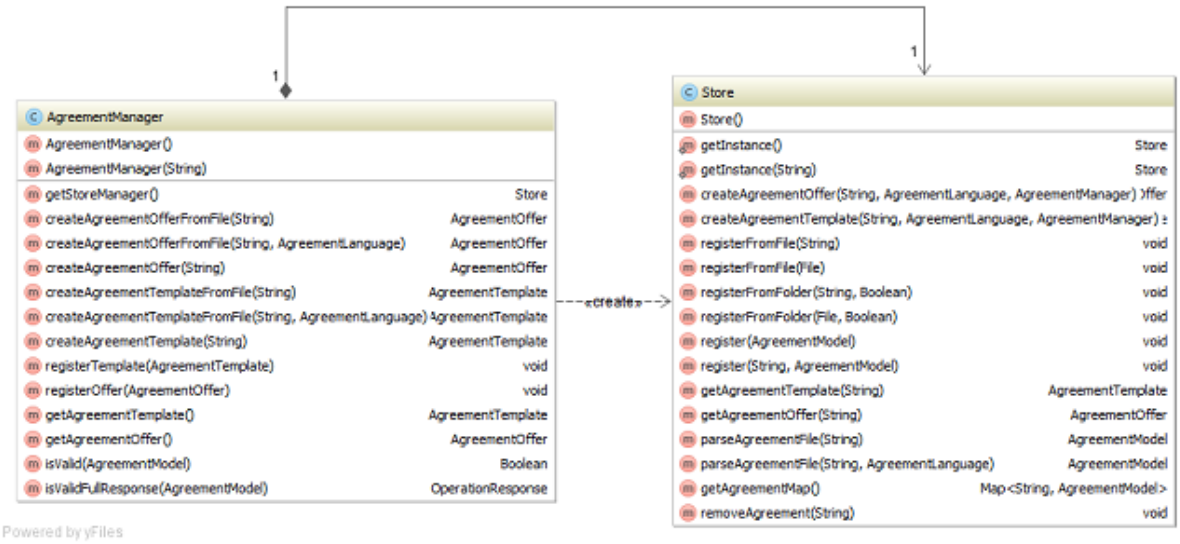

Figura A.5: Diagrama de clases de las clases principales.

A.3 CÓDIGOS

```
1  public class Helper {
2    private static final Logger LOG =
      ↪ Logger.getLogger(Helper.class.getName());
3    private static Helper instance;
4    private static AgreementManager agr;
5
6    protected Helper() {
7    }
8
9    public static Helper getInstance() {
10     if (instance == null) {
11       init();
12       instance = new Helper();
13       loadTestAgreements();
14     }
15     return instance;
16   }
17
18   public static AgreementManager getAgreementManager() {
19     return agr;
20   }
21
22   private static void init() {
23     agr = new AgreementManager();
24     agr.getStoreManager().registerFromFolder(new File("templates").
      ↪ getAbsolutePath(), false);
25   }
26
27   private static void loadTestAgreements() {
28     loadTestAgreement("basicPlanT", "basicUser1");
29     loadTestAgreement("basicPlanT", "basicUser2");
30     loadTestAgreement("mediumPlanT", "mediumUser1");
31     loadTestAgreement("mediumPlanT", "mediumUser2");
32     loadTestAgreement("proPlanT", "proUser1");
33     loadTestAgreement("proPlanT", "proUser2");
34   }
35
```

```
36    private static void loadTestAgreement(String templateName, String
       ↪ clientId) {
37      try {
38        agr.getStoreManager(). getAgreementTemplate(templateName).
       ↪ generateAgreementOffer(clientId). generateAgreement(clientId).
       ↪ register(clientId);
39      } catch (Exception e) {
40        LOG.log(Level.SEVERE, "Agreement Template " + templateName + "
       ↪ could not be loaded");
41      }
42    }
43
44    public Translator getIAgreeTranslator() {
45      return new Translator(new IAgreeBuilder());
46    }
47  }
```

Código A.2: Clase Helper

```
1   public class AgreementFilter implements Filter {
2
3     private static final Logger LOG =
       ↪ Logger.getLogger(AgreementFilter.class.getName());
4     private AgreementManager agr =
       ↪ Helper.getInstance().getAgreementManager();
5     private String clientId;
6
7     @Override
8     public void init(FilterConfig fConfig) throws ServletException {
9       ServletContext context = fConfig.getServletContext();
10    }
11
12    @Override
13    public void doFilter(ServletRequest request, ServletResponse
       ↪ response, FilterChain chain) throws IOException,
       ↪ ServletException {
14      HttpServletRequest req = (HttpServletRequest) request;
15      HttpServletResponse resp = (HttpServletResponse) response;
16      clientId = req.getParameter("user");
17      requestDone();
```

```
18      if (authorizeRequest()) {
19        chain.doFilter(request, response);
20        LOG.log(Level.INFO, "Request accepted");
21      } else {
22        resp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
23        resp.sendRedirect("/error.html");
24        LOG.log(Level.INFO, "Request rejected");
25      }
26    }
27
28    @Override
29    public void destroy() {
30
31    }
32
33    private boolean authorizeRequest() {
34      Agreement agreement =
       ↪ agr.getStoreManager().getAgreement(clientId);
35      if (agreement != null) {
36        return agreement.evaluateGT("RequestTerm");
37      }
38      return false;
39    }
40
41    private void requestDone() {
42      Integer numReq;
43      Agreement agreement =
       ↪ agr.getStoreManager().getAgreement(clientId);
44      if (agreement != null) {
45        if (agreement.getProperty("Requests").getExpression() == null)
       ↪ {
46          agreement.setProperty("Requests", 0);
47        }
48        numReq = agreement.getProperty("Requests").intValue();
49        numReq++;
50        agreement.setProperty("Requests", numReq);
51      }
52    }
53  }
```

Código A.3: Clase AgreementFilter

```
1  public class AgreementsServlet extends HttpServlet {
2
3    @Override
4    public void doGet(HttpServletRequest req, HttpServletResponse
      ↪ resp) throws IOException {
5      AgreementManager agr =
      ↪ Helper.getInstance().getAgreementManager();
6      Translator t = Helper.getInstance().getIAgreeTranslator();
7
8      String resourcePath = req.getPathInfo();
9      String agreement = "";
10
11     try {
12       if (resourcePath != null && !resourcePath.equals("/")) {
13         String[] resources = resourcePath.split("/");
14         if (resources.length > 2) {
15           resp.sendError(HttpServletResponse.SC_BAD_REQUEST, "Check
      ↪ URI");
16         } else {
17           String clientId = resources[1];
18           if (agr.getStoreManager().getAgreementMap().
      ↪ containsKey(clientId)) {
19             agreement = t.export(agr.getStoreManager().
      ↪ getAgreementMap().get(clientId));
20           } else {
21             resp.sendError(HttpServletResponse.SC_NO_CONTENT, "No
      ↪ data");
22           }
23         }
24       } else {
25         StringBuilder sb = new StringBuilder();
26         List<AgreementModel> models = new
      ↪ LinkedList<>(agr.getStoreManager().
      ↪ getAgreementMap().values());
27
28         Collections.sort(models, new Comparator<AgreementModel>() {
```

```
29            @Override
30            public int compare(AgreementModel o1, AgreementModel o2) {
31              return o1.getDocType().toString().
    ↪ compareToIgnoreCase(o2.getDocType().toString());
32            }
33          });
34
35          for (AgreementModel am : models) {
36            sb.append(t.export(am)).append("\n--------------\n");
37          }
38          agreement = sb.toString();
39        }
40        resp.setStatus(HttpServletResponse.SC_FOUND);
41        resp.setContentType("text/plain");
42        resp.setCharacterEncoding("UTF-8");
43        resp.getWriter().write(agreement);
44
45      } catch (Exception e) {
46        resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
47        LOG.log(Level.WARNING, null, e);
48      }
49    }
50  }
```

Código A.4: Clase AgreementsServlet