

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 continua	15

6. Iteracion 6	16
6.1. Objetivos de iteración	16
7. Iteracion 7	17
7.1. Objetivos de iteración	17

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

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: recibir el nombre de un jugador

1.3. Direcciones de descarga y planificación

Código fuente	https://repositorio.informatica.us.es/svn/lq3vqrtzfnh2nx9yhp
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

Estudiante	Iteración								Total	Pond.
	1	2	3	4	5	6	7	Final		
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

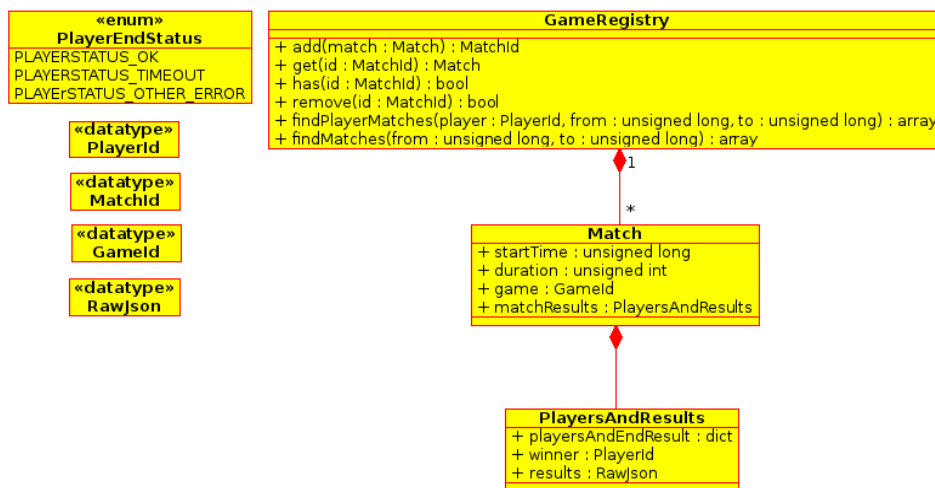


Figura 2.1: Modelo de clases

2.2. Arquitectura del sistema

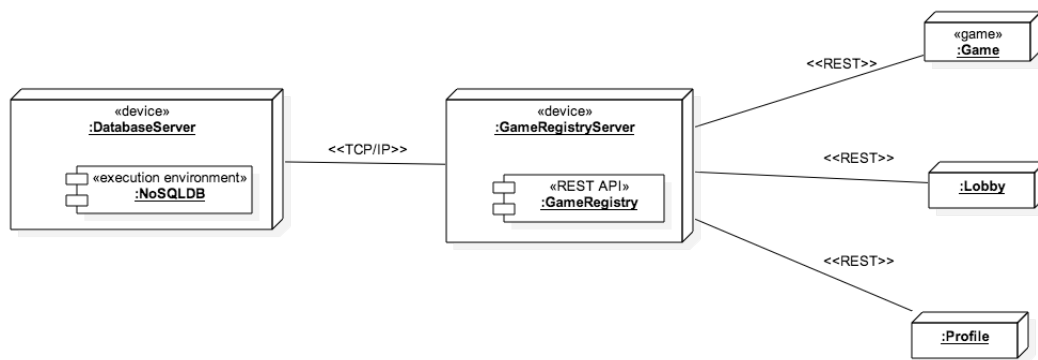


Figura 2.2: Modelo de despliegue del sistema

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 *Vertex* 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 sesiones en el base de datos.

4.3. Diagrama de paquetes

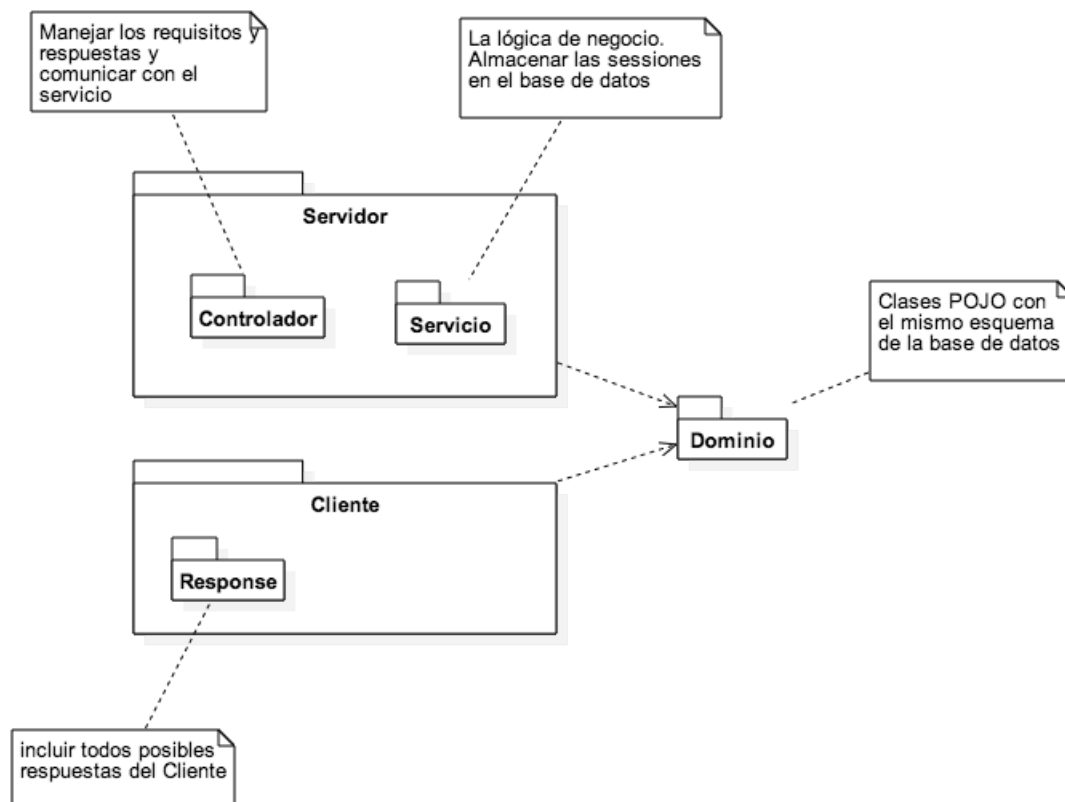


Figura 4.1: Organización de paquetes

4.4. Diagrama de clases

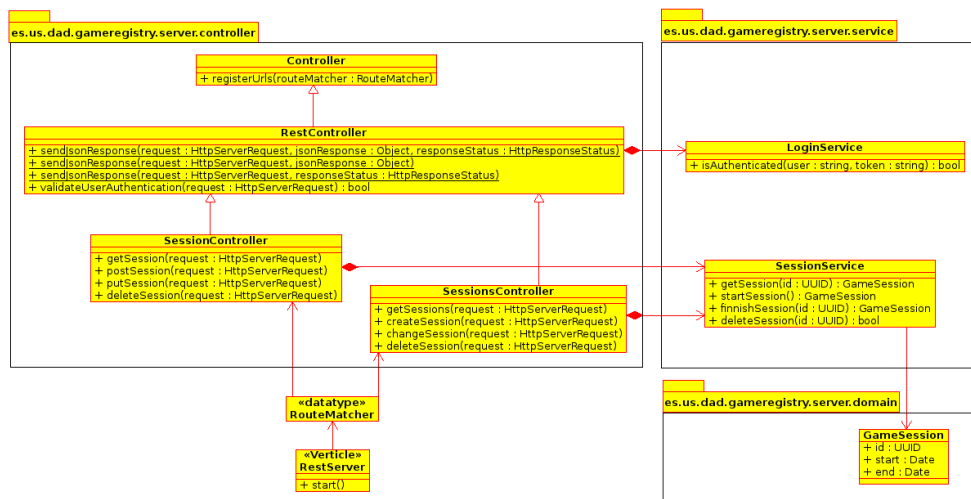


Figura 4.2: Diagrama de clases actualizado (iteración 4)

Capítulo 5

Iteración 5

5.1. Objetivos de iteración

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

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.

`/session/: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 sesión.

- `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.

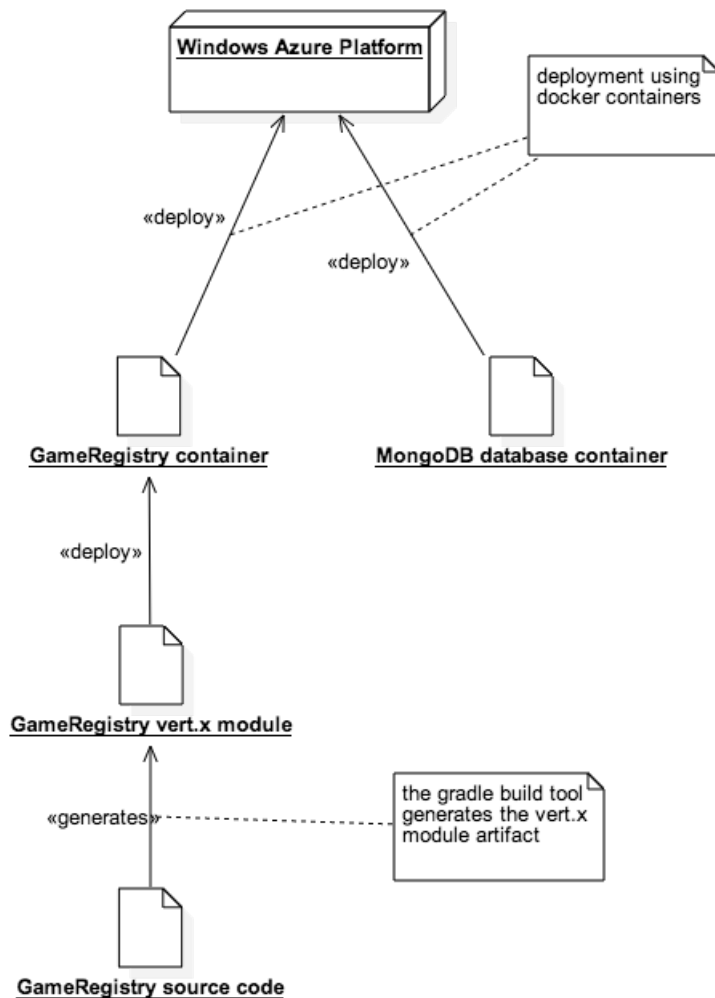


Figura 5.1: Diagrama de despliegue (iteración 5)

5.5. Investigar integración continua

Algún método de integración continua 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 continua 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.
- Reestructuración de los contenedores para hacer más sencillo el despliegue en Azure
- Máquina virtual de producción, procedimiento de despliegue determinado.
- Testing.

Capítulo 7

Iteracion 7

7.1. Objetivos de iteración

Subir a repositorio Maven.