Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Generación de datos en bases Mongo para mejorar el testing automático

Tesis de Licenciatura en Ciencias de la Computación

Hernán Ariel Ghianni

Director: Juan Pablo Galeotti

Buenos Aires, 2024

GENERACIÓN DE DATOS EN BASES MONGO PARA MEJORAR EL TESTING AUTOMÁTICO

Es común que los microservicios interactúen con una base de datos. A la hora de generar tests de caja blanca, es necesario tener en cuenta el estado de la base de datos para lograr una cobertura más amplia y descubrir nuevas fallas. En este trabajo, se presentan técnicas para mejorar el testing de software basadas en búsqueda para microservicios que utilizan bases de datos MongoDB, considerando su estado. Además, se permite la inserción de datos directamente desde los tests. Esto resulta particularmente útil cuando es difícil o lleva mucho tiempo generar la secuencia correcta de eventos para poner la base de datos en el estado interesante para ser ejercitado. También es beneficioso cuando se trata de microservicios de "solo lectura". Esta técnica está implementada como una extensión de EvoMaster, una herramienta de código abierto para generar tests automáticos para API REST. Los experimentos realizados en 5 APIs REST mostraron mejoras significativas en el cubrimiento de líneas de código (hasta +30 %).

Palabras claves: Mongo, base de datos, generación de test automática, heurísticas, API REST.

MONGO DATA GENERATION TO ENHANCE SEARCH-BASED SYSTEM TESTING

It is common for applications to interact with a database. When generating white-box tests, it is necessary to consider the state of the database to achieve higher coverage and discover new faults. In this work, techniques are presented to enhance search-based software testing for applications using MongoDB databases, taking their state into account. Additionally, insertion of data directly from test cases is enabled. This is particularly useful when it is difficult or time-consuming to generate the correct sequence of events to set the database in the appropriate state. It is also beneficial when dealing with read-only"microservices. This technique is implemented as an extension of EvoMaster, an open-source tool for generating automated tests for RESTful APIs. The experiments on five RESTful APIs showed significant improvements in code coverage (up to +30 %).

Keywords: Mongo, database, automated test generation, heuristics, RESTful API.

A mi familia por apoyarme y soportarme desde el primer hasta el último día.
A mis amigos por hacer el camino más llevadero.

Índice general

1.. Introducción 1

2.. Contexto 3
- 2.1. API REST 3
- 2.2. MongoDB 4
- 2.3. Algoritmos evolutivos 6
- 2.4. Algoritmo MIO 6
- 2.5. EvoMaster 7

3.. Motivación 9

4.. Heurísticas para Mongo 13
- 4.1. Captura de comandos de Mongo 13
- 4.2. Cálculo de distancia 13
- 4.3. Integración de heurísticas al Fitness 16

5.. Generación de datos en bases Mongo 19

6.. Experimentación 23

7.. Conclusión 25

1. INTRODUCCIÓN

En la actualidad, las microservicios juegan un papel crucial en nuestra vida cotidiana. Sin embargo, para un funcionamiento eficiente y sin errores, los tests resultan esenciales. La automatización del testing de microservicios puede resultar desafiante debido a su complejidad. Por lo tanto, es común que los enfoques automatizados se centren principalmente en tests de *caja negra*, los cuales se abstraen del funcionamiento interno del microservicio.

Las herramientas de testing basadas en búsqueda buscan generar tests de *caja blanca*, donde se analiza e instrumenta el código del sistema bajo análisis (SUT por sus siglas en ingles). Al recolectar información y aplicar heurísticas, estas herramientas pueden generar datos más sólidos para lograr una cobertura de código mas extensa y detectar posibles fallas de manera mucho mas precisa. Sin embargo, las microservicios a menudo interactúan con sistemas externos, como bases de datos. El comportamiento del SUT puede depender del estado actual de la base de datos, introduciendo desafíos adicionales. Por ejemplo, las heurísticas basadas en bytecode del SUT pueden no ofrecer orientación para insertar los datos adecuados en la base de datos.

En este trabajo se propone adaptar una técnica desarrollada para bases SQL [1] para aplicarla con bases de datos Mongo [2]. La propuesta implica interceptar cada consulta `FIND` realizada por el SUT y a través de heurísticas optimizar la generación de datos para guiar la búsqueda hacia individuos que retornen conjuntos de datos no vacíos.

No obstante, poblar la base con datos que permitan un comportamiento interesante del microservicio puede no ser sencillo. Por ejemplo, en casos donde el microservicio es "de solo lectura" o la secuencia de operaciones HTTP para poblar la base de datos es complicada, se puede recurrir a la inserción directa de datos Mongo desde los tests.

La adopción de esta técnica plantea desafíos adicionales, ya que un test no solo implica operaciones en el SUT sino también inserciones Mongo directas en la base de datos.

Las nuevas técnicas se implementan como una extensión de EvoMaster, una herramienta de código abierto diseñada para generar tests de sistema para API REST. Esta extensión permite la generación de tests en formato JUnit, los cuales son autocontenidos y pueden ejecutarse directamente desde un IDE (por ejemplo, IntelliJ y Eclipse) o desde herramientas de gestión de proyectos (por ejemplo, Maven y Gradle). Los resultados de esta tesis representan un nuevo avance en la automatización de testing de microservicios complejos.

2. CONTEXTO

2.1. API REST

En nuestra vida cotidiana, es común encontrar conexiones entre tecnologías aparentemente independientes. Por ejemplo, al controlar dispositivos mediante asistentes de voz [3]. Esto es posible gracias a las Interfaces de Programación de Aplicaciones (API) [4] [5], que operan como un mecanismo que permite que una aplicación o servicio acceda a los recursos de otro servicio o aplicación. En este contexto, el cliente emite solicitudes al servidor para llevar a cabo diversas operaciones, como la creación, la lectura, la actualización y la eliminación de datos.

La importancia de las APIs en la conectividad y la innovación en la era digital no puede ser subestimada. Estas juegan un papel fundamental al proporcionar la infraestructura necesaria para la integración de sistemas, fomentando la creación de ecosistemas digitales interconectados que impulsan la transformación digital.

La transferencia de estado representacional (REST) [6] es una arquitectura de software que impone condiciones sobre cómo debe funcionar una API. Las APIs que siguen el estilo arquitectónico de REST se llaman API REST [7].

Para que una API se considere REST, debe cumplir los siguientes criterios:

1. **Arquitectura Cliente-Servidor:** La arquitectura debe seguir el modelo cliente-servidor, donde el cliente y el servidor son entidades separadas que se comunican a través de una interfaz (la API).

2. **Sin Estado (Stateless):** Cada solicitud del cliente al servidor es independiente, debe contener toda la información necesaria para entender y procesar la solicitud. El servidor no debe almacenar ningún estado sobre el cliente entre las solicitudes.

3. **Interfaz Uniforme:** La interfaz de la API debe ser uniforme y consistente. Esto incluye la identificación de recursos a través de URIs (Uniform Resource Identifiers), la manipulación de recursos a través de representaciones, y la navegación entre recursos utilizando hipervínculos.

4. **Representación de Recursos:** Los recursos, que pueden ser datos o servicios, deben ser representados de manera clara y consistente. Las representaciones pueden ser en diferentes formatos como JSON o XML.

5. **Sin Condiciones (Cacheable):** Las respuestas del servidor deben indicar si la información puede ser almacenada en caché por el cliente. Esto mejora la eficiencia y la escalabilidad de la API.

6. **Sistema de Capas (Layered System):** La arquitectura puede ser compuesta por capas, donde cada capa tiene una funcionalidad específica. Cada capa debe ser independiente y no debe conocer la implementación de las capas adyacentes.

En la figura 2.1 se puede observar un esquema que representa los componentes de una arquitectura de API REST. A continuación se describe como podría funcionar una solicitud de un usuario para obtener una lista de personas:

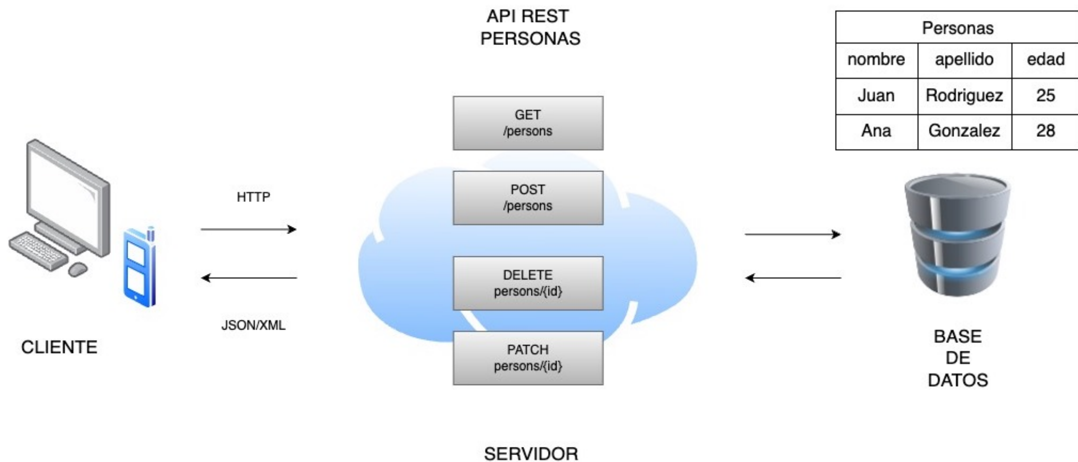

Fig. 2.1: Esquema API Rest.

1. El usuario inicia una solicitud *GET* al endpoint */persons*.
2. La solicitud se envía al servidor que aloja la API.
3. El servidor recibe la solicitud y la procesa utilizando la lógica definida para el endpoint */persons*.
4. La API accede a la base de datos para recuperar la información sobre las personas.
5. La API genera una respuesta que incluye los datos solicitados, en formato adecuado (por ejemplo, JSON o XML).
6. La respuesta se envía de vuelta al cliente que realizó la solicitud.

2.2. MongoDB

MongoDB [2] es una base de datos NoSQL que almacena datos en documentos Binary JSON (BSON) [8], estructuras flexibles similares a JSON. A diferencia del modelo relacional, los documentos se agrupan en colecciones en lugar de de tablas y no requieren un esquema fijo. Cada documento puede tener su propia estructura y no todos los documentos en una colección deben tener las mismas propiedades. Los documentos se identifican utilizando el campo `_id`, que actúa como clave primaria.

Se destaca por ser escalable y distribuido, lo que permite agregar más servidores para manejar grandes volúmenes de datos o mejorar el rendimiento de las consultas. También admite la replicación, lo que garantiza la disponibilidad y tolerancia a fallos al tener copias de los datos en varios servidores.

Las operaciones disponibles en MongoDB incluyen inserción de nuevos documentos, búsqueda, actualización y eliminación de documentos existentes. Además, MongoDB ofrece operaciones de agregación, indexación y soporte para consultas geoespaciales.

En el contexto de esta tesis, es importante conocer como se realizan y funcionan las consultas. El objetivo de una búsqueda en Mongo es encontrar los documentos que cumplen un determinado criterio dentro de una colección. Para realizar una búsqueda se usan

```
{
 "_id": ObjectId("603e06b5a19f2802b8442210"),
 "nombre": "Juan",
 "apellido": "Rodriguez"
 "edad": 25,
},
{
 "_id": ObjectId("603e06b5a19f2802b8442211"),
 "nombre": "Ana",
 "apellido": "Gonzalez"
 "edad": 28,
}
```

Fig. 2.2: Ejemplo de documentos BSON.

los distintos métodos *FIND* disponibles. Estos métodos permiten utilizar operadores de consulta y proyección. Los primeros establecen criterios para seleccionar únicamente los documentos que los cumplan. Los operadores de proyección, en cambio, determinan los campos de los documentos que se incluirán en el resultado.

A modo de ejemplo se presenta una operación de búsqueda que utiliza el operador *$eq* para obtener los documentos cuyo campo edad tiene el valor 25:

```
db.miColeccion.find({ edad: { $eq: 25 } });
```

Fig. 2.3: Consulta con operador de selección.

```
{
  "_id": ObjectId("603e06b5a19f2802b8442210"),
  "nombre": "Juan",
  "apellido": "Rodriguez"
  "edad": 25
}
```

Fig. 2.4: Resultado de la búsqueda.

Si al ejemplo anterior se le agrega un operador de proyección, se pueden filtrar los campos de los documentos del resultado:

```
db.miColeccion.find({ edad: { $eq: 25 } }, { nombre: 1});
```

Fig. 2.5: Consulta con operador de selección y proyección.

```
{
  "_id": ObjectId("603e06b5a19f2802b8442210"),
  "nombre": "Juan"
}
```

Fig. 2.6: Resultado de la búsqueda con proyección.

2.3. Algoritmos evolutivos

Los algoritmos evolutivos son una categoría de algoritmos de optimización inspirados en la teoría de la evolución biológica [9]. Un algoritmo evolutivo típico implica la creación de una población inicial de individuos, cada uno representando una posible solución al problema. Estos individuos se evalúan utilizando una función de *fitness* que mide su desempeño en relación con el objetivo deseado. Los individuos de mejor rendimiento tienen más probabilidades de sobrevivir y reproducirse, contribuyendo así a la evolución de la población hacia soluciones más óptimas. Luego, los individuos se someten a un proceso de mutación, que genera nuevas soluciones modificando las características de los individuos existentes. Al repetir estos pasos, la población evoluciona, y se espera que las soluciones más aptas se vuelvan más comunes. Con el tiempo, idealmente, el algoritmo converge hacia soluciones óptimas.

2.4. Algoritmo MIO

El algoritmo Many Independent Objective (MIO) [10], es un algoritmo evolutivo cuyo objetivo es generar una test suite de manera automática. Por cada objetivo de testing (por ejemplo, cubrir un branch de un `if`) mantiene una población compuesta por tests. Cada vez que se alcanza un nuevo objetivo y no se cubre (por ejemplo, se llega a un `if` pero solo se cubre la rama false, nunca la rama true), se crea una nueva población.

En la figura 2.7 se presenta el pseudocódigo del algoritmo. Conserva un archivo con tests que, inicialmente, se encuentra vacío. En cada iteración, si no se ha cumplido la condición de parada, se procede a generar un nuevo test en la línea 6 de forma aleatoria o seleccionar uno existente para someterlo a un proceso de mutación en las líneas 8-9, como cambiar un carácter de un `string`.

Para cada objetivo que puede alcanzar el test, se evalúa si efectivamente se ha alcanzado en la línea 12. En caso afirmativo, el test se guarda en el archivo y se elimina el resto de la población en las líneas 13-14. De lo contrario, se agrega el test a la población correspondiente al objetivo en la línea 16. Cuando se supera el número máximo de tests en alguna población, se descarta el que obtenga el peor resultado según la función de *fitness* en las líneas 17-18.

Como resultado final, se obtiene una suite compuesta por los mejores tests de cada población.

Algoritmo Many Independent Objective (MIO)

```
Entrada: Condición de parada C, Tamaño máximo de población n, Función de fitness δ
Salida: Archivo con los mejores individuos A
 1: P ← ConjuntoDePoblacionesVacias()
 2: A ← {}
 3: while ¬C do
 4:     generarIndividuoAleatorio ← ObtenerBooleanoAleatorio()
 5:     if generarIndividuoAleatorio then
 6:         i ← IndividuoAleatorio()
 7:     else
 8:         i ← SeleccionarIndividuo(P)
 9:         i ← Mutar(i)
10:     end if
11:     for all k ∈ ObjetivosAlcanzables(i) do
12:         if EsObjetivoCubierto(k) then
13:             AgregarIndividuoAlArchivo(A, i)
14:             P ← P \ {P_k}
15:         else
16:             P_k ← P_k ∪ {i}
17:             if |P_k| > n then
18:                 EliminarPeorTest(P_k, δ)
19:             end if
20:         end if
21:     end for
22: end while
23: return A
```

Fig. 2.7: Algoritmo Many Independent Objective (MIO).

2.5. EvoMaster

EvoMaster [11] es una herramienta que tiene como finalidad generar casos de test a nivel del sistema para APIs REST. Para lograrlo emplea algoritmos evolutivos, como por ejemplo MIO [12]. Se divide en dos partes principales: (1) un proceso central responsable de la interfaz de línea de comandos, los algoritmos de búsqueda y la generación de casos de test; y (2) una biblioteca de controladores que el usuario necesita para escribir clases de configuración manuales que indiquen a EvoMaster cómo iniciar, reiniciar y detener el sistema bajo análisis (SUT). Dicha biblioteca de controladores también se encarga de instrumentar automáticamente el SUT al inicio para recopilar heurísticas.

EvoMaster genera casos de test en formato JUnit. Estos tests son autocontenidos, ya que utilizan el controlador para administrar el SUT cuando sea necesario. Estos tests, se pueden iniciar directamente desde un entorno de desarrollo integrado (IDE) como IntelliJ o como parte de un sistema de compilación (por ejemplo, Maven y Gradle). Cada test generado se compone de una serie de llamadas HTTP al SUT, utilizando la popular biblioteca RestAssured [13].

3. MOTIVACIÓN

En esta sección, se presenta un ejemplo sencillo para comprender la finalidad de la tesis. El ejemplo consta de una API REST escrita en Java utilizando el framework SpringBoot [14] y accediendo a una base de datos Mongo. Incluye una clase llamada `PersonController` y una interfaz `PersonRepository`. Se omiten archivos adicionales por simplicidad. El objetivo es utilizar EvoMaster para generar tests con la mayor cobertura posible.

```
@RestController
@RequestMapping("/api/persons")
public class PersonController {

    @Autowired
    private PersonRepository personRepository;

    @GetMapping("/listHernanGhianni")
    public ResponseEntity<List<Person>> get {
        List<Person> persons =
            personRepository.findByNameAndLastName("Hernan","Ghianni")

        if (persons.isEmpty()) {
            return new ResponseEntity.status(400).build();
        } else {
            return new ResponseEntity.status(200).build();
        }
    }

    @PostMapping("/{name}/{lastName}")
    public Person post(
        @PathVariable String name,
        @PathVariable String lastName) {

        Person person = new Person(name, lastName);
        return personRepository.save(person);
    }
}

public interface PersonRepository extends MongoRepository<Person, String> {
    List<Person> findByNameAndLastName(String name, String lastName);
}
```

Fig. 3.1: Ejemplo de una Web API en Java SpringBoot que utiliza una base de datos Mongo.

En el ejemplo, se permiten dos operaciones: `POST` y `GET`. La operación `POST` agrega una instancia de la clase `Person` al repositorio con los valores de `name` y `lastName` que se pasan como parámetros. La operación `GET` consulta la base de datos y busca todas las instancias de `Person` que tengan `name = Hernan` y `lastName = Ghianni`. Si existe al menos una instancia en el repositorio que cumpla esa condición, se devuelve un código de estado `200`; de lo contrario, `400`.

Es importante destacar que en este ejemplo no se realizan consultas directas a la base de datos. El controlador REST, `PersonController`, se encarga de ello al analizar los nombres

de los métodos en la interfaz. Para cada uno de estos métodos, Spring crea automáticamente una implementación concreta que realiza un comando Mongo basado en el nombre del método. Por ejemplo, el método `findByNameAndLastName` generará automáticamente un código con el comando `FIND` de Mongo correspondiente:

```
persons.find({
  $and: [
    { name: { $eq: "Hernan" } },
    { lastName: { $eq: "Ghianni" } }
  ]
})
```

A su vez, el repositorio funciona también como abstracción; internamente utiliza una colección de Mongo cuyos documentos tendrán la misma estructura que las instancias de Person. La ejecución del código `personRepository.findByNameAndLastName("Hernan", "Ghianni")` es equivalente a realizar un `FIND` en la colección `persons`.

Para cumplir con el objetivo, el caso más complejo consiste en crear un test cuya respuesta sea el código `200` en el endpoint `GET`, ya que esto depende del estado específico de la base de datos. En el ejemplo, se utiliza el método `findByNameAndLastName` para determinar si la colección contiene un documento con ciertas características. Esta operación es la que determina el resultado del método. Para cubrir el caso donde el código de respuesta es 200, la operación debe devolver un conjunto no vacío. EvoMaster debe generar un input específico mediante el método `POST` para que esto suceda. El conflicto radica en cómo EvoMaster puede determinar el input concreto que debe generar. Este problema ya ha sido resuelto para bases de datos `SQL` en EvoMaster [1].

El método `findByNameAndLastName` determina si la colección contiene o no un documento con el nombre `Hernan` y el apellido `Ghianni`. En SQL, esto se traduce como un `SELECT`. La extensión almacena este tipo de consultas y calcula un valor heurístico que representa qué tan cerca estuvo de encontrar un valor en la colección que satisficiera esa consulta. Por ejemplo, si ya hay una persona guardada con ese nombre y apellido, el valor será 0. De esta manera, sabiendo el último input generado y la distancia al valor esperado, cuando EvoMaster genere un input para insertar en la colección mediante el método `POST`, podrá utilizar esos valores como guía.

La técnica propuesta en esta tesis consiste en analizar los comandos equivalentes al SELECT en MongoDB, es decir, los `FIND` de MongoDB que se ejecuten, y calcular heurísticas a partir de estos.

```
@Test
    fun test() {

        given().accept("*/*")
                .post("${baseUrlOfSut}/persons/Hernan/Ghianni")
                .then()
                .statusCode(200)


        given().accept("*/*")
                .get("${baseUrlOfSut}/persons/listHernanGhianni")
                .then()
                .statusCode(200)

    }
```

Fig. 3.2: Ejemplo de un JUnit test generado para la API de la Figura 3.1.

Pero consideremos un escenario más desafiante en el que solo se define un endpoint `GET` y por lo tanto no hay forma de, utilizando los endpoints provistos, insertar documentos en la base de datos. En este escenario, la única forma de lograr una cobertura completa en el método `GET` es insertando documentos directamente en la base desde los tests. A partir del trabajo realizado se extiende EvoMaster de manera de poder generar documentos en una colección de Mongo como parte de la búsqueda, junto con la generación de las entradas para el SUT. La Figura 3.2 muestra un ejemplo de test JUnit generado con la extensión. Este test primero agregará un documento con los valores `Hernan` y `Ghianni` en la colección, y luego realizará un `GET` con los mismos valores. Hemos desarrollado nuestra propia biblioteca con un conjunto específico de comandos diseñados para interactuar con MongoDB. Esta biblioteca se utilizará para realizar las inserciones en la base de datos, integrándose de manera automática al generar las tests. Sin esta extensión, sería imposible para una herramienta como EvoMaster lograr una cobertura completa.

```
@Test
fun test() {
        val insertions = mongo().insertInto("persons", "person")
            .d("{\"name\":\"Hernan\",\"lastName\":\"Ghianni\"}")
            .dtos()
        controller.execInsertionsIntoMongoDatabase(insertions)

        given().accept("*/*")
                .get("${baseUrlOfSut}/persons/listHernanGhianni")
                .then()
                .statusCode(200)
    }
```

Fig. 3.3: Ejemplo de un JUnit test generado para la versión sin POST de la API de la Figura 3.1

4. HEURÍSTICAS PARA MONGO

Durante la ejecución de una acción en el SUT (por ejemplo, una llamada HTTP en un servicio web), uno o más comandos de MongoDB pueden ser ejecutados. Los caminos de ejecución en el SUT pueden depender del resultado de ese comando, como se muestra en la sección 3. La ejecución de dichos comandos puede depender, a su vez, de los datos almacenados en la base de datos, los cuales podrían haber sido modificados por acciones anteriores en el SUT (por ejemplo, operaciones POST/PUT en una API REST) u otros servicios externos que trabajan en la misma base de datos. Por lo tanto, resulta relevante conocer el estado de la base de datos y determinar la información necesaria para cubrir los distintos caminos de ejecución.

EvoMaster utiliza el algoritmo evolutivo MIO [12] para generar los tests. Para mejorar la generación cuando el SUT utiliza una base de datos MongoDB, se propone analizar los comandos `FIND` y desarrollar heurísticas basadas en ellos para orientar la búsqueda. Para incorporar estos nuevos valores en el algoritmo, se requiere modificar la función de *fitness*.

Se plantea el siguiente esquema:

- Cada vez que el SUT realiza una operación `FIND` en una colección para la cual no se devuelven documentos, se calcula un valor heurístico que aproxima qué tan cerca estuvo de encontrar al menos una coincidencia.
- Estos valores heurísticos se integran en la búsqueda (*fitness*) como objetivos secundarios en la optimización.

4.1. Captura de comandos de Mongo

Para calcular los valores heurísticos, es necesario monitorear los comandos de MongoDB que se ejecutan. EvoMaster posee la capacidad de instrumentar el código ejecutado y, a través de la configuración de clases específicas, almacenar información relacionada con la ejecución de un método. MongoDB provee un driver para Java [15] que será el encargado de ejecutar los comandos sobre las colecciones. En consecuencia, para lograr el objetivo se creó una nueva clase con el propósito de capturar y almacenar específicamente la información de las consultas `FIND` realizadas.

4.2. Cálculo de distancia

A partir de cada comando `FIND` que no devolvió documentos, se calcula un valor heurístico con el fin de orientar la búsqueda hacia la obtención de un conjunto no vacío. Este valor heurístico consiste en una medida de distancia inspirada en *branch distance* [16], que representa qué tan cerca estuvo el comando `FIND` de devolver al menos algún documento. Para realizar este cálculo, se necesita conocer los documentos almacenados en la colección. Esto es posible simplemente ejecutando un `FIND` con un filtro de selección vacío. Se calcula una distancia entre cada documento de la colección y el comando `FIND`. El mínimo de esas distancias será el valor utilizado.

Los comandos `FIND` se componen, entre otras cosas, de filtros que a su vez están compuestos por operadores que permiten obtener solo aquellos documentos que cumplan ciertas condiciones. Es importante tener en cuenta estos operadores al momento de calcular las distancias, ya que los resultados del comando dependen de ellos. En el ejemplo de la sección 3, se utilizan los operadores `and` y `eq` para obtener solo los documentos que contengan `Hernán` en el campo `name` y `Ghianni` en el campo `lastName`.

Se determinan heurísticas específicas para un subconjunto de las operaciones:

Operación	**Heurística**
Equals(*valor v*)	$\mathrm{D}(d[c], v)$
Not Equals(*valor v*)	$\begin{cases} \text{si } d[c] \neq v: & 0 \\ \text{sino:} & 1 \end{cases}$
Greater Than(*valor v*)	$\begin{cases} \text{si } d[c] - v > 0: & 0 \\ \text{sino:} & 1 - (d[c] - v) \end{cases}$
Greater Or Equals Than(*valor v*)	$\begin{cases} \text{si } d[c] - v \geq 0: & 0 \\ \text{sino:} & -(d[c] - v) \end{cases}$
Less Than(*valor v*)	$\begin{cases} \text{si } d[c] - v < 0: & 0 \\ \text{sino:} & 1 + (d[c] - v) \end{cases}$
Less Or Equals Than(*valor v*)	$\begin{cases} \text{si } d[c] - v \leq 0: & 0 \\ \text{sino:} & d[c] - v \end{cases}$
In(*valores* $v_1 ... v_n$)	$\min \mathrm{D}(d[c], v_i)$
Not In(*valores* $v_1 ... v_n$)	$\begin{cases} \text{si } v_i = d[c]: & 1 \\ \text{sino}: & 0 \end{cases}$
And(*operación* o_1, *operación* o_2, ...)	$\sum_{o_i} \mathrm{H}(o_i)$
Nor(*operación* o_1, *operación* o_2, ...)	$\sum_{o_i} \mathrm{H}(\neg o_i)$
Not(*operación o*)	$\mathrm{H}(\neg o)$
Or(*operación* o_1, *operación* o_2, ...)	$\min \mathrm{H}(o_i)$
Exists(*campo f*)	$\min \text{distanciaLexicografica}(c, f),\ c \in \mathit{campos}(d)$
NotExists(*campo f*)	$\begin{cases} \text{si } f \in \mathit{campos}(d): & 1 \\ \text{sino}: & 0 \end{cases}$
Type(*tipo t*)	$\text{distanciaLexicografica}(\mathit{tipo}(d[c]), t)$
Size(*valor s*)	$\lvert \text{longitudArray}(d[c]) - s \rvert$
Mod(*divisor div*, *resto r*)	$\lvert (d[c] \bmod \mathit{div}) - r \rvert$
All(*valores* $v_1 ... v_n$)	$\sum_{i=1}^{n} \min_{j=1}^{m} \mathrm{D}(v_i, d[c]_j)$

- d es un documento.
- Todas las operaciones aplican sobre un campo c del documento, salvo `Exists`.
- $d[c]$ representa el valor del campo c en el documento d.
- `D` es una función que toma dos valores concretos y devuelve una distancia entre ellos. Por ejemplo, para valores numéricos, la diferencia absoluta.
- `campos` es una función que devuelve una lista de los campos de un documento.

- `tipo` es una función que devuelve el nombre del tipo de un elemento.
- La función `H` calcula el valor de la heurística para una operación dada.

A continuación se presenta un ejemplo que incluye un documento Mongo y el cálculo de heurísticas de algunas operaciones:

```
{
  "_id": ObjectId("603e06b5a19f2802b8442210"),
  "nombre": "Juan",
  "apellido": "Rodriguez"
  "edad": 25,
  "gastos": [20000, 1000, 15, 23, 900]
}
```

Fig. 4.1: Documento Mongo para ejemplificar el cálculo de heurísticas.

```
db.collection.find({ "edad": 30 })
```

El valor heurístico para la primera consulta es 5. Al referirse a la tabla, se puede observar que para la operación `Equals` se debe calcular el valor de la función `D` entre el valor del campo edad en el documento, que es 25, y 30. La función `D` para enteros consiste en la diferencia absoluta entre ambos valores; por lo tanto, el resultado es 5.

```
db.collection.find({ "edad": { $gt: 30 } })
```

Para la operación `Greater Than`, se calcula el predicado $d[c] - v > 0$. En este caso, el resultado es falso para el documento, por lo que se utiliza la fórmula $1 - (d[c] - v)$ para calcular la distancia, que resulta en 6.

```
db.collection.find({ "edad": { $lte: 30 } })
```

Para `Less Or Equals Than` se debe calcular el predicado $d[c] - v \leq 0$. En este caso es verdadero y, por lo tanto, el cálculo da 0.

```
db.collection.find({ "edad": { $in: [10, 24] } })
```

La fórmula en este caso es $\min \mathrm{D}(d[c], v_i)$. $\mathrm{D}(25, 10)$ es 15 y $\mathrm{D}(25, 24)$ es 1. El resultado es el mínimo entre ambos, en este caso, 1.

```
db.collection.find({ "edad": { $nin: [10, 24] } })
```

Para esta operación, basta con que alguno de los valores sea igual al del documento para que el resultado sea 1. En este caso, como el valor en el documento es 25 y tanto 24 como 10 son distintos, el resultado es 0.

```
db.collection.find({ "edad": { $gt: 20 }, "edad": { $gt: 30 } })
```

El cálculo de la operación `And` se compone como la suma del cálculo de las operaciones que la componen. Si se calcula para la operación **Greater Than** como se explicó anteriormente, el valor heurístico obtenido sería de $0 + 6 = 6$.

```
db.collection.find({ "edad": { $not: { $eq: 30 } } })
```

En esta ocasión, se requiere realizar una negación de la operación. Esto implica utilizar una operación alternativa que genere resultados opuestos a aquellos que la operación original produce. Para `Equals`, se puede emplear `Not Equals`. Dado que el valor en el documento no coincide con el de la operación, el resultado es 0.

```
db.collection.find({ "edad": { $type: "string" } })
```

Para la operación `Type`, se calcula la distancia lexicográfica entre el nombre del tipo del campo en cuestión y el de la operación. Como el tipo del campo `edad` es int, entonces la comparación se realiza entre los strings "`int`" y "`string`".

```
db.collection.find({ "gastos": { $size: 10 } })
```

Esta operación aplica para arrays y se calcula como la diferencia entre la cantidad de elementos del array en el documento y la cantidad que indica la operación. Dado que `gastos` tiene 5 elementos y el valor es 10, se obtiene 5.

```
db.collection.find({ "edad": { $mod: [5, 1] } })
```

El cálculo se realiza como la diferencia absoluta entre los restos. El resto de 25 mod 5 es 0, por lo tanto, el resultado es 1.

```
db.collection.find({ "gastos": { $all: [1, 2] } })
```

Por último, para la operación `All`, se realiza la suma de la mínima distancia entre 1 y 2 y cada valor en el campo `gastos`. Dado que el valor más cercano a ambos es el 15, la distancia mínima es de 14 y 13 respectivamente, siendo 27 el resultado final.

4.3. Integración de heurísticas al Fitness

Una vez obtenida una distancia heurística para cada operación `FIND`, estas deben utilizarse para mejorar la búsqueda. El objetivo principal es aumentar la cobertura de código en el SUT. Sin embargo, estas heurísticas no garantizan necesariamente una mayor cobertura, sino que pueden contribuir a alcanzar el objetivo. Por lo tanto, se considerarán estas heurísticas como secundarias. Cuando dos individuos tienen el mismo valor principal, la función de *fitness* analiza el objetivo secundario para determinar cuál de los dos individuos

es más adecuado para la reproducción. Esto otorga mayor valor a los tests que se acerquen más a tener operaciones `FIND` que devuelvan conjuntos de documentos no vacíos.

5. GENERACIÓN DE DATOS EN BASES MONGO

Como se explicó en la sección 3, existen situaciones más desafiantes en las que las heurísticas resultarán insuficientes para la generación de tests. Un caso particularmente interesante se presenta cuando el servicio no ofrece una manera directa de insertar documentos en la base de datos. En tales circunstancias, podría ser necesario que la base de datos contenga información específica para poder generar un test. Por ejemplo, esto ocurre cuando un hay un `if` que depende del resultado de una operación `FIND`. Este escenario plantea un problema significativo, dado que la ausencia de un mecanismo para la inserción de documentos impide la generación de este tipo de tests.

Para abordar estos problemas, se propone ampliar EvoMaster con la capacidad de insertar información directamente en bases de datos Mongo. Se reutilizará el mecanismo de captura de comandos de la sección 4 de manera de insertar datos cuando se detecte una operación FIND aplicada a una colección que esté vacía. La razón detrás de esta decisión es que la función de *fitness*, junto con las heurísticas descriptas anteriormente, guiarán los valores en la base de datos hacia los deseados. Sin embargo, si la colección está vacía, esto no será posible. Por lo tanto, el objetivo es insertar información que sea válida para la colección, sin importar los valores concretos, para que eventualmente converjan a los buscados.

Para insertar datos en la base de datos, es fundamental comprender su estructura. En el caso de las bases de datos relacionales, están compuestas por tablas, cada una con una estructura definida que incluye columnas de tipos específicos. En contraste, MongoDB presenta una situación más compleja debido a su estructura flexible. En lugar de tablas, MongoDB utiliza colecciones, las cuales pueden contener documentos con una variada cantidad de campos y tipos.

Sin embargo, al utilizar un framework como Spring, es común asignar un tipo a los repositorios. Por ejemplo, en el caso del repositorio de `Person` mencionado en la sección 3, las operaciones se realizan internamente sobre una colección de Mongo. Dado que el tipo del repositorio está limitado, los documentos que se inserten también estarán limitados por la estructura de `Person`. Por lo tanto, para que un documento sea válido, debe contener campos como `name` y `lastName`, con valores de tipo `String`.

Para resolver este inconveniente, se propone la creación de una nueva clase encargada de capturar el tipo de la colección en el momento de su inicialización, similar a cómo se manejan los comandos `FIND`, y así permitir la inserción de información válida.

Cuando se detecta un `FIND` que se ejecutó en una colección vacía, se debe generar un documento en la colección correspondiente. Para ello se busca su tipo, el cual incluye una serie de campos con un tipo determinado, y se crean genes para cada uno en EvoMaster. Esto va a permitir generar un documento valido pero con valores aleatorios durante el proceso de mutación de los individuos. Cabe aclarar que la mutación funciona de forma muy similar a la ya implementada en la herramienta, con la excepción de que se deben tener en cuenta tipos específicos de Mongo. Posteriormente se deben traducir los genes en información que Mongo pueda interpretar. Para esto se utiliza Extended JSON [17], una extensión de JSON que contempla todos los tipos de Mongo de manera de permitir inserciones a partir de `Strings` en formato JSON. Cada documento a insertar, se transforma en un `String` en Extended JSON y se inserta en la colección.

Una vez finalizada la búsqueda, EvoMaster genera un archivo de clase con tests JUnit. Para poder ejecutar las inserciones Mongo generadas durante la búsqueda, se amplia EvoMaster para manejar comandos Mongo directamente desde los tests. Los tests resultantes contendrán una secuencia de cero o mas inserciones en la base de datos seguidos de las acciones en el SUT (ej. un `GET`).

En la figura 5.1 puede observarse el proceso descripto.

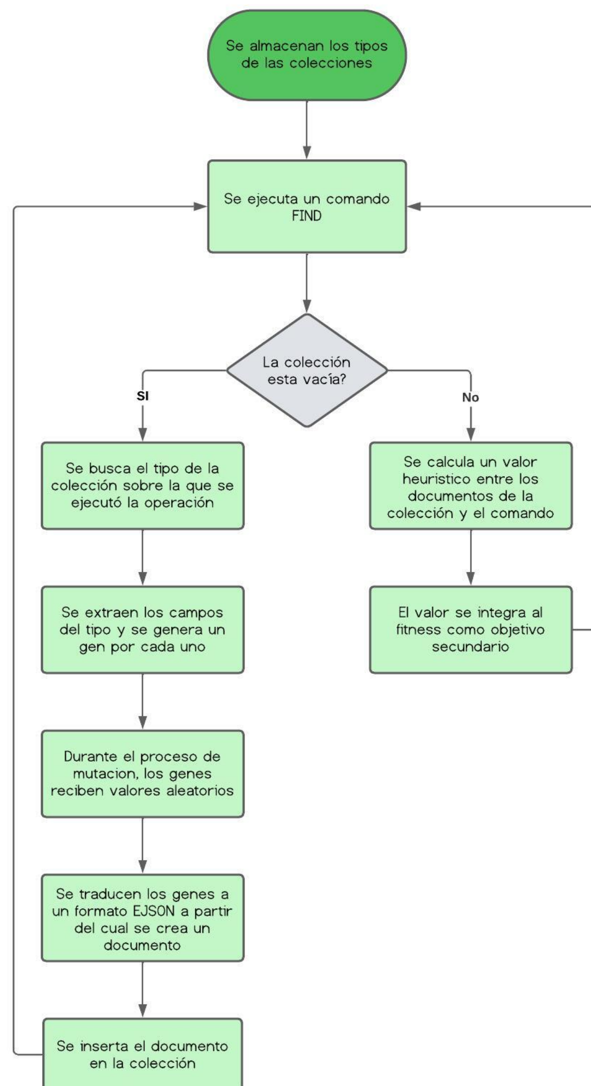

Fig. 5.1: Proceso de mejora de testing automático para bases Mongo.

Para comprender mejor la solución, se presenta el ejemplo ilustrado en la figura 5.2. Este ejemplo consiste en una clase llamada `Person`, un repositorio de tipo `Person`, y una API con un único método GET. Este método devuelve un código de estado 200 si alguna instancia en el repositorio de *persons* tiene el campo *age* con el valor 27, y un código de estado 400 en caso contrario.

Como se mencionó anteriormente, internamente, un repositorio funciona como una colección. Por lo tanto, un repositorio de tipo `Person` equivale a una colección cuyos

documentos tienen la misma estructura que los objetos de tipo `Person`. Es importante notar que para generar un test que devuelva un código de estado 200, es necesario que la colección contenga un documento con una estructura y un valor específico.

```
@RestController
@RequestMapping("/api/persons")
public class PersonController {

    @Autowired
    private PersonRepository personRepository;

    @GetMapping("/listTwentySeven")
    public ResponseEntity<List<Person>> get {
        List<Person> persons = personRepository.findByAge(27)

        if (persons.isEmpty()) {
            return new ResponseEntity.status(400).build();
        } else {
            return new ResponseEntity.status(200).build();
        }
    }
}

public class Person {
    private String firstName;
    private String lastName;
    private Date dateOfBirth;
    private Integer age;
}
```

Fig. 5.2: Ejemplo de una Web API en Java SpringBoot que utiliza una base de datos Mongo.

Al inicio, EvoMaster captura la información de las colecciones que están presentes en el SUT. Por ejemplo, guarda que la colección *persons* tiene el tipo *Person*. Cuando se ejecuta una operación *FIND* sobre la colección *persons*, EvoMaster detecta esta acción y comienza un proceso para insertar información válida en la base de datos (ver figura 5.3). En primer lugar, busca el tipo de la colección, el cual ha almacenado previamente. Luego, extrae los campos de la clase *Person* y sus respectivos tipos, como *name* y *lastName* de tipo *String*, *birthDate* de tipo *Date*, y *age* de tipo *Integer*. Durante el proceso de mutación, se asigna a estos campos un valor aleatorio. Finalmente, se crea un documento en formato Extended JSON a partir de estos campos y se inserta directamente en la base de datos. La próxima vez que ocurra una operación *FIND*, la colección ya no estará vacía y, por lo tanto, el cálculo de heurísticas será factible. Esto permite que eventualmente la colección contenga un *Person* con los valores deseados, como por ejemplo, el campo *age* con valor 27.

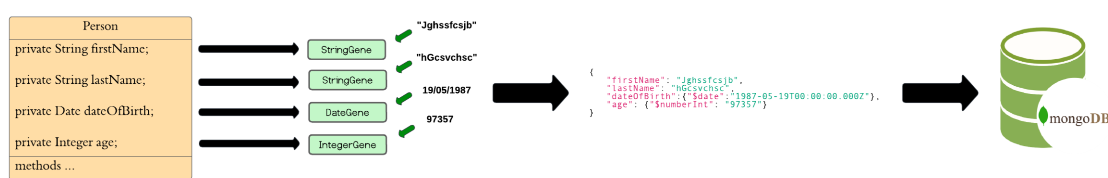

Fig. 5.3: Proceso de mutación en EvoMaster.

Una vez generado el test deseado, debe incluirse una forma de insertar en la base de datos desde allí. Para eso se desarrolló un DSL (Lenguaje Específico de Dominio) específico para Mongo que permite realizar este tipo de operaciones. En la figura 5.4 se presenta el archivo generado por EvoMaster que posibilita obtener una respuesta 200.

```
@Test
fun test() {
    val insertions = mongo().insertInto("persons", "person")
    .d("""
        {
            "firstName": "Jghssfcsjb",
            "lastName": "hGcsvchsc",
            "dateOfBirth": {"$date": "1987-05-19T00:00:00.000Z"},
            "age": {"$numberInt": "27"}
        }
    """.trimIndent())
    .dtos()

    controller.execInsertionsIntoMongoDatabase(insertions)

    given().accept("*/*")
                .get("${baseUrlOfSut}/persons/listTwentySeven")
                .then()
                .statusCode(200)
    }
```

Fig. 5.4: Ejemplo de un JUnit test generado para la API de la Figura 5.2

6. EXPERIMENTACIÓN

Para estudiar el rendimiento de esta extensión, se eligieron cinco APIs REST de distintos tamaños que hacen uso de MongoDB: *bibliothek* [18], *genome-nexus* [19], *gestaohospital* [20], *ocvn* [21] y *reservations* [22]. Estas APIs son parte de EvoMaster Benchmark (EMB) [23], una compilación de APIs para experimentar con EvoMaster. Para habilitar su integración, se creó un controlador para cada una, permitiendo así el acceso a la base de datos. La tabla 6.1 muestra algunas estadísticas de las APIs utilizadas, incluyendo la cantidad de clases, líneas de código y la cantidad de endpoints.

API	**#Clases**	**#LDC**	**#Endpoints**
bibliothek	53	1561	9
genome-nexus	405	30004	23
gestaohospital	33	3506	20
ocvn	526	45521	258
reservations	31	1576	9

Tab. 6.1: Estadísticas de las APIs utilizadas en la experimentación.

Para cada API, se ejecutó EvoMaster en diez ocasiones, utilizando tres configuraciones diferentes, con una duración de una hora para cada ejecución. Esto se debe a que EvoMaster no es determinístico. La primera configuración corresponde a la versión predeterminada de EvoMaster. La segunda configuración incluye las heurísticas de Mongo, que consiste en la incorporación de estos valores como objetivos secundarios en la función de fitness de la sección 4. Por último, una configuración que integra la generación de datos MongoDB directamente, tal como se explica en la sección 5.

API	*bibliothek*	*genome-nexus*	*gestaohospital*	*ocvn*	*reservations*
Base	99	1735,7	465,6	1725	165,4
H	99	1738,6	454,6	1716	169
G+H	117,1	1712,5	606,6	1717	186,6
p-valor$_{\mathbf{hb}}$	-	0,97	0,51	0,79	$9,66 \times 10^{-5}$
p-valor$_{\mathbf{gb}}$	$5,98 \times 10^{-5}$	0,44	$1,58 \times 10^{-4}$	0,6	$3,58 \times 10^{-5}$
p-valor$_{\mathbf{gh}}$	$5,98 \times 10^{-5}$	0,34	$1,49 \times 10^{-4}$	0,73	$2,43 \times 10^{-5}$
$\mathbf{\hat{A}_{hb}}$	0,5	0,51	0,42	0,46	0,95
$\mathbf{\hat{A}_{gb}}$	1	0,39	1	0,41	1
$\mathbf{\hat{A}_{gh}}$	1	0,37	1	0,44	1

Tab. 6.2: Promedio de líneas de código cubiertas para las configuraciones base, heurísticas de MongoDB (H) y generación de datos MongoDB directamente (G+H). $\hat{A}_{12}$ y *p-valor* para cada combinación de dos configuraciones.

La tabla 6.2 presenta los resultados de los experimentos para cada configuración. El valor obtenido en cada ejecución de EvoMaster es la cantidad de lineas de código cubiertas.

Para analizar los resultados, se siguió la metodología recomendada en [24]. Se calculó el promedio de las líneas de código cubiertas en las 10 ejecuciones, se aplicó el test estadístico no paramétrico *U de Mann-Whitney* y se empleó la medida $\hat{A}_{12}$ de *Vargha-Delaney.*

El test de *Mann-Whitney* permite determinar si las diferencias observadas entre dos grupos son significativas o si podrían ser atribuidas al azar. En este contexto, se utiliza para comparar diferentes configuraciones. La hipótesis nula asume que las diferencias son aleatorias. Se rechaza la hipótesis nula si $p \leq 0{,}05$, lo que indica que las diferencias entre las configuraciones no son resultado del azar. Por lo tanto, valores de p más bajos proporcionan mayor confianza en que las diferencias entre las configuraciones son reales y no aleatorias. Los *p-valores* obtenidos se muestran en la tabla.

La medida de *Vargha-Delaney* se emplea para evaluar la magnitud de la diferencia entre configuraciones. Por ejemplo, $\hat{A}_{hb} = 0{,}70$ indica que el 70 % de los resultados fueron mejores para la configuración con heurísticas en comparación con la configuración predeterminada. Según sus autores, estos resultados pueden interpretarse como que $\hat{A}_{12} \geq 0{,}56$ es una diferencia pequeña, $\hat{A}_{12} \geq 0{,}64$ es mediana y $\hat{A}_{12} \geq 0{,}71$ es grande.

Las heurísticas de MongoDB por si solas muestran mejoras solo en dos APIs, y estas mejoras son bastante modestas. La API *reservations* obtiene un resultado alto de $\hat{A}_{hb}$ y un *p-value* mas bajo que 0,05, siendo la mas destacada. Por otro lado, en dos casos, las heurísticas resultan en un rendimiento inferior, mientras que en un caso, el rendimiento es igual al de la configuración base. Se puede concluir que las heurísticas no resultan suficientes para generar mejoras importantes.

La generación directa de datos en MongoDB, por otro lado, ofrece beneficios significativos en comparación. Tres casos presentan mejoras, siendo el más notable el de *gestaohospital*, que muestra un aumento del 30,3 % en relación con la versión base. Estos resultados están respaldados por las medidas, ya que $\hat{A}gb$ y $\hat{A}gh$ tienen un valor de 1 en todos los casos, lo que indica que los resultados son superiores en cada una de las 10 iteraciones para la configuración con generación directa de datos en comparación con las otras 2 configuraciones en sus 10 ejecuciones. Además, los valores de p son bajos, lo que también avalan este resultado.

7. CONCLUSIÓN

En este trabajo se introduce una técnica innovadora que mejora el testing automático para microservicios que hacen uso de la bases de datos MongoDB. Esta técnica no solo permite generar heurísticas a partir de las operaciones FIND, que se integran en la función de fitness, sino que también mejora el proceso incluso en casos donde el microservicio solo permite operaciones de lectura, al generar datos directamente en la base de datos.

La técnica se implementó como una extensión de EvoMaster, una herramienta de generación automática de tests. Experimentos realizados en 5 APIs demostraron mejoras significativas, con un incremento en el cubrimiento de líneas de código de hasta un 30 %.

La investigación y desarrollo llevados a cabo permiten a EvoMaster ofrecer una funcionalidad más completa y efectiva. Además, abren la posibilidad de integrar en el futuro otras bases de datos NoSQL, como por ejemplo DynamoDB.

Bibliografía

[1] Andrea Arcuri and Juan P. Galeotti. SQL Data Generation to Enhance Search-Based System Testing. 2019.

[2] [n. d.]. MongoDB. https://www.mongodb.com/es.

[3] Neil Perkin and Peter Abraham. *The API Economy: Disruption and the Business of APIs*. CreateSpace Independent Publishing Platform, 2016.

[4] Mark Masse. *REST API Design Rulebook*. 2011.

[5] Daniel Jacobson, Greg Brail, and Dan Woods. *APIs: A Strategy Guide*. 2011.

[6] Stefan Tilkov. A Brief Introduction to REST. 2007.

[7] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[8] [n. d.]. BSON. https://bsonspec.org/.

[9] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[10] Andrea Arcuri. Many Independent Objective (MIO) Algorithm for Test Suite Generation. *International Symposium on Search Based Software Engineering (SSBSE)*, pages 3–17, 2017.

[11] [n. d.]. EvoMaster. https://github.com/EMResearch/EvoMaster.

[12] A. Arcuri. Test Suite Generation with the Many Independent Objective (MIO) Algorithm. *Information and Software Technology (IST)*, 2018.

[13] [n. d.]. REST Assured. https://github.com/rest-assured/rest-assured.

[14] [n. d.]. SpringBoot. https://spring.io/projects/spring-boot/.

[15] [n. d.]. Mongo Java. https://www.mongodb.com/docs/drivers/java-drivers/.

[16] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[17] [n. d.]. Extended JSON. https://www.mongodb.com/docs/manual/reference/mongodb-extended-json/.

[18] [n. d.]. Bibliothek API. https://github.com/PaperMC/bibliothek.

[19] [n. d.]. Genome Nexus API. https://github.com/genome-nexus/genome-nexus.

[20] [n. d.]. Hospital API. https://github.com/ValchanOficial/GestaoHospital.

[21] [n. d.]. OCVN API. `https://github.com/devgateway/ocvn`.

[22] [n. d.]. Reservations API. `https://github.com/cyrilgavala/reservations-api`.

[23] [n. d.]. EvoMaster Benchmark (EMB). `https://github.com/EMResearch/EMB`.

[24] A. Arcuri and L. Briand. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)*, 24(3):219–250, 2014.