

Sistemas Distribuidos

REST

Rodrigo Santamaría

+ REST

- Uso de un servicio web
 - Autenticación
 - Mediante URIs
 - Mediante una aplicación Java
 - Parseo de XMLs
- Creación de un servicio web en Java



Uso de un servicio web

Autenticación



- Actualmente, casi todos los servicios web requieren algún tipo de autenticación previa
 - Generalmente a través de **OAuth** (Open Authorization), un protocolo de autenticación de APIs
 - O mediante algún sistema más sencillo de registro
 - Complica las invocaciones a la API (sobre todo de manera ‘manual’)
 - Mejora la seguridad de los servidores de servicios web



Uso de un servicio web

Servicios



- Existen muchos servicios web cuya API se puede utilizar (generalmente, previa autenticación)
- Una buena colección actualizada:
 - <http://www.programmableweb.com/>
- Un par de ejemplos que no necesitan autenticación:
 - Agencia Estatal de Meteorología (Aemet)
 - http://www.aemet.es/xml/municipios/localidad_37274.xml
 - Kyoto Encyclopedia of Genes and Genomes (KEGG)
 - <http://rest.kegg.jp/find/genes/shiga+toxin>

+ REST

Uso de un servicio web en Java

Utilizamos clases de *java.net* y *java.io*, como para acceder a cualquier otro recurso web:

```
URL url = new URL("http://rest.kegg.jp/find/genes/shiga+toxin");
URLConnection conn = (URLConnection) url.openConnection();
conn.setRequestMethod("GET");
```

```
if (conn.getResponseCode() != 200) {
    throw new RuntimeException("Failed : HTTP error code : "
        + conn.getResponseCode());
}
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(
    (conn.getInputStream())));
```

```
String output;
System.out.println("Output from Server .... \n");
while ((output = br.readLine()) != null) {
    System.out.println(output);
}
```

```
conn.disconnect();
```



REST

Parseo de XML en Java



- El servicio REST devuelve texto en algún formato
 - Debemos analizarlo para extraer la información que nos interese
 - Un formato muy común es XML
- Varias opciones para parsear XML
 - Si es un fichero sencillo: *BufferedReader* y *String*
 - Si es un fichero complejo:
 - Análisis (parseo) basado en etiquetas: *DOM*
 - Crea una estructura con un elemento por etiqueta
 - Análisis (parseo) basado en eventos: *SAX*
 - Lee el fichero y lanza un evento por cada etiqueta
 - Un buen tutorial:
 - <http://www.java-samples.com/showtutorial.php?tutorialid=152>

+ REST

Parseo: ejemplo con SAX

```
public class AemetXMLSAX extends DefaultHandler{

    public static void main(String[] args)
    {
        AemetXMLSAX axs=new AemetXMLSAX();
        axs.searchAndParse("09059");
    }

    public void searchAndParse(String query)
    {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        SAXParser sp;
        try {
            sp = spf.newSAXParser();
            sp.parse("http://www.aemet.es/xml/municipios/localidad_"+query
                    +".xml", this);
        }
        catch (ParserConfigurationException e){ e.printStackTrace(); }
        catch (SAXException e)           {e.printStackTrace();}
        catch (IOException e)             { e.printStackTrace(); }
    }
    ...
}
```

+ REST

Parseo: ejemplo con SAX

```
public void startElement(String uri, String localName, String qName, Attributes
attributes) throws SAXException
{
    System.out.println(" start element: "+qName);
}
```

```
public void endElement(String uri, String localName, String qName)
                                                                    throws SAXException
{
    System.out.println(" end element: "+qName);
}
```

```
public void characters(char[] ch, int start, int length) throws SAXException
{
    System.out.println(" start characters: "+ch);
}
```




REST

Parseo: ejemplo con SAX

```
<root>
  <child>
    <grandchild>text 1</grandchild>
  </child>
  <child>
    <grandchild>text 2</grandchild>
  </child>
</root>
```

```
start document :
start element  : root (qName)
characters     :
```

```
start element  : child
characters     :
```

```
start element  : grandchild
characters     : text 1 (ch)
end element    : grandchild
characters     :
```

```
end element    : child
characters     :
```

```
start element  : child
characters     :
```

```
start element  : grandchild
characters     : text 2
end element    : grandchild
characters     :
```

```
end element    : child
characters     :
```

```
end element    : root
end document   :
```



+ REST

- Uso de un servicio web
- Creación de un servicio web en Java
 - JAX-RS y anotaciones
 - Eclipse + Tomcat + Jersey
 - Programas servidor y cliente
 - Interfaces



Creación de un servicio REST

JAX-RS



- Para crear un servicio web necesitamos algo más que los objetos de Java para manejo de conexiones
- JAX-RS (Java API for RESTful web services) es una API de Java para crear servicios web tipo REST
 - Jersey (jersey.java.net) es su implementación más estable
- Un objeto java (*POJO* – Plain Old Java Object) se convierte en un recurso web añadiéndole **anotaciones**
 - Sintaxis incorporada a Java en la versión 1.5
 - Provee información sobre el código, pero no es código
 - Información para la compilación, desarrollo o ejecución



Creación de un servicio REST

JAX-RS: anotaciones



- `@Path` indica la ruta relativa a añadir a la URI para acceder a una clase o método
- `@GET`, `@PUT`, `@POST`, `@DELETE`, `@HEAD` hacen referencia al tipo de petición HTTP que satisface un método
- `@Produces` especifica el tipo MIME que retorna (plain, html, json, xml, etc.) un método
 - `@Consumes` especifica el tipo MIME que requiere un método
- Existen más anotaciones, éstas son sólo las esenciales



Creación de un servicio REST

JAX-RS: anotaciones

- Por ejemplo:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/saludo");
public String saludar(){ return "Hola"; }
```

- Retornará un mensaje en texto plano que dice “Hola” al acceder a <http://host:port/saludo> (método GET)



Creación de un servicio REST

Esquema

JAX-RS

Anotaciones
Java 1.5

Cliente REST
(vía JAX-RS)

Servicio REST

```
ClientConfig conf = new DefaultClientConfig();
Client client = Client.create(conf);

URI uri=UriBuilder
    .fromUri("http://ip:8080/servicePath").build();
WebResource service= client.resource(uri);

System.out.println(service.path("classPath")
    .path("hello").accept(MediaType.TEXT_PLAIN)
    .get(String.class))
```

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("hello");

public String saludar(){ return "Hola"; }
```



Creación de un servicio REST

Preparación del entorno



- Descargar **Tomcat** 6.0 de <http://tomcat.apache.org/>
- Descargar **Eclipse**
 - Instalar plugin para desarrollo web: WTP
 - Help/Install New Software...
 - <http://download.eclipse.org/releases/indigo>
 - Web, XML, Java EE, etc.
 - O bien descargar la versión para desarrolladores EE
- Descargar **Jersey** (<http://jersey.java.net>), buscar el enlace en Downloads (JAX-RS 2.0 API jar)
 - Al crear el proyecto tendremos que agregar dichos jars



Creación de un servicio REST

Creación del proyecto

- Crear un nuevo proyecto web:
 - File/New/Project... → Web/Dynamic Web Project
- En la carpeta `WebContent/WEB-INF/lib`, incluir todos los jars que hay en las carpetas `jersey/lib`, `jersey/api` y `jersey/ext`
- En <http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/> se encuentran algunas de las clases y ficheros que vamos a usar de ejemplo



Creación de un servicio REST

Fichero web.xml



- Modificar el fichero `WebContent/WEB-INF/web.xml` por este otro:
 - <http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/web.xml>
- `display-name` debe coincidir con el nombre del proyecto
- `jersey.config.server.provider.packages` debe tener como valor una lista de nombres de paquetes en los que tenemos recursos REST, separados por punto y coma.
- `url-pattern` dentro de `servlet-mapping` debe ser la ruta base a partir de la que se ubicarán los recursos REST



Creación de un servicio REST

Ejemplo de servicio

```
//Sets the path to base URL + /hello
@Path("/hello")
public class Hello
{
    // This method is called if TEXT_PLAIN is request
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayPlainTextHello() {
        return "Hello Jersey";
    }

    // This method is called if XML is request
    @GET
    @Produces(MediaType.TEXT_XML)
    public String sayXMLHello() {
        return "<?xml version='1.0'?>" + "<hello> Hello Jersey" + "</hello>";
    }

    // This method is called if HTML is request
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String sayHtmlHello() {
        return "<html> " + "<title>" + "Hello Jersey" + "</title>"
            + "<body><h1>" + "Hello Jersey" + "</body></h1>" + "</html> ";
    }
}
```



Creación de un servicio REST

Ruta del servicio

■ http://ip:8080/proyecto/servlet/clase/metodo

localhost o la ip del equipo remoto
(mejor ips que nombres, pues pueden estar corruptos en el lab. de informática)

indicado con la anotación `@Path` antes de un método
puede no usarse si se tiene ruta ya en la clase

indicado con la anotación `@Path` antes de una clase
podemos no usarlo, pero es recomendable para
guardar un cierto orden

indicado en el tag `<url-pattern>` de `<servlet-mapping>` en `web.xml`
p. ej. si queremos que sea servlet usamos `/servlet/*`
Podemos no usarlo, poniendo simplemente `/*`

nombre del proyecto en el IDE, que debe
coincidir con el tag `<display-name>` de `web.xml`



Creación de un servicio REST

Arranque del servicio



- Arrancar el servicio: Run/Run As.../Run on Server
 - Especificar Tomcat como servidor en el que arrancarlo
 - Target runtime (o *New...* si no está)
- Errores frecuentes:
 - **java.lang.ClassNotFoundException:** com.sun.jersey.spi.container.servlet.ServletContainer
 - Los jar de Jersey no se han incluido correctamente en WebContent/WEB-INF/lib
 - **com.sun.jersey.api.container.ContainerException:** The ResourceConfig instance does not contain any root resource classes.
 - El parámetro com.sun.jersey.config.property.packages no se ha configurado correctamente en web.xml: debe contener los nombres de los paquetes que contienen clases anotadas.
 - El servidor arranca pero no hay nada en las rutas esperadas
 - El parámetro com.sun.jersey.config.property.packages no se ha configurado correctamente en web.xml: debe contener los nombres de los paquetes que contienen clases anotadas.
 - Revisar los @Path, y los tags <display-name> y <servlet-mapping> en web.xml



Creación de un servicio REST

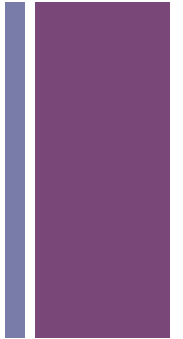
Ejemplo de cliente

```
public class Test {  
    public static void main(String args[])  
    {  
        Client client=ClientBuilder.newClient();  
        URI uri=UriBuilder.fromUri("http://localhost:8080/pruebasREST").build();  
  
        WebTarget target = client.target(uri);  
  
        System.out.println(target.path("rest").path("hello").request(MediaType.TEXT_PLAIN).get  
            (String.class));  
        System.out.println(target.path("rest").path("hello").request(MediaType.TEXT_XML).get  
            (String.class));  
        System.out.println(target.path("rest").path("hello").request(MediaType.TEXT_HTML).get  
            (String.class));  
    }  
}
```

Se ejecuta como una aplicación Java normal

<http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/Test.java>

+ Ejercicio



- Crear un servicio REST *hello* mediante Eclipse, Tomcat y Jersey.
- Iniciar en la máquina local y probar accesos de clientes
 - Desde un navegador y desde java
 - Desde la máquina local y desde otras máquinas



Creación de un servicio REST

Paso de argumentos

■ Paso de argumentos: anotación `@QueryParam`:

```
@Path("calculator")
public class Calculator
{
    @Path("sq")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String square(@DefaultValue("2") @QueryParam(value="num") long num)
    {
        return ""+num*num;
    }
}
```

■ Desde URL <http://hostname:port/calculator/sq?num=3>

■ Desde Java

- `service.path("calculator/sq").queryParams("num", ""+3).request(MediaType.TEXT_PLAIN).get(String.class)`



Creación de un servicio REST

Retorno de objetos



- En principio, Jersey retorna tipos MIME (es decir, texto, en distintos formatos)
 - Jersey no soporta la serialización de tipos primitivos
 - Debemos usar String + documentación de la API
 - Si intentamos, por ejemplo, retornar un long:
 - [com.sun.jersey.api.MessageException: A message body writer for Java class java.lang.Long, and Java type long, and MIME media type XXX was not found](#)
- Solución: convertir objetos Java en texto (p. ej. XML)
 - Jersey da soporte para ello a **JAXB**, una arquitectura para asociar clases Java a representaciones XML



Creación de un servicio REST

Retorno de objetos 'nuevos'

■ Usamos @XmlRootElement + APPLICATION_XML

declaración

```
@XmlRootElement
public class Planet
{
    public int id;
    public String name;
    public double radius;
}
```

cliente

```
Planet planet = service.path("rest/planet").request
(MediaType.APPLICATION_XML_TYPE).get(Planet.class);
```

servicio

```
@Path("planet")
public class Resource {
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public Planet getPlanet() {
        Planet p = new Planet();
        p.id = 1;
        p.name = "Earth";
        p.radius = 1.0;

        return p;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <planet>
    <id>1</id>
    <name>Earth</name>
    <radius>1.0</radius>
  </planet>
```



Creación de un servicio REST

Retorno de POJOs

- Usamos la clase JAXBElement + APPLICATION_XML

servicio

```
@Path("calendario")
public class Calendario {
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public JAXBElement<Date> getDate()
    {
        Date p = new Date(System.currentTimeMillis());
        return new JAXBElement<Date>(new QName("date"), Date.class, p);
    }
}
```

cliente

```
GenericType<JAXBElement<Date>> dateType = new GenericType<JAXBElement<Date>>() {};
Date fecha = (Date) service.path("rest/calendario").request
    (MediaType.APPLICATION_XML_TYPE).get(dateType).getValue();
System.out.println("### " + fecha.getTime());
```



Creación de un servicio REST

Minimización de interfaces



- Respecto al uso de argumentos y el retorno de objetos, un buen diseño de un sistema distribuido minimiza las interfaces
 - Suponen una carga en el tráfico de red
 - Y más si hay que convertirlos a XML
 - Incrementan el riesgo de errores
 - Interpretaciones equivocadas de la API
 - Las clases tienen que estar disponibles por los clientes
 - Etc.
 - Muchos objetos son evitables con un uso inteligente de String



Creación de un servicio REST

Ciclo de vida de los objetos

- En Jersey, los objetos tienen un ciclo de vida '*per-request*'
 - Cada clase que se ofrece como recurso se instancia con cada nueva petición y se destruye al terminar dicha petición
 - Esto impide mantener objetos que varían su estado a lo largo del tiempo (a través de distintas peticiones)
 - Solución:
 - Utilizar la anotación **@Singleton** para la clase
 - Así, la clase se instancia una vez por aplicación web, y permanece instanciada hasta que se apague o reinicie el servicio



+ Ejercicio

- Crear un servicio REST *calculator* que permita realizar potencias cuadradas (*sq*) y sumas de dos elementos (*add*)
 - Obtendrá mediante parámetros el número a elevar al cuadrado y los dos números a sumar (todos enteros)
 - Retornará el resultado como una cadena de texto
- Añadir una tercera función *stack*(int *n*) que sume el valor *n* a una variable interna del servicio que comienza en 0



REST

Tutoriales



- <http://www.vogella.com/articles/REST/article.html>
 - Preparación básica para trabajar con Jersey+Tomcat+Eclipse
- <https://jersey.java.net/documentation/latest/user-guide.html>
 - Manual completo de Jersey, en especial:
 - Paso de argumentos (cap 3.2)
 - Ciclo de vida de los recursos (3.4)

Carrera 100m lisos





Carrera 100m lisos

Servicio



- Crear un servicio REST mediante una clase Carrera100
 - El servicio se llamará **carrera100** y aceptará 4 atletas
 - Mantendrá información sobre
 - Número de atletas inscritos en la carrera
 - Tiempo de inicio de la carrera y de llegada de cada atleta
 - Ofrecerá los métodos
 - **reinicio**: pone los tiempos y los atletas inscritos a cero
 - **preparado**: detiene al atleta que lo llama hasta que todos los atletas estén preparados
 - **listo**: detiene al atleta que lo llama hasta que todos los atletas estén listos. Una vez estén todos listos, la carrera empieza
 - **llegada(dorsal)**: guarda el tiempo de llegada del atleta y retorna el tiempo obtenido por el atleta.
 - **resultados**: retorna una cadena con algún formato que muestre los resultados de la carrera

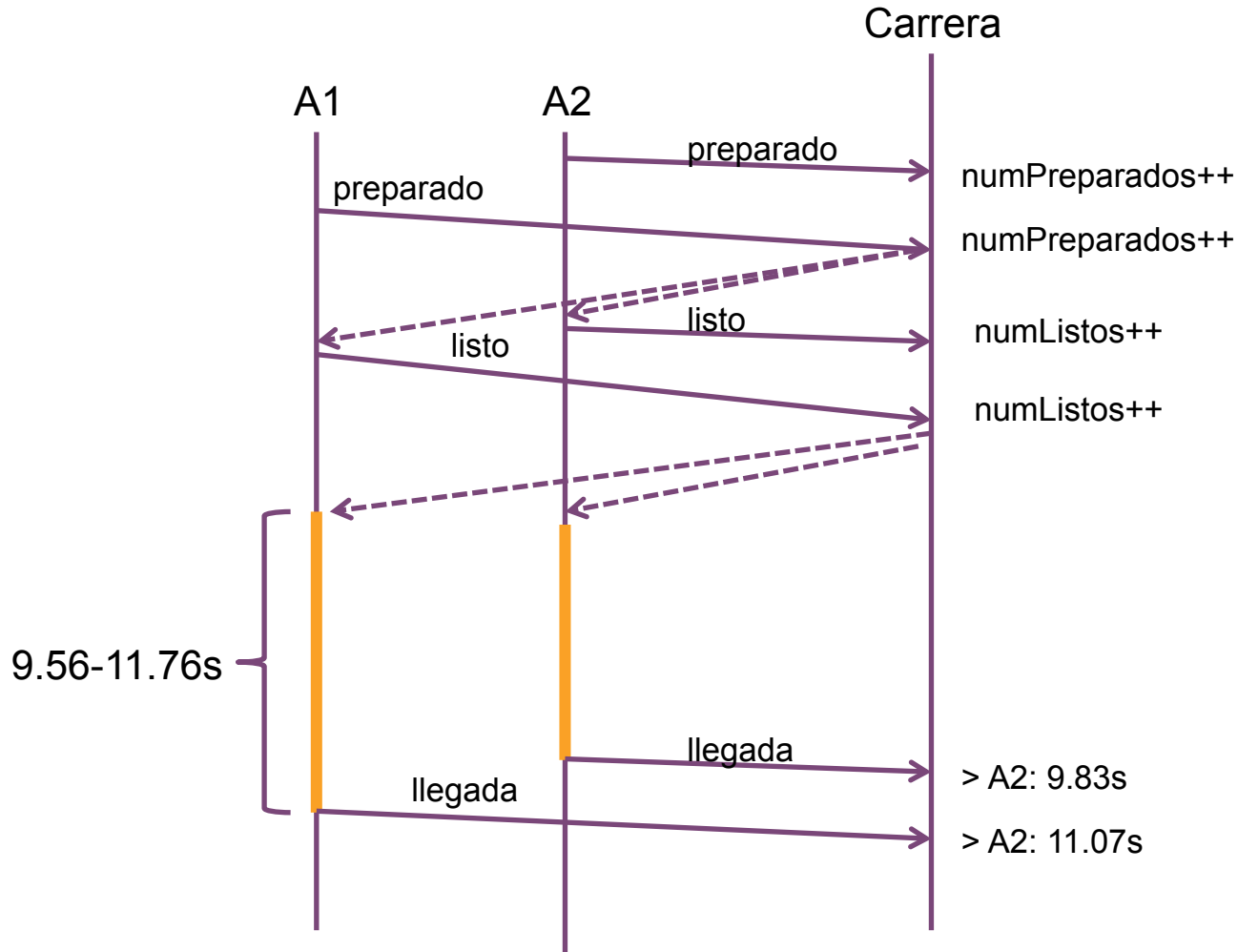
+ Carrera 100m lisos

Cliente

- La clase `Atleta` será un hilo (`Thread`) que:
 - Se construirá con un determinado dorsal
 - Durante su ejecución
 1. Invoca `carrera100/preparado`
 2. Invoca `carrera100/listo`
 3. Corre (duerme entre 9.56 y 11.76s)
 4. Invoca `carrera100/llegada?dorsal=midorsal`
- Para hacer una carrera puede haber una clase `MainCarrera`
 1. Invoca `carrera100/reinicio`
 2. Crea 4 `Atletas` y los pone a correr
 3. Invoca `carrera100/resultados`

+ Carrera 100m lisos

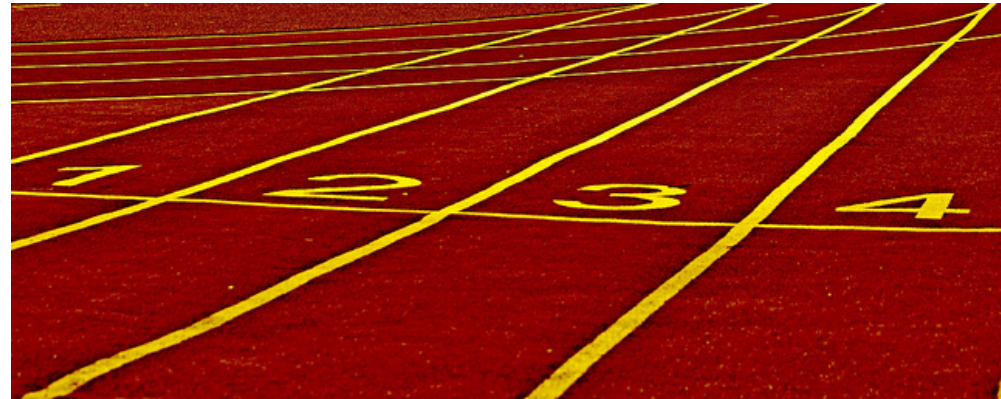
Ejemplo con 2 procesos



+ Carrera 100m lisos

Despliegue

- Ejecutar el servicio y la carrera en el mismo ordenador
- Probar con 2 ordenadores
 - En uno corre el servicio y dos atletas
 - En el otro corren los otros dos atletas
- Probar con 3 ordenadores, con 6 atletas
 - En cada uno corren dos atletas
 - En uno de ellos corre el servicio





Carrera 100m lisos

Despliegue: determinar IP del servidor



- Para que los clientes sepan dónde está

- `/sbin/ifconfig`

- `/sbin/ifconfig | grep 'inet addr:' | grep -v '127.0.0.1' | cut -d: f2 | awk '{print $1}'`

- Para extraer los número de la ip



Carrera 100m lisos

Despliegue: reparto de clases



■ Básico:

- Almacenar las clases en Z:
- Estarán disponibles en todos los ordenadores si nos conectamos con el mismo usuario

■ Avanzado:

- Pensando en otros sistemas donde no tengamos un servicio distribuido de directorios
- Podemos generar un script de envío remoto mediante ssh/scp
 - Ver los scripts lanzar.sh y shareKeys.sh en <http://vis.usal.es/rodrigo/documentos/sisdis/scripts/>

■ Pro:

- Podemos generar un .jar con las clases y bibliotecas necesarias y enviarlas mediante scripts/ssh

+ Carrera 100m lisos

Despliegue: ejecución

- El servidor se arranca inicialmente
 - **Básico:** usaremos Eclipse para ello
 - Avanzado: generar proyecto .war o similares
 - Pro: crear un demonio que arranque con el ordenador
- Luego arrancamos los clientes
 - Básico: a través de Eclipse (requiere arrancar Eclipse en todos los ordenadores)
 - Avanzado: ejecutarlos desde consola, localmente (requiere acceso físico a todos los ordenadores)
 - **Pro:** ejecutarlos desde consola, remotamente (todo se hace desde un solo ordenador)
 - Podemos usar los scripts vistos en el reparto de clases



Carrera 100m lisos

Despliegue: reinicio y resultados



■ Reinicio

- → Clase MainCarrera que reinicie el servicio
- → Reinicio manual a través del navegador
- Reinicio a través de un Atleta (p.ej. el que tiene dorsal 0)
 - Nos tenemos que asegurar que arranca antes que el resto

■ Resultados

- Las mismas opciones que para el reinicio



Carrera 100m lisos

Coordinación y tiempos



- Probar qué pasa si los Atletas no esperan a las órdenes de 'preparados' y 'listos', y empiezan a correr en cuanto pueden
 - En distintos despliegues
- Probar qué pasa si los tiempos los miden los propios Atletas
 - En distintos despliegues
 - De forma relativa (he tardado t_{final} menos $t_{inicial}$)
 - Obteniendo ellos el $t_{inicial}$
 - Tomándolo el $t_{inicial}$ de la carrera
 - De forma absoluta (he llegado en t_{final})

+ Carrera 100m lisos

Análisis

- ¿Qué posibles fallos encuentras en el sistema implementado?
 - Relativos a los tiempos
 - Relativos a la coordinación
 - Relativos a posibles fallos de proceso
 - Relativos a posibles fallos de comunicación
- ¿Se te ocurren mejoras posibles para el sistema?

