Desarrollo de Aplicaciones Distribuidas: Registrador de juegos

Rafael Gálvez-Cañero, Andreas Gerstmayr

Iteración 5 - 28 de Abril de 2015

Índice general

1. Datos generales **2**
1.1. Miembros del grupo . . . 2
1.2. Descripción del sistema . . . 2
1.2.1. Funcionalidad observable . . . 2
1.2.2. Servicios ofrecidos . . . 2
1.2.3. Servicios demandados . . . 3
1.3. Direcciones de descarga y planificación . . . 3
1.4. Seguimiento . . . 3

2. Modelado **4**
2.1. Análisis del sistema . . . 4
2.2. Arquitectura del sistema . . . 4

3. Iteración 3 **6**
3.1. Objetivos de iteración . . . 6
3.2. Gradle . . . 6
3.3. Dockerización . . . 7

4. Iteración 4 **8**
4.1. Objetivos de iteración . . . 8
4.2. Integración de MongoDB . . . 8
4.2.1. MongoDB y desarrollo . . . 8
4.2.2. MongoDB y despliegue . . . 9
4.2.3. Distinción de dominio, controlador y servicio . . . 9
4.3. Diagrama de paquetes . . . 10
4.4. Diagrama de clases . . . 10

5. Iteración 5 **12**
5.1. Objetivos de iteración . . . 12
5.2. Implementación inicial de API REST . . . 12
5.2.1. API . . . 12
5.3. Primeros tests . . . 13
5.4. Despliegue en plataforma Azure . . . 14
5.4.1. Test vm . . . 14
5.4.2. VM de producción . . . 14
5.5. Investigar integración contínua . . . 15

6. Iteracion 6 **16**
6.1. Objetivos de iteración 16
6.2. Integración de Swagger 16
6.2.1. Generar la documentación del API REST 16
6.2.2. Consumir la documentación 16
6.3. Testing, Integración contínua 17
6.4. UML actualizado 17

7. Iteracion 7 **19**
7.1. Objetivos de iteración 19
7.2. Finalizar cliente de GameRegistry para vert.x 19
7.3. Máquina virtual en producción 19
7.3.1. La máquina virtual 20
7.4. Test sobre *Promises* 20
7.4.1. Motivación 20
7.4.2. El test 21
7.5. Diagrama UML actualizado 22

8. Documentación de API GameRegister **23**

Índice de figuras

2.1. Modelo de clases 4
2.2. Modelo de despliegue del sistema 5

4.1. Organización de paquetes 10
4.2. Diagrama de clases actualizado (iteración 4) 11

5.1. Diagrama de despliegue (iteración 5) 15

6.1. Diagrama de clases (iteración 6) 18

Índice de cuadros

1.1. Miembros del grupo 2
1.2. Datos generales del trabajo en grupo 3
1.3. Tabla de seguimiento 3

Capítulo 1

Datos generales

1.1. Miembros del grupo

Apellidos	Nombre	Correo-e	Grupo
Gálvez-Cañero	Rafael	galvesband@gmail.com	18
Gerstmayr	Andreas	andreas.gerstmayr@gmail.com	18

Cuadro 1.1: Miembros del grupo

1.2. Descripción del sistema

- **Tipo de sistema distribuido**: Sistema de información.
- **Nombre del proyecto**: Plataforma de juegos, Game Registry.
- **Breve descripción**: Sub-sistema para registrar sesiones de juego e información asociada.

1.2.1. Funcionalidad observable

- Registrar el inicio y el término de todas las sesiones de juego.
- Visualizar el historial de juegos.

1.2.2. Servicios ofrecidos

- Servicio de Registro: Capacidad de aceptar la información de una sesión de juego (juego ID, jugador ID y fecha de inicio y término).
- Servicio de Historial: Ofrece métodos para consultar el historial de sesiones.

1.2.3. Servicios demandados

- Servicio de Juego: avisar de término de una sesión de juego
- Servicio de Juego: recibir el título de un juego
- Servicio de Perfil: rebibir el nombre de un jugador

1.3. Direcciones de descarga y planificación

Código fuente	`https://github.com/andihit/dad-gameregistry.git`
Planificación temporal	
Iteración 1	24/02/2015
Iteración 2	03/03/2015
Iteración 3	26/03/2015
Iteración 4	7/04/2015
Iteración 5	28/04/2015
Iteración 6	12/05/2015
Iteración 7	26/05/2015
Entrega Final	02/06/2015

Cuadro 1.2: Datos generales del trabajo en grupo

1.4. Seguimiento

	Iteración									
Estudiante	1	2	3	4	5	6	7	Final	Total	Pond.
Rafael Gálvez-Cañero	5	5	5	5	5	-	-	-	**25**	1
Andreas Gerstmayr	5	5	5	5	5	-	-	-	**25**	1
Total	10	10	10	10	10	0	0			

Cuadro 1.3: Tabla de seguimiento

Capítulo 2

Modelado

2.1. Análisis del sistema

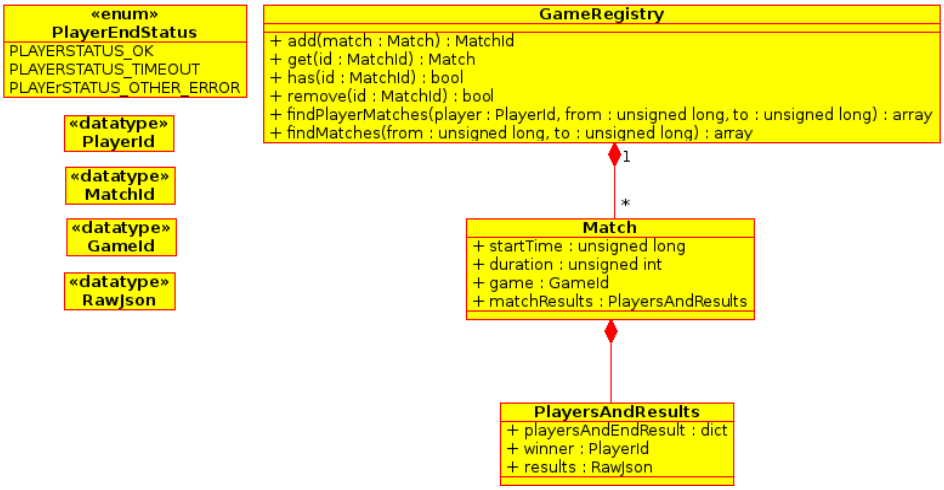

Figura 2.1: Modelo de clases

2.2. Arquitectura del sistema

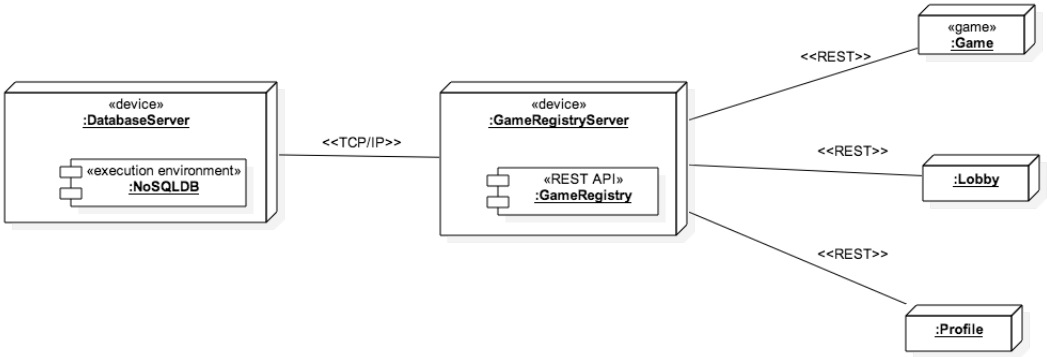

Figura 2.2: Modelo de despliegue del sistema

Capítulo 3

Iteración 3

3.1. Objetivos de iteración

- Integración de Gradle
- Servidor dockerizado
- Estructura inicial del Cliente vertx.

3.2. Gradle

Gradle es un gestor de construcción especialmente indicado para proyectos Java y con soporte para *Groovy*, *Vertx* y *Maven*.

Ha sido integrado mediante el wrapper `gradlew` que permite utilizar Gradle sin instalarlo de forma global en el sistema de desarrollo. La primera vez que se lanza descargará todas las bibliotecas necesarias.

En el archivo **Readme.md** hay información básica sobre como construir el proyecto. Algunos comandos útiles:

- Para **construir** el proyecto:
  `$ ./gradlew clean modZip`
- Para **lanzar** el servidor en la máquina local:
  `$ ./gradlew runMod -i`
- Para lanzar los **tests**:
  `$ ./gradlew clean test`
- Para preparar el proyecto para un **entorno de desarrollo**:
  - Eclipse: `$ ./gradlew eclipse`
  - IDEA: `$ ./gradlew idea`

3.3. Dockerización

Docker es una tecnología que permite utilizar contenedores sobre *Linux* para ejecutar procesos de forma aislada y con un runtime reproducible.

Nuestro proyecto proporciona un archivo **Dockerfile** con las instrucciones necesarias para construir el contenedor de la aplicación. Además, el archivo **Readme.md** contiene información sobre el procedimiento para lanzar el proyecto en *Docker*.

El procedimiento para lanzar el servidor con *Docker* ahora mismo es el siguiente:

- Limpiar y contruir el proyecto:
  `$ ./gradlew clean modZip`

- Construir el contenedor con el servidor:
  `$ docker build -t distributedsystems/gameregistry .`
  (notar el punto final del comando que indica a *Docker* dónde buscar el archivo *Dockerfile* con las instrucciones de construcción). La construcción incluye el resultado de compilación del proyecto por lo que cada vez que este cambie el contenedor debe ser reconstruido.

- Lanzar el contenedor del servidor:
  `$ docker run distributedsystems/gameregistry`

Capítulo 4

Iteración 4

4.1. Objetivos de iteración

- Integración de MongoDB (como contenedor docker)
- Cliente más avanzado. Servidor estructurado en Servicio / Controlador.

4.2. Integración de MongoDB

MongoDB es una base de datos no relacional (*NoSQL*) basada en documentos y que utiliza *JSon* como formato de intercambio de información.

La integración se ha realizado mediante un contenedor *Docker* de forma que tanto el desarrollo como el despliegue es fácilmente reproducible.

El servidor GameRegistry necesita una instancia de *MongoDB* para funcionar. La integración, realizada mediante el módulo *Vertx* llamado *MongoDB persistor*, depende de un archivo de configuración para obtener los datos relativos a la conexión con *MongoDB*. Cambiando los parámetros de este archivo de configuración es posible hacer que el servidor GameRegistry se conecte a una u otra instancia de *MongoDB*.

Hay que tener en cuenta que el contenedor de *MongoDB* utiliza un *Volumen* (un espacio de almacenamiento "externo" al contenedor que persiste de un lanzamiento a otro). A menudo es conveniente configurar el lanzamiento del contenedor de *MongoDB* de forma que dicho volumen corresponda a una carpeta del host:

```
$ docker run -v [local path]:/data/db/ --name mongo-server mongo:3.0.1
```

donde `local path` es la ruta a una carpeta del host que será utilizada por *Docker* para montar la carpeta `/data/db` en el contenedor de *Docker*. De este modo los archivos relativos a la base de datos de *MongoDB* quedan fuera del contenedor y es posible examinarlos de forma externa o hacer copias de seguridad fácilmente.

4.2.1. MongoDB y desarrollo

Para desarrollar el servidor GameRegistry o lanzarlo de forma local fuera de un entorno de contenedores *Docker* tan solo es necesario obtener una instancia de *MongoDB* a la que se pueda conectar y un archivo de configuración con la información necesaria para la conexión.

En el archivo **Readme.md** hay información básica sobre cómo conseguir una instancia de MongoDB utilizando *Docker*, que resulta a menudo más conveniente que instalar una instancia local del mismo.

Hay que tener en cuenta que los contenedores de *Docker* por defecto no exponen los puertos al host. Las instrucciones básicas para lanzar *MongoDB* como contenedor para desarrollo serían:

- Descargar el contenedor (sólo la primera vez):
  `$ docker pull mongo:3.0.1`
- Lanzar la instancia de *MongoDB* exponiendo el puerto al host:
  `$ docker run -P mongo:3.0.1`
- Modificar la configuración de GameRegistry si es necesario en el archivo *conf.json*
- Lanzar el servidor GameRegistry:
  `$ ./gradlew runMod -i`

4.2.2. MongoDB y despliegue

El *Dockerfile* que describe cómo construir el contenedor de nuestro servidor GameRegistry ha sido actualizado para integrar un archivo de configuración separado dedicado al contenedor llamado *conf-docker.json*. Este archivo contiene la configuración necesaria (mínima) para poder conectar con un servidor *MongoDB* que corre en una máquina llamada `mongo-server`. Esto resulta conveniente en un entorno *Docker* enlazando ambos contenedores. Por ejemplo:

- Descargar el contenedor de *MongoDB* (sólo la primera vez):
  `$ docker pull mongo:3.0.1`
- Lanzar la instancia de *MongoDB* sin exponer puertos al host y nombrando la instancia del contenedor:
  `$ docker run --name mongo-server mongo:3.0.1`
- Construir el contenedor para nuestro servidor:
  `$ docker build -t distributedsystems/gameregistry .`
- Lanzar el contenedor de GameRegistry enlazándolo con el de la instancia de *Docker* de forma que compartan la pila de red (y puedan así comunicarse), exponiendo el puerto de GameRegistry en el host:
  `$ docker run -p 8080:8080`
  `--link mongo-server:mongo-server distributedsystems/gameregistry`

4.2.3. Distinción de dominio, controlador y servicio

Dominio

Clases POJO con el mismo esquema de la base de datos.

Controlador
Manejar los requisitos y respuestas y comunicar con el servicio.

Servicio
La lógica de negocio. Almacenar las sessiones en el base de datos.

4.3. Diagrama de paquetes

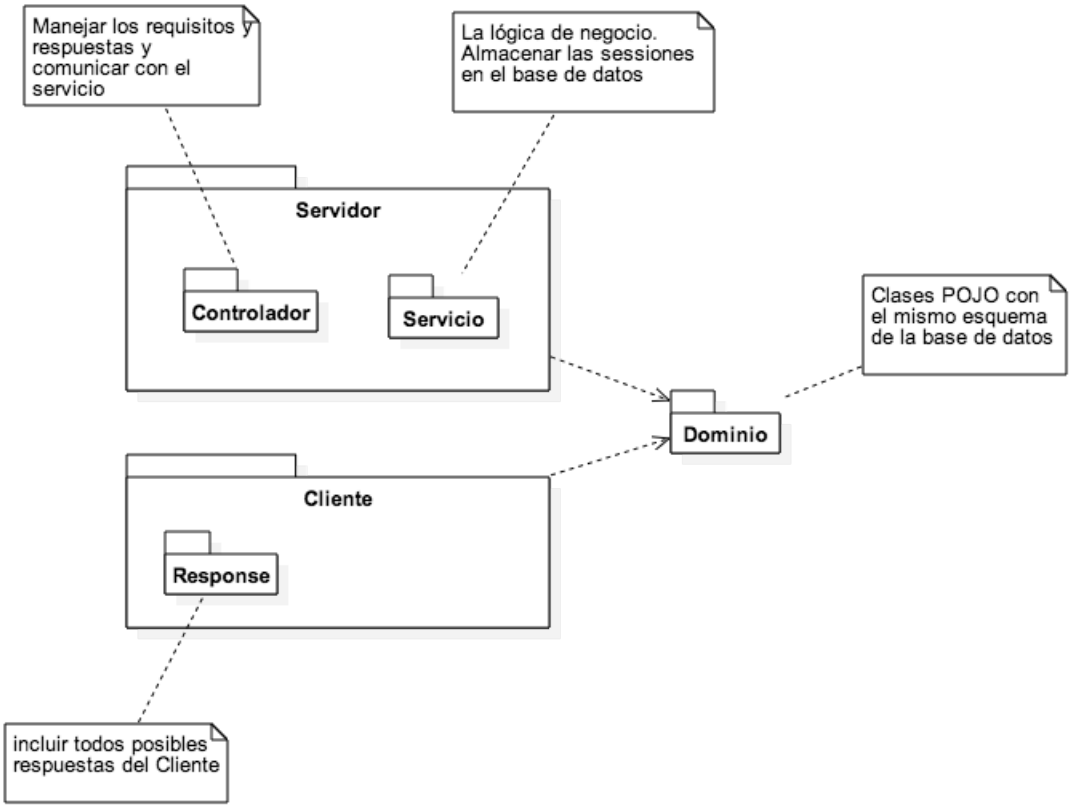

Figura 4.1: Organización de paquetes

4.4. Diagrama de clases

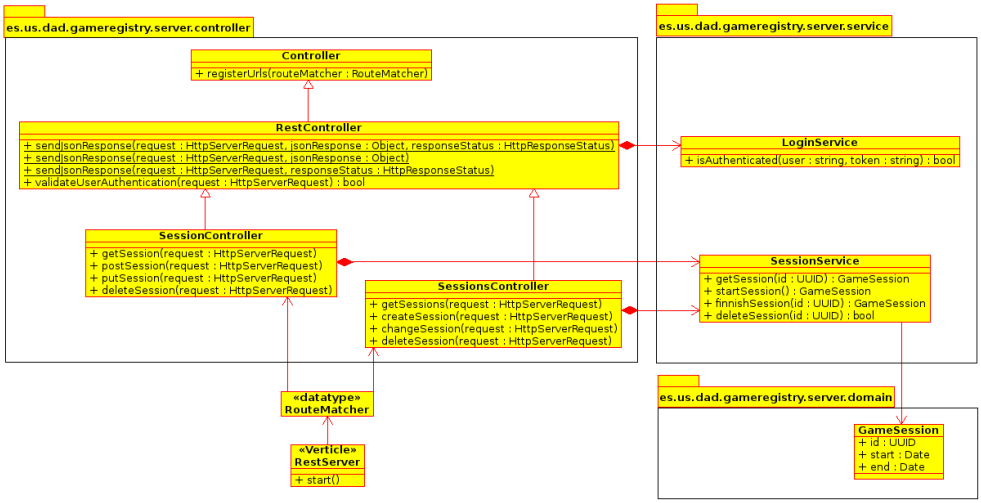

Figura 4.2: Diagrama de clases actualizado (iteración 4)

Capítulo 5

Iteración 5

5.1. Objetivos de iteración

- Implementación inicial de API REST
- Primeros tests
- Despliegue en plataforma Azure
- Investigar integración contínua

5.2. Implementación inicial de API REST

Hemos definido un primer borrador del API REST a implementar por el servidor. No es una especificación exhaustiva o completa pero las partes aún no determinadas de la misma necesitan de información aún no disponible (como por ejemplo las necesidades concretas de filtrado de los clientes del servidor).

En siguientes iteraciones queremos integrar la documentación generada por *Swagger* al final de la memoria. Mientras tanto documento aquí el borrador actual de la especificación de la API y su implementación actual.

5.2.1. API

El API tiene dos puntos de entrada: `sessions` y `session`.

Todos los recursos responden con contenido `application/json` con codificación `UTF-8`. Los códigos de respuesta HTTP serán siempre 2xx para peticiones correctas, 4xx para errores en la petición y 5xx cuando hay un error en el servidor.

Toda petición debe incluir dos cabeceras específicas de *GameRegistry* donde se comunique el identificador de usuario y su token tal y como lo validó/generó el *Servidor de Login* del sistema distribuido. Ambos parámetros serán validados por *GameRegistry* contra el *Servidor de Login*.

/sessions

Representa una colección de sesiones de juego. Métodos HTTP:

- **GET**: Devuelve una colección de sesiones de juego. Debe aceptar parámetros que permitan filtrar la colección. Los parámetros aún no han sido definidos (a la espera de conocer las necesidades de los clientes de *GameRegistry*). Devolverá `200 OK` y una colección de sesiones si tiene éxito.
- **POST**: Añade una nueva sesión de juego a la colección. Devolverá `201 Created` si tiene éxito.
- **PUT**: En una especificación REST el comando PUT significa *reemplazar* el recurso con otro. Dicha operación no debe permitirse en este contexto por lo que *GameRegistry* devolverá ante peticiones como esta el código `405 Method Not Allowed`.
- **DELETE**: Al igual que PUT, el servidor no debe permitir borrar toda la colección de sesiones. Devolverá `405 Method Not Allowed`.

/sessions/:id

Representa una única session de juego determinada por su identificador (`id`). Métodos HTTP:

- **GET**: Devuelve la sesión de juego identificada por `id` con código HTTP `200 OK`. Si no hay sesión con dicho `id` entonces devolverá `404 Not Found`.
- **POST**: En una especificación REST este comando significa *añadir una nueva entrada a la colección* pero la URL `/session/:id` no representa una colección sino una única sesión de juego. Por tanto este método no tiene sentido y el servidor devolverá `405 Method Not Allowed`.
- **PUT**: Reemplaza la sesión de juego por la nueva sesión suministrada en la petición. Devolverá `200 OK` o `404 Not Found` si no hay ninguna sesión con ese identificador.
- **DELETE**: Elimina la sesión de juego con el identificador dado. Devolverá `204 No Content` si tiene éxito o `404 Not Found` si no encuentra la sesión de juego.

Otros

Es muy posible que los requerimientos de los sistemas que dependen de *GameRegistry* obliguen a implementar nuevos puntos de entrada con comportamientos específicos.

5.3. Primeros tests

Hemos comenzado a escribir una batería de tests automáticos que comprueben el correcto funcionamiento del servidor.

Los tests incluidos son:

- `testNotFound`: Busca una sesión inexistente, comprueba resultado.
- `testCreate`: Crea una nueva sessión.

- `testUpdate`: Crea una sesión, sube una nueva versión.
- `testDelete`: Crea una sesión, la borra , comprueba que no existe a continuación.
- `testFindSessions`: Crea una sesión y después la busca y comprueba el contador de resultados.
- `testNotAuthenticated`: Intenta una petición sin especificar usuario y token, comprueba error.

5.4. Despliegue en plataforma Azure

Tras evaluar las opciones de despliegue de *Docker* en *Azure* hemos optado por construir una máquina virtual donde administraremos manualmente una instancia de *Docker* en vez de utilizar la extensión de máquina virtual que implementa *Azure* para *Docker*. Utilizar esta última habría implicado una complejidad añadida por la parafernalia necesaria para mantener certificados de seguridad para poder comunicarnos con la instancia de *Docker*.

En lugar de la extensión de máquina virtual de *Azure* administraremos la máquina virtual manualmente.

5.4.1. Test vm

Para esta iteración hemos creado una máquina virtual cuyo propósito es familiarizarnos con el entorno *Azure*. Para ello los requisitos de esta máquina serían:

- Ser capaz de ejecutar el servidor *GameRegistry* y *MongoDB* como contenedores de *Docker*.
- Ser capaz de construir el contenedor de *GameRegistry* a partir del repositorio SVN del proyecto.

Para ello se ha basado la máquina virtual en una imagen de *Ubuntu 15.04*, que contiene una versión reciente y razonablemente testada de *Docker*. Además hemos añadido el cliente de *Subversion* y el *OpenJDK-8*.

5.4.2. VM de producción

Aún no creada. Sus requisitos son similares pero no iguales a la máquina test. El propósito de esta máquina virtual será el de ejecutar el servidor, evitando la instalación de cualquier pieza de software no necesaria para esa tarea, lo cual en este caso implica no instalar ningún *JDK* ni *Subversion*.

Esto requiere algún método para hacer llegar el contenedor con el servidor GameRegistry a la máquina que no implique a *Subversion* ni el uso de *Java* para compilar el código fuente. Lo resolveremos en la próxima iteración.

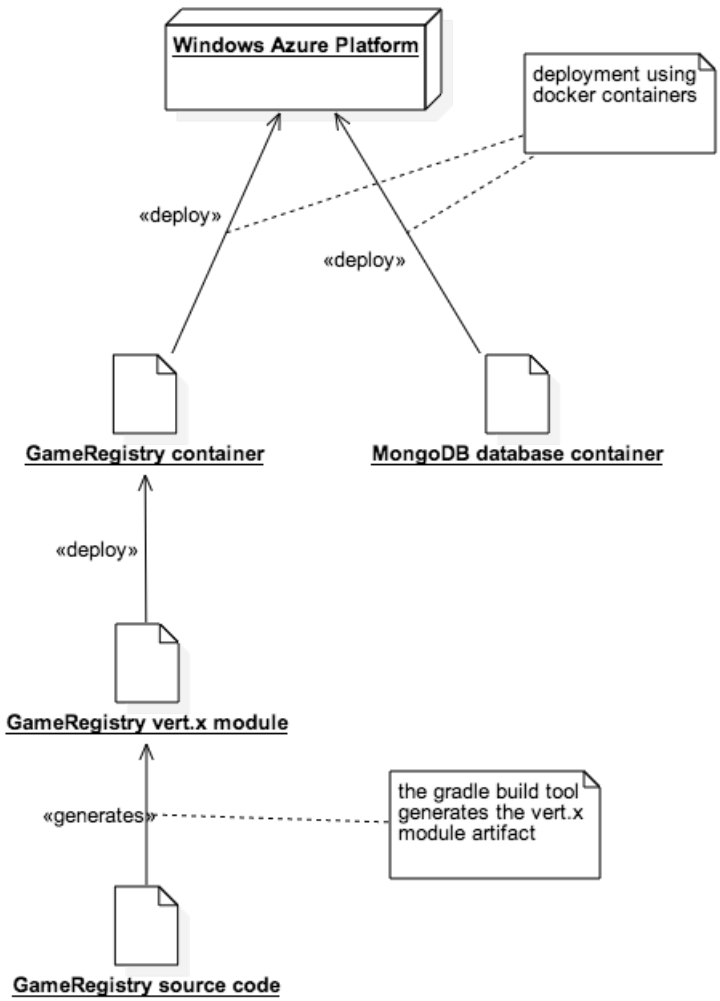

Figura 5.1: Diagrama de despliegue (iteración 5)

5.5. Investigar integración contínua

Algún método de integración contínua sería interesante para el proyecto de forma que los tests sean ejecutados en cada revisión del proyecto. Hay sitios web que ofrecen una instancia gratuita de integración contínua que podrían ser utilizados, como por ejemplo:

- CircleCI (`https://circleci.com/`)
- CodeShip (`https://codeship.com/`)

Estamos aún estudiando la viabilidad y los requisitos impuestos por dichas opciones.

Capítulo 6

Iteracion 6

6.1. Objetivos de iteración

- Integración de documentación de Swagger en la memoria y el proyecto.
- Testing, integración contínua

6.2. Integración de Swagger

6.2.1. Generar la documentación del API REST

Swagger es una especificación que permite documentar servicios REST. Alrededor de esta especificación existen multitud de herramientas y proyectos que se encargan de generar dicha documentación a partir de código o de generar otras formas de consumir la documentación.

Para nuestro caso la generación de la documentación a partir de código, anotaciones, etcétera no es posible. El tipo de herramientas que genera la documentación se apoyan en *frameworks* con estructuras bien definidas para servicios REST, lo que les permite saber cómo extraer esa documentación. *Vert.x* es un *framework* de propósito demasiado general como para que exista una herramienta capaz de extraer del código de un servidor *Vert.x* dicha información.

Por tanto nos limitaremos a escribir un archivo *Json* con la documentación del servicio de forma manual.

6.2.2. Consumir la documentación

Hemos desarrollado dos métodos para hacer que la documentación sea accesible para terceros:

- Html en el servidor
- PDF en la memoria

Html en el servidor

El proyecto *Swagger-ui* consiste en una serie de archivos *html* y *javascript* capaces de leer un archivo de documentación *json* de *Swagger* y mostrarla como *html*, dando incluso la posibilidad de ejecutar peticiones al servicio basándose en dicha documentación. Para que funcione sólo es necesario suministrarle la URL del archivo *Json* con la misma.

Para integrarlo en nuestro servicio hemos escrito un pequeño servidor HTTP que se integra con nuestro servicio a nivel de *RouteMatcher* y sirve el contenido de *Swagger-ui* bajo un subdirectorio del servidor (`/doc/`).

PDF en la memoria

Utilizando una combinación de los paquetes *npm* y *Bootprint* de *node.js* y *PhantomJS* somos capaces de generar primero unos archivos HTML describiendo el servicio REST y después, a partir de estos, un archivo PDF. Todo el proceso esta automatizado a través de tareas de *Gradle*, siendo necesaria de forma manual unicamente la instalación de *PhantomJS* (un motor *Javascript* "sin navegador", *headless*).

Dicho archivo PDF es después incluido en la memoria como anexo.

6.3. Testing, Integración contínua

Tras evaluar el servicio *CircleCI* (`https://circleci.com/`) hemos decidido utilizarlo.

CircleCI sólo nos impone tener como VCS *GIT* de forma accesible. Concretamente esta muy integrado con la plataforma *Git-Hub*, por lo que hemos mudado el desarrollo del proyecto a dicha plataforma. La url actual del repositorio es: `https://github.com/andihit/dad-gamer`

Los datos generales del proyecto han sido actualizados.

Como resultado de la integración, tras el envío de una nueva revisión a *Git-Hub* este automaticamente (a traves de un *Post-build-hook*) avisa a *CircleCI* de la nueva revisión y este a su vez descarga el código, lo compila, ejecuta los tests y guarda el informe.

6.4. UML actualizado

El tamaño del diagrama y la lista de argumentos empieza a ser demasiado como para mostrarlo entero. Obviando los parámetros de las operaciones, sin embargo, la estructura de clases actual resulta en el siguiente diagrama.

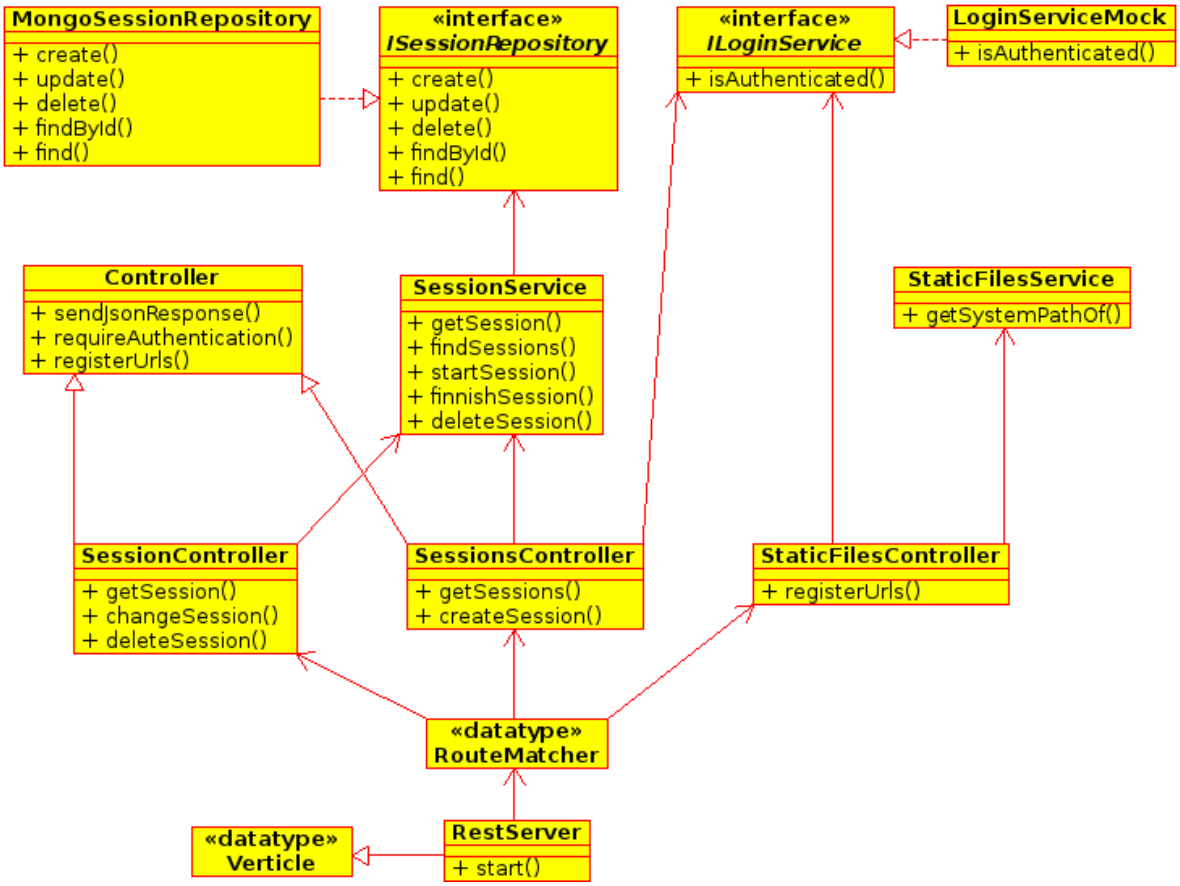

Figura 6.1: Diagrama de clases (iteración 6)

Capítulo 7

Iteracion 7

7.1. Objetivos de iteración

- Finalizar cliente de GameRegistry para vertx.
- Máquina virtual en producción.
- Test sobre *Promises*.

7.2. Finalizar cliente de GameRegistry para vert.x

El cliente asíncrono para el servidor GameRegistry en Java es funcional. En el diseño de la API se ha tenido en cuenta los siguientes factores:

- API con estilo fluído. Debería resultar razonablemente familiar para usuarios de otros objetos Vert.x como `HttpClient` o `DnsClient`.
- Sin dependencias adicionales. Utilizar el cliente no debe obligar al usuario a utilizar más dependencias. Esto es lo que ha motivado la elección del lenguaje (*Java*) y la ausencia en el diseño de *Lambdas* (obligaría a usar *Java 8*) o *Promises*.
- Simpleza de uso. El cliente siempre devolverá tras una petición y de forma asíncrona un objeto *GameRegistryResponse* que contendrá la información del resultado, sea un error (con un código de error y el *Throwable* causante del error incluído) o una colección de sesiones de juego. Sólo requiere, por tanto, un manejador (*Handler*) por petición y es sencillo detectar el resultado de la petición con tan solo comprobar el código de resultado.
- Hemos procurado que sea capaz de detectar e informar de los errores a través de un código de resultado.

Los parámetros de filtrado para peticiones GET a la colección de sesiones de juego siguen sin especificar y por tanto aún no están implementados.

7.3. Máquina virtual en producción

El servidor esta funcionando en la plataforma *Azure* sobre una máquina virtual con el siguiente registro DNS:

`gameregistry.cloudapp.net`

Nuestro servidor escucha peticiones en el puerto **8080**. Para acceder a la documentación de *Swagger* se puede utilizar la siguiente dirección:

`http://gameregistry.cloudapp.net:8080/doc/`

La API vive en la siguiente dirección:

`http://gameregistry.cloudapp.net:8080/api/v1`

7.3.1. La máquina virtual

El sistema de despliegue en estos momentos requiere intervención manual. Los pasos han sido automatizados en gran medida a través de *scripts Bash* pero sigue siendo necesario identificarse en la máquina a través de *ssh* y ejecutar el proceso de despliegue, dividido en 2 o 3 pasos fundamentales (con sus correspondientes *scripts*):

- Construcción del contenedor *Docker* del servidor.
- Si no esta previamente funcionando, lanzar el contenedor *Docker* de *MongoDB*.
- Lanzar el contenedor recién construido del servidor.

7.4. Test sobre *Promises*

7.4.1. Motivación

En una sesión de clase el profesor nos indicó que debíamos *demostrar* que el uso de *Promises* en el servidor no bloqueaba el hilo principal de *Vert.x*.

En nuestro servidor utilizamos *Promises* para hacer más legible y sencilla la programación asíncrona de algunos de los servicios internos (como el servicio de autenticación que debe validar con el *LoginServer* los pares usuario-token de las peticiones realizadas a nuestro servidor). En concreto utilizamos una implementación para *Groovy*:

`com.darylteo.vertx:vertx-promises-groovy:1.2.1`

Dicha biblioteca establece en su `Readme` la siguiente información:

"Esta biblioteca implementa la mayor parte de la especificación *Promises/A+*. Esta basado en *Q* para *Node.js*, que añade facilidades adicionales como `reject` y `finally`. Finalmente, esta construida utilizando la biblioteca *RxJava*."

También menciona la siguiente información:

"Esta biblioteca no se parece a otras implementaciones similares basadas en lenguaje (como *Futures* o *GroovyPromise* en tanto a que esta completamente libre de bloqueos. Esta diseñada para trabajar con plataformas que hacen uso intensivo de *callbacks* asíncronos. Es más, tiene varias clases adicionales que intentan mejorar la seguridad de tipos del uso basado en *Java* a la vez que minimiza el exceso de *verbosidad* a través de la inferencia de *generics*.

Por tanto es de esperar que seamos capaces de demostrarlo con un test razonablemente simple."

7.4.2. El test

La secuencia del test es la siguiente:

- Lanzar desde el test una petición con un *path* específico al servidor.
- El servidor, en respuesta a esta petición al *path* específico, debe utilizar un *Timer* de *Vert.x* para retrasar su respuesta artificialmente un cierto tiempo (20 segundos). Para que el test demuestre que *Promises no bloquea* el *Timer* debe iniciarse desde dentro de un método que devuelve una *promesa* que sólo será cumplida desde el código invocado por el *Timer* al terminar la espera.
- Un cierto tiempo después de la primera petición el test debe lanzar una segunda petición a un *path* del servidor con comportamiento normal de forma que este responda antes de que el tiempo de espera artificial de la primera petición se cumpla.
- Finalmente, el test debe recibir la respuesta a la segunda petición y al cabo de 20 segundos de lanzar la primera petición debe recibir la respuesta a esta.

Implementación

En el lado del servidor el código de inicialización del *RouteMatcher* añade de forma opcional (según el contenido de *conf.json* una ruta adicional con un manejador simple que llama a un nuevo servicio interno llamado *DebugPromiseService*, que a través de una *promesa* devuelve una cadena de texto que se utilizará en la respuesta. Más adelante en esta iteración hay un diagrama de clases actualizado que muestra la estructura general del mismo.

A continuación se muestra el código pertinente del servicio (*Groovy*):

```
Promise<String> doSomething(HttpServerRequest request) {
  final Promise<HttpServerResponse> p = new Promise()

  vertx.setTimer(wait_time * 1000, { time ->
    p.fulfill("<html><body>Waited "
              + wait_time
              + " seconds to give this answer.</body></html>")
  })

  return p;
}
```

El test tiene la siguiente implementación:

```
def testDebugPromiseIsNotBlocking() {
  HttpClient client = vertx.createHttpClient().setPort(8080)
```

```
  HttpClient client2 = vertx.createHttpClient().setPort(8080)
  String path = "/debug_promise"
  List<Integer> order = []

  // This gets the first request going
  container.logger.info("#1: Launching first request!")
  order.add(1)

  client.getNow(path, { response ->
    // This code will execute around 20 secs after the getNow call was performed
    container.logger.info("#4: First request answered.")
    order.add(4)

    assertEquals([1,2,3,4], order)
    testComplete()
  })

  // Wait for 2 seconds
  vertx.setTimer(2 * 1000, {long time ->
    // And then do another request that should end before the first one.
    container.logger.info("#2: Launching second request!")
    order.add(2)

    client2.getNow("/doc", { response ->
      container.logger.info("#3: Second request answered.")
      order.add(3)
    })
  })
}
```

Cabe destacar que la implementación de `HttpClient` no es capaz de realizar dos peticiones simultaneas al mismo host en el mismo *verticle*, o eso parece. Resultadaba confuso porque los resultados del test parecían sugerir que efectivamente el servidor estaba bloqueando tras la primera petición pero realmente era el cliente Http el problema. Con dos instancias el test pasa sin problemas.

7.5. Diagrama UML actualizado

Capítulo 8

Documentación de API GameRegister

GameRegistry API

us.es ETSII Computer Engineering, Distributed Systems course, a simple information system that stores game sessions.

Summary

Path	Method	Description
/sessions	GET	Returns a collection of game sessions.
	POST	Adds a new session to the game session collection.
/sessions/{id}	DELETE	Removes a game session from the server.
	GET	Returns a game session given an id.
	PUT	Replaces the game session identified by 'id' with the included game session.

Paths

GET /sessions

Returns a collection of game sessions.

DESCRIPTION

This resource represents the collection of game sessions. It accepts query arguments for filtering (not implemented or defined as of yet, waiting for the games groups to 'express' their needs).

REQUEST PARAMETERS

Name	Description	Type	Data type
gameregistry-user	The used identifier used in the Login Server.	header	string

gameregistry-token	The token related to the user specified in 'gameregistry-user'.	header	string
user	Example filter parameter. TODO.	query	string
start	Example filter parameter. TODO.	query	number
end	Example filter parameter. TODO.	query	number

RESPONSE BODY

Code	Description
200	A collection of game sessions. **ITEMS** GameSession

POST /sessions Adds a new session to the game session collection.

DESCRIPTION

The new session will have the end date open. To close it see PUT /sessions/{id}.

REQUEST PARAMETERS

Name	Description	Type	Data type

gameregistry-user	The used identifier used in the Login Server.	header	string
gameregistry-token	The token related to the user specified in 'gameregistry-user'.	header	string
game session	The game session to add to the system.	body	object

RESPONSE BODY

Code	Description
201	The game session was successfully added to the server. GameSession

DELETE /sessions/{id} Removes a game session from the server.

DESCRIPTION

REQUEST PARAMETERS

Name	Description	Type	Data type
gameregistry-user	The used identifier used in the Login Server.	header	string

gameregistry-token	The token related to the user specified in 'gameregistry-user'.	header	string
id	The game session identifier.	path	number

RESPONSE BODY

Code	Description
204	The game session was removed.
404	No game session with the given 'id' has been found.

GET /sessions/{id}

Returns a game session given an id.

DESCRIPTION

REQUEST PARAMETERS

Name	Description	Type	Data type
gameregistry-user	The used identifier used in the Login Server.	header	string
gameregistry-token	The token related to the user specified in 'gameregistry-user'.	header	string
id	The game session identifier.	path	number

RESPONSE BODY

Code	Description
200	The game session is returned. GameSession

Replaces the game session identified by 'id' with the included game session.

PUT /sessions/{id}

DESCRIPTION

REQUEST PARAMETERS

Name	Description	Type	Data type
gameregistry-user	The used identifier used in the Login Server.	header	string
gameregistry-token	The token related to the user specified in 'gameregistry-user'.	header	string
id	The game session identifier.	path	number

RESPONSE BODY

Code	Description

200	The game session was replaced. GameSession
404	If the game session was not found.

Definitions

GameSession: *object*

PROPERTIES

id: *string*
an UUID that uniquely identifies the game session.

user: *string*
User identifier.

game: *string*
Game name or identifier.

start: *string*
Start date of the game session.

end: *string*
End date of the game session.