



Máster Universitario En Investigación En Ingeniería De Software Y Sistemas Informáticos

Generación Automática de Código

Autor

César Hugo Bárzano Cruz



TRABAJO DE INVESTIGACIÓN

—
2017/2018

Índice general

1. Resumen	7
2. Introducción	9
3. El Problema	11
3.1. Metodologías Ágiles	11
3.2. Servicios web: REST	15
3.2.1. Arquitecturas Basadas en micro-servicios	15
3.3. GAC en Servicios Web	16
4. Desarrollo de la práctica	19
4.1. Recursos de programación utilizados	19
4.2. Especificación de la solución	20
4.2.1. Comportamiento de la Solución	21
4.3. Alcance y Limitaciones	21
4.4. Pruebas	22
4.4.1. Escenario 1: People Service	22
4.4.2. Escenario 2: Players Location	27
4.4.3. Escenario 3: Miauuu as a service	33
4.4.4. Escenario 4: Múltiples Servicios	34
5. Entrega	37
5.1. Generador	37
5.2. Makefile	37
5.3. TI.CesarHugoBarzanoCruz.pdf	37
5.4. Directorio DOC	37
5.5. Directorio src/testgenerator/input	38
5.6. Directorio src/testgenerator/output	38
6. Anexo	41

Índice de figuras

3.1. Scrum Process[4]	12
3.2. Kamban Table[5]	13
3.3. Lean Process[6]	14
3.4. XP Practices[8]	15
3.5. Arquitectura Basada en Microservicios[9]	16
4.1. Servicio generado people.json	23
4.2. Servicio generado people.json	23
4.3. Ejemplo Ejecución /people	24
4.4. test_people.sh result	26
4.5. Servicio generado location_api.json	28
4.6. Servicio generado location_api.json	28
4.7. Servicio corriendo /location	30
4.8. test_location.sh result	32
4.9. HTTP Status Codes	33
4.10. Servicios /cats generados	34
4.11. Múltiples Inputs	35
4.12. Múltiples Outputs	35

Capítulo 1

Resumen

En el dominio de la WEB 2.0 el nuevo modelo de desarrollo al que tienden las empresas se basa en metodologías agile, donde se premia la rápida entrega de pequeñas funcionalidades junto con iteraciones mas seguidas con los equipos. Es común la creación de pequeños servicios, con funcionalidad muy específica para cubrir cierta necesidad de negocio y ademas de una manera dinámica y volátil. Esto puede suponer una carga de trabajo importante para los equipos de desarrollo, pues ademas de mantener los servicios funcionales han de crear nuevos constantemente, por este motivo se propone el siguiente trabajo basado en la generación dinámica de servicios web. La idea de este generador es la de reducir el tiempo de desarrollo de nuevos servicios procurando al desarrollador de un esqueleto o scaffolding funcional de un modelo de negocio muy concreto. La generación automática de servicios también puede resultar muy útil en el campo de la formación o capacitación ya que puede proporcionar el proyecto o estructura base para que el capacitador pueda mostrar los conceptos importantes en el desarrollo de servicios web a los alumnos o técnicos que reciban dicha capacitación.

El objetivo de esta investigación es el de automatizar mediante técnicas de generación automática de código la generación de servicios web base REST. El resultado de esta investigación serán un generador de servicios web en distintos lenguajes, así como un pequeño caso de uso a modo de conjunto de pruebas automáticas que permita validar el correcto funcionamiento del servicio generado.

Capítulo 2

Introducción

En los últimos años, la WEB 2.0 ha evolucionado de manera veloz debido a la gran demanda de información que tanto usuario como empresa han generado con el uso de las nuevas tecnologías. El desarrollo de aplicaciones web basadas en servicios ha marcado un antes y un después, haciendo que los datos tomen un nuevo valor. La transformación digital es un hecho que está cambiando el modo en que las empresas y usuarios interactúan con la información puesto que el acceso a los datos facilita la toma de decisiones, datos que hasta hace unos años no se pensaba que tendrían el valor social y económico que tienen hoy en día. Es notable como la información y los canales de comunicación crecen día a día satisfaciendo necesidades encubiertas que el usuario no conocía, automatizando procesos de negocio y mejorando tareas de gobierno.

El desarrollo de aplicaciones web normalmente va asociado a procesos de desarrollo ágil donde se premia la alta disponibilidad, la rápida creación y la integración de nuevos servicios de información con los ya disponibles. Estas metodologías son responsables en su justa medida del rápido crecimiento de lo denominado WEB 2.0 o infraestructura de información distribuida a la cual, todos podemos acceder con los innumerables dispositivos que nos rodean.

Estas metodologías, con aspectos muy beneficiosos para el ingeniería del software, tienen ciertas carencias con respecto a las metodologías de desarrollo tradicional. Si por un momento, dejamos de lado las tecnologías de la información y nos centramos en otros sectores donde el software cumple un papel vital, como por ejemplo proyectos en el ámbito espacial, defensa, automoción o aeronáutica, las líneas de desarrollo ágil son rechazadas por completo, puesto que el software resultante de estos proyectos ha sido validado y testeado durante largos años, sometidos a largos procesos de calidad y pruebas, procesos muy costosos y lentos para ser aplicados en el desarrollo de aplicaciones web o servicios cloud.

La gran demanda de información y soluciones cloud en ocasiones puede

ocasionar ciertos problemas ya que se disponibilizan servicios o versiones cuyo estado no es fiable, bajo el escudo de que los servicios de la información son de carácter volátil. Esta demanda produce una gran cantidad de software que ha de ser mantenido y actualizado, tarea complicada de manejar si se tiene en cuenta que cada solución puede ser implementada por distintos equipos, con distintos lenguajes de programación o tecnologías. Es aquí donde la generación automática de código marca la diferencia. Como se verá en los siguientes capítulos se propone la implementación de un generador de servicios cloud basados en REST cuyo objetivo sea la unificación del desarrollo de servicios. Dando un punto común y estructura base que permita a los desarrolladores avanzar rápidamente, produciendo de esta forma productos software de similar estructura, aunque desarrollen con distintas tecnologías.

Para ello, con el foco en el desarrollo ágil de aplicaciones se va a plantear una solución a como los servicios de la información han de exponerse al mundo, garantizando que su resultado final es el esperado aplicando técnicas de validación tradicional a los procesos de desarrollo ágil, siendo aquí donde la generación automática de código desempeñará un papel fundamental, ya que se intentará automatizar el proceso generativo del plan de validación de estos servicios. De esta manera, la solución final, verificará el desarrollo de aplicaciones sin incluir la dilación en el tiempo que conlleva la ejecución de un plan de validación completo sobre un proyecto software.

Capítulo 3

El Problema

El desarrollo basado en metodologías ágiles principalmente va enfocado a proyectos que precisan de una especial rapidez y flexibilidad en su proceso. En muchas ocasiones son proyectos relacionados con el desarrollo de software o el mundo de internet. En sectores constantemente cambiantes, las organizaciones necesitan desarrollar sus servicios rápidamente para ser altamente competitivos, y esto no es tarea fácil. Muchas veces es necesario ir probando las distintas funcionalidades del servicio sobre la marcha y medir si está funcionando o no para acabar ofreciendo una solución final.

Si se utilizan metodologías de desarrollo tradicionales, estas revisiones (o tests) pueden suponer un retraso en las fechas de entrega, aumento de costes y del volumen de trabajo. Además, también podría suceder que para cuando se tenga el producto final éste ya quede obsoleto. He aquí la importancia del desarrollo Agile.

Las metodologías ágiles se basan en un enfoque flexible. Los miembros del equipo trabajan en pequeñas fases y equipos sobre actualizaciones concretas del producto. Después, se prueba cada actualización en función de las necesidades del cliente. El producto final de un proyecto ágil puede perfectamente ser distinto al que se había previsto inicialmente. No obstante, durante los procesos de pruebas se sigue trabajando según los requerimientos del cliente, de forma que el producto final sigue respondiendo a sus necesidades. En las siguientes secciones se abordarán las metodologías ágiles más comunes y como afectan al desarrollo de servicios cloud bajo arquitecturas basadas en microservicios, y como la generación automática de código desempeña un gran papel.

3.1. Metodologías Ágiles

Scrum

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener

el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

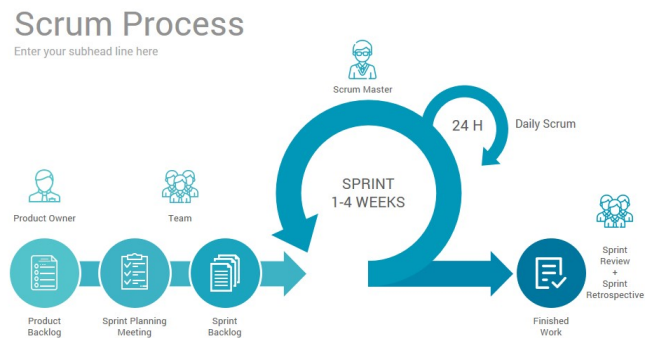


Figura 3.1: Scrum Process[4]

Kanban

La metodología Kanban es una manera de gestionar el trabajo de forma fluida. La palabra Kanban proveniente de Japón, es un símbolo visual que se utiliza para desencadenar una acción. A menudo se representa en un tablero Kanban para reflejar los procesos de su flujo de trabajo.

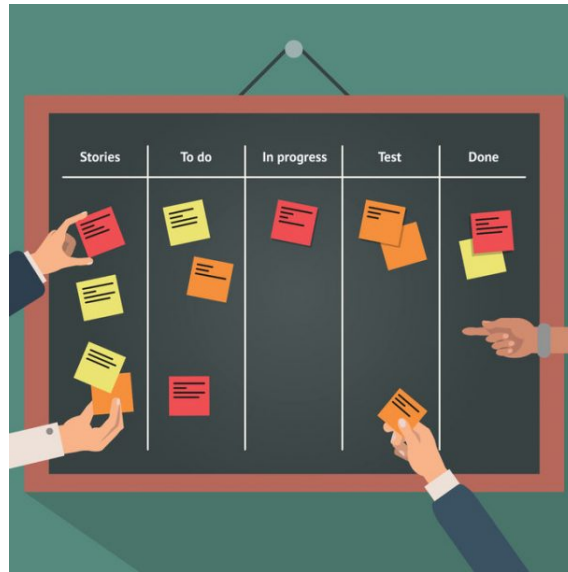


Figura 3.2: Kamban Table[5]

Lean

Lean son una serie de principios enfocados a eliminar todas las tareas que no aporten valor sobre aquello que estamos realizando. Aplicado a un proyecto, aquello que no aporte valor al producto o servicio final. Para ello, visualiza todo el proceso que se produce en una cadena de valor a lo largo de un proyecto y, en todos aquellos puntos que se produce desperdicio, lo elimina.

LEAN Process

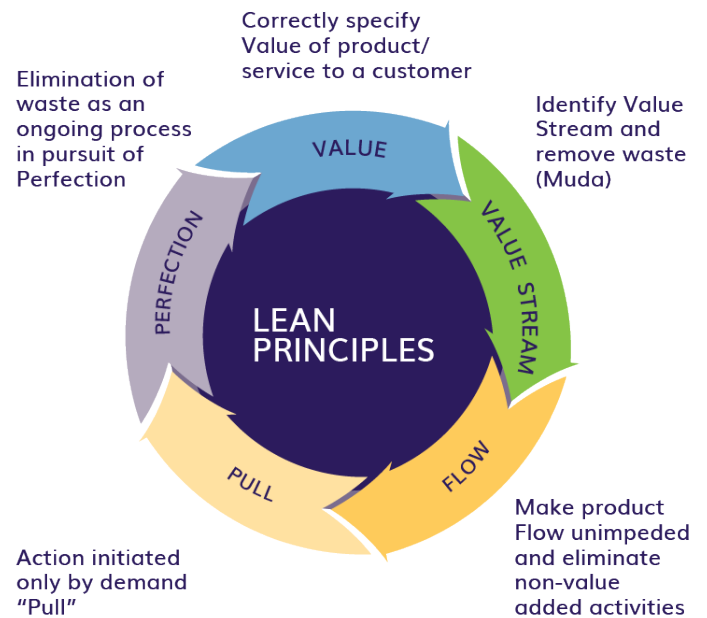


Figura 3.3: Lean Process[6]

XP Programming

La programación extrema o eXtreme Programming (XP) es un enfoque de la ingeniería de software formulado por Kent Beck, autor del primer libro sobre la materia, *Extreme Programming Explained: Embrace Change*[7]

Es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

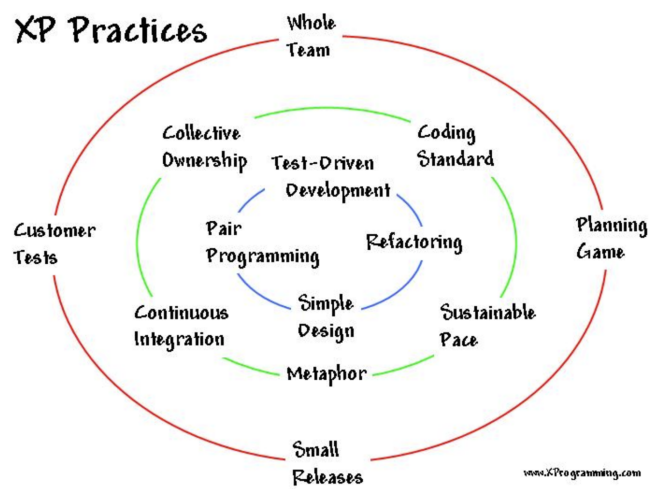


Figura 3.4: XP Practices[8]

3.2. Servicios web: REST

La Transferencia de Estado Representacional (REST - Representational State Transfer) fue ganando amplia adopción en toda la web como una alternativa más simple a SOAP y a los servicios web basados en el Language de Descripción de Servicios Web (Web Services Description Language - WSDL).

REST define un set de principios arquitectónicos por los cuales se diseñan servicios web haciendo foco en los recursos del sistema, incluyendo cómo se accede al estado de dichos recursos y cómo se transfieren por HTTP hacia clientes escritos en diversos lenguajes. Los 4 principios de REST son:

1. Utiliza los métodos o verbos HTTP de manera explícita
2. No mantiene estado
3. Expone URIs
4. Transfiere XML, JavaScript Object Notation (JSON), o ambos

3.2.1. Arquitecturas Basadas en micro-servicios

El enfoque tradicional sobre el desarrollo de aplicaciones se centró en el monolito, donde todas las partes de la aplicación que se pueden implementar están contenidas en esa única aplicación. Esto tiene sus desventajas: cuanto más grande es la aplicación, más difícil resulta solucionar problemas nuevos

y agregar funciones nuevas con rapidez. Un enfoque basado en los microservicios para el diseño de las aplicaciones resuelve estos problemas e impulsa el desarrollo y la capacidad de respuesta.

Los microservicios se pueden considerar tanto un tipo de arquitectura como un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus componentes más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se integra en una única pieza, los microservicios son independientes y funcionan en conjunto para llevar a cabo las mismas tareas. Cada uno de estos elementos o procesos es un microservicio. Este enfoque sobre el desarrollo de software privilegia el nivel de detalle, la sencillez y la capacidad de compartir procesos similares en varias aplicaciones. Es un componente fundamental de la optimización del desarrollo de aplicaciones hacia un modelo nativo de la nube.

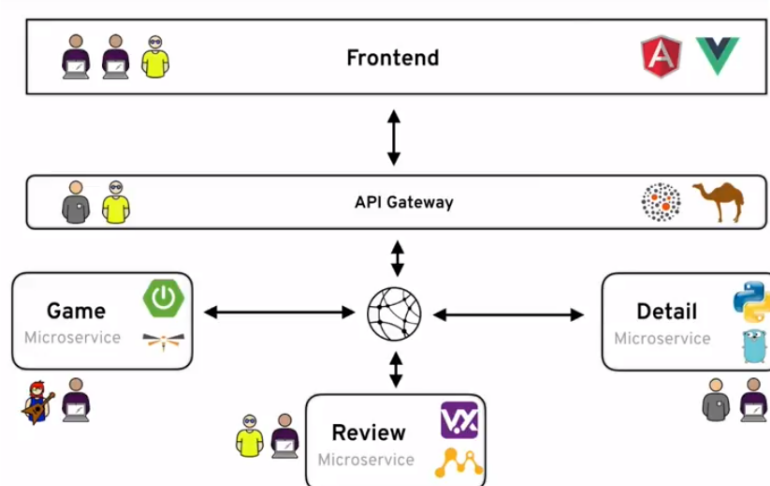


Figura 3.5: Arquitectura Basada en Microservicios[9]

3.3. GAC en Servicios Web

De las distintas metodologías ágiles utilizadas en el proceso de desarrollo de servicios web, se pone de manifiesto que el objetivo es cubrir rápidamente la demanda de nuevas funcionalidades o features que no estaban previstas inicialmente, evolucionando bajo demanda del cliente final e iterando con pequeñas entregas del producto. Por otro lado, se ha visto que la arquitectura orientada a microservicios está ligada a este tipo de productos e intenta descomponer en la medida de lo posible todas las funcionalidades o recursos rompiendo con las arquitecturas monolíticas tradicionales y mostrando el producto o sistema final como la suma de muchas piezas pequeñas que cumplen una funcionalidad concreta. Es aquí donde la generación automáti-

ca de código puede marcar la diferencia, ya que ante la demanda de nuevas funcionalidades es preferible la creación (o generación) de nuevas piezas que intenten resolver el problema concreto de la manera más atómica posible.

Para ello, se propone en este trabajo la implementación de un generador de servicios web basado en REST que permita a los equipos de desarrollo generar nuevas funcionalidades asegurando que el código de los servicios generados sea todo lo genérico que se pueda, de manera que las reglas de codificación, Apis, estructura de los proyectos, separación entre capas, acceso a base de datos, pruebas y en general toda funcionalidad base esperada por un servicio web sea proporcionada por el generador de manera automática, transversal y común.

Capítulo 4

Desarrollo de la práctica

Visto el enunciado del problema en el capítulo anterior, se propone la implementación de un generador de servicios web basados en REST.

4.1. Recursos de programación utilizados

El generador ha sido desarrollado mediante el lenguaje de programación GO[1]. Go es un lenguaje de programación inspirado en C, compilado, estáticamente tipado, concurrente y con un enfoque de programación orientada a objetos. Principales características de GO:

1. Compilado: No es necesario instalar ningún programa para que el programa desarrollado funcione en el sistema operativo para el que se compiló.
2. Estáticamente Tipado: Las variables son tipadas de manera estática, así que si la variable `x` se define como entera, será entera durante todo su alcance.
3. Concurrente: Está inspirado en CSP - Communicating sequential processes[10]
4. Uso de paquetes: Usa paquetes para organizar el código.

El generador propuesto es capaz de generar servicios web en el lenguaje de programación GO y también en el lenguaje de programación Javascript[11] haciendo uso del framework NODE. JavaScript es un lenguaje de programación interpretado. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Node[12] (`node.js`) es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome.

De manera adicional y con el objetivo de validar que efectivamente el código generado para el servicio es correcto, el generador también produce un caso de uso básico en el que probar las capacidades CRUD(Create,

Read, Update and Delete) del servicio. Para ello, el generador producirá un caso de uso en lenguaje de scripting BASH[13] donde usando la instrucción cURL[14] se automatizarán distintas peticiones HTTP al servicio generado.

Finalmente para la confección de esta memoria se ha utilizado LaTeX con el paquete de librerías texlive-full y el editor texmaker. Los siguientes enlaces muestran como instalar y utilizar correctamente LaTeX, han sido utilizados como referencia para el presente documento.

1. Instalar LaTeX
2. Usar LaTeX

4.2. Especificación de la solución

El generador puede ser encontrado en el siguiente repositorio de GitHub [hugobarzano/restandtestgenerator](https://github.com/hugobarzano/restandtestgenerator) el código relativo al generador será entregado como parte del trabajo de investigación, pero para facilitar la comprensión de las partes que forman el generador, se hará referencia a dicho repositorio de GitHub.

A grandes rasgos, en una ejecución nominal del generador, se leerán todos los ficheros que se encuentren en el directorio input/. Por cada uno de los ficheros de este directorio, el generador producirá el código funcional correspondiente a un servicio web en GO y en NODE. De manera adicional, se generará un caso de prueba donde se realizaran peticiones a la API generada mediante Shell scripting. El siguiente bloque muestra un ejemplo de configuración, correspondiente al fichero de ejemplo `restandtestgenerator/input/cats.cloud.service.json`.

Listing 4.1: Ejemplo Entrada Generador

```
1 [{  
2   "name": "Miauuu as a service",  
3   "service": "/cats",  
4   "url": "http://localhost",  
5   "port": "8080",  
6   "body": {  
7     "name": "Wurst",  
8     "alias": "Mi bolita gatita bonita",  
9     "reina": true,  
10    "age": 1.5,  
11    "color": "Negro como mi alma"  
12  }  
13 }]
```

Los campos correspondientes al ejemplo de configuración representan:

1. "name": Nombre común del servicio.
2. "service": Api generada donde se expondrá el servicio.

3. "url": Host donde se probará el servicio.
4. "port": Puerto donde el servicio será expuesto.
5. "body": Representa el modelo de negocio o modelo de datos. Es la entidad que el servicio será capaz de manejar. Los datos de ejemplo dados serán usados en la generación de las pruebas.

4.2.1. Comportamiento de la Solución

En la siguiente sección dedicada a Pruebas se ejemplificará el comportamiento del generador mediante unos casos de prueba. De manera literal, el generador cargará los ficheros de configuración (directorio input) para cada uno de estos ficheros que especifican un servicio web:

1. El generador producirá el Scaffolding o estructura base del servicio GO.
2. El generador producirá el Scaffolding o estructura base del servicio NODE.
3. El generador producirá el Modelo de negocio y la configuración necesaria para la base de datos del servicio GO.
4. El generador producirá el modelo de negocio para el servicio NODE.
5. El generador producirá el TestCase relativo al servicio generado. El Test Case será producido en shell scripting por lo que puede ser independientemente ejecutado contra cualquiera de las tecnologías con las que el servicio es generado.

4.3. Alcance y Limitaciones

El generador propuesto es capaz de generar servicios funcionales en GO y en NODE. Los servicios generados están constituidos por una API REST CRUD que soporta los verbos http: GET, POST, PUT, DELETE. El modelo de negocio generado para el servicio es dinámico. El Generador produce toda la configuración necesaria para la capa de persistencia. El generador produce todas las dependencias necesarias para la correcta ejecución del servicio. El generador produce un caso de uso que permite validar el correcto funcionamiento del servicio generado.

Por otra parte, el generador solo soporta tipos de dato String, Number y Boolean, podría ser extendido mediante un JsonSchema para aceptar tipos de datos y objetos complejos. La capa de persistencia de los servicios generados es mongoDB, es decir, están acoplados a este tipo de tecnología,

podría ser extendido para soportar distintas tecnologías como capa de persistencia. El generador solo produce servicios en Go y NODE, podría ser extendido para generar servicios en Python. El generador podría utilizar algún framework de testing para generar más pruebas para los servicios.

4.4. Pruebas

En esta Sección, se plantean distintos escenarios con el objetivo de mostrar el comportamiento del generador ante distintos casos. Se realizará un total de 4 escenarios distintos. El código generado formará parte de la entrega.

4.4.1. Escenario 1: People Service

En este escenario se va a generar el siguiente servicio:

Listing 4.2: Ejemplo Entrada Generador: people.json

```
1 [{  
2   "name": "People for your bushiness",  
3   "service": "/people",  
4   "url": "http://localhost",  
5   "port": "8080",  
6   "body": {  
7     "name": "John Smith",  
8     "company": "CesarCorp",  
9     "job": "develop",  
10    "city": "CDMX"  
11  }  
12 }]
```

Básicamente consiste en una API para disponibilizar información sobre los empleados de una empresa. Como se puede observar, el modelo de negocio de este servicio está formado íntegramente por strings. En este escenario el análisis se centrará en mostrar el código GO generado, su ejecución y posteriormente validación con el propio test generado. Para ello, es necesario guardar la entrada arriba indicada en un fichero denominado people.json dentro del directorio input/ del proyecto. Una vez establecida la entrada en el directorio necesario, es necesario ejecutar el generador usando el ejecutable adecuado para la plataforma sobre la que se realicen las pruebas, en este caso MAC:

Listing 4.3: Ejemplo Entrada Generador: people.json

```
1 \ $ ./restandtestgenerator.darwin
```

Tras la ejecución, en el directorio output/ se ha generado el siguiente contenido:

```
→ output git:(master) x tree -L 2
.
├── file.txt
├── goservice_people
│   ├── controller
│   ├── go.mod
│   ├── go.sum
│   ├── main.go
│   ├── models
│   └── mongo
├── nodeservice_people
│   ├── README.md
│   ├── api
│   ├── package.json
│   ├── server.js
└── test_people.sh
```

Figura 4.1: Servicio generado people.json

Como se puede observar, se ha generado el servicio Go, el servicio NODE y el test_case.sh. En este escenario se va a poner el foco en goservice_people. La estructura generada es la siguiente:

```
→ goservice_people git:(master) x tree -L 2
.
├── controller
│   ├── controller.go
│   └── router.go
├── go.mod
├── go.sum
├── main.go
├── models
│   └── bussinessObject.go
├── mongo
│   ├── session.go
│   └── storer.go
└── 3 directories, 8 files
```

Figura 4.2: Servicio generado people.json

Se pueden apreciar los siguientes paquetes:

1. controller: Encargado de todo lo relacionado con la API del servicio
2. models: Donde el modelo de negocio toma entidad
3. mongo: Paquete responsable de la comunicación con la capa de persistencia
4. main.go: Programa principal
5. ficheros go.mod y go.sum: Lugar en el que se especifican las dependencias del servicio

Para ejecutar el servicio generado /people seria necesario:

Listing 4.4: Ejemplo Ejecución /people

```

1 \ $ cd output/goservice_people
2 \ $ go run main.go

```

```

/Users/cesarhugo.barzano/Desktop/CesarCorp/I+D/restandtestgenerator
➔ restandtestgenerator git:(master) ✗ cd output/goservice_people
➔ goservice_people git:(master) ✗ ll
total 24
drwxr-xr-x  4 cesarhugo.barzano  staff   128  5 jun 16:51 controller
-rw-r--r--  1 cesarhugo.barzano  staff   167  5 jun 16:51 go.mod
-rw-r--r--  1 cesarhugo.barzano  staff   541  5 jun 16:51 go.sum
-rwxr-xr-x  1 cesarhugo.barzano  staff   778  5 jun 16:51 main.go
drwxr-xr-x  3 cesarhugo.barzano  staff    96  5 jun 16:51 models
drwxr-xr-x  4 cesarhugo.barzano  staff   128  5 jun 16:51 mongo
➔ goservice_people git:(master) ✗ go run main.go
go: downloading github.com/gorilla/handlers v1.4.0
go: downloading github.com/gorilla/mux v1.7.1
go: extracting github.com/gorilla/handlers v1.4.0
go: extracting github.com/gorilla/mux v1.7.1
$PORT must be set
API End-points::
2019/06/05 17:07:40 Index
2019/06/05 17:07:40 AddBusinessObject
2019/06/05 17:07:40 UpdateBusinessObject
2019/06/05 17:07:40 GetBusinessObject
2019/06/05 17:07:40 DeleteBusinessObject
2019/06/05 17:07:40 SearchBusinessObject
Listening on ::8080

```

Figura 4.3: Ejemplo Ejecución /people

De manera adicional, el generador ha producido el siguiente test_people.sh para el escenario:

Listing 4.5: test_people.sh

```

1 #!/usr/bin/env bash
2
3 TestStep_1() {
4     echo "----- Test Step - 1 -----"
5     echo "TEST STEP - 1 "
6     echo "API NAME: People for your bushiness"
7     echo "URL: http://localhost:8080/people"
8     echo "Scripting field..."
9     #<<SCRIPT_PLACEHOLDER>>
10
11     response_code=$(curl -XPOST -i -k "http://localhost:8080/people" --
12         write-out %{http_code} --output /dev/null -d '{"city":"CDMX",
13             company":"CesarCorp","job":"develop","name":"John Smith"}' )
14
15     if [ $response_code = "201" ]; then
16         echo "STEP - 1: PASS"
17         echo "-----"
18         return 0
19     else
20         echo "STEP - 1: FAIL"
21         return 1
22     fi
23 }
24
25 TestStep_2() {
26     echo "----- Test Step - 2 -----"

```



```
26 echo "TEST STEP - 2 "
27 echo "API NAME: People for your bushiness"
28 echo "URL: http://localhost:8080/people"
29 echo "Scripting field..."
30 get=$(curl -sb -H "Accept: application/json" "http://localhost:8080/
    people" | jq '.[0]._id')
31 export ID=$get
32
33 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
    dev/null http://localhost:8080/people)
34
35 if [ $response_code = "200" ]; then
36     echo "STEP - 2: PASS"
37     echo "-----"
38     return 0
39 else
40     echo "STEP - 2: FAIL"
41     return 1
42 fi
43
44 }
45
46 TestStep_3() {
47 echo "----- Test Step - 3 -----"
48 echo "TEST STEP - 3 "
49 echo "API NAME: People for your bushiness"
50 echo "URL: http://localhost:8080/people"
51 echo "Scripting field..."
52 #<<SCRIPT_PLACEHOLDER>>
53
54 response_code=$(curl -XPUT -i -k "http://localhost:8080/people/${ID}
    /\\"" --write-out %{http_code} --output /dev/null -d '{"city":
    "CDMX_Update","company":"CesarCorp_Update","job":"develop_Update",
    name":"John Smith_Update"}')
55
56 if [ $response_code = "200" ]; then
57     echo "STEP - 3: PASS"
58     echo "-----"
59     return 0
60 else
61     echo "STEP - 3: FAIL"
62     return 1
63 fi
64
65 }
66
67 TestStep_4() {
68 echo "----- Test Step - 4 -----"
69 echo "TEST STEP - 4 "
70 echo "API NAME: People for your bushiness"
71 echo "URL: http://localhost:8080/people"
72 echo "Scripting field..."
73 echo $ID
74
75 response_code=$(curl -XDELETE -i -k --write-out %{http_code} --output
    /dev/null http://localhost:8080/people/${ID}/\")
76
77 if [ $response_code = "200" ]; then
78     echo "STEP - 4: PASS"
79     echo "-----"
80     return 0
81 else
```

```

82     echo "STEP - 4: FAIL"
83     return 1
84 fi
85
86 }
87
88 TEST_PASS=0
89 TEST_FAIL=0
90 TOTAL_TEST=0
91
92 declare -a arr=("TestStep_1" "TestStep_2" "TestStep_3" "TestStep_4" )
93
94 for i in "${arr[@]}"
95 do
96     if $i; then
97         TEST_PASS=$((TEST_PASS+1));
98     else
99         TEST_FAIL=$((TEST_FAIL+1));
100 fi
101 done
102
103 echo "--- TEST CASE REPORT ---"
104 echo "TEST PASS: " $TEST_PASS
105 echo "TEST FAIL: " $TEST_FAIL
106 echo "TOTAL EXECUTED: " ${#arr[@]}
107 echo "----"

```

Dicho test puede ser ejecutado contra el actual servicio en ejecución de la siguiente manera:

Listing 4.6: exec test_people.sh

```

1 chmod +x test_people.sh
2 ./test_people.sh

```

El resultado de dicha ejecución valida que el servicio generado funciona correctamente:

Figura 4.4: test_people.sh result

4.4.2. Escenario 2: Players Location

En este escenario, se va a generar el servicio `/location` capaz de manejar las posiciones de jugadores en un sistema de coordenadas, de gran utilidad por ejemplo en el back-end de juegos móviles. El objeto de este escenario es mostrar la capacidad del generador para generar objetos de negocio tipo numérico. De manera adicional, en este escenario se hará foco en el servicio NODE generado. El servicio generado en cuestión es:

Listing 4.7: Ejemplo Entrada Generador: `location_api.json`

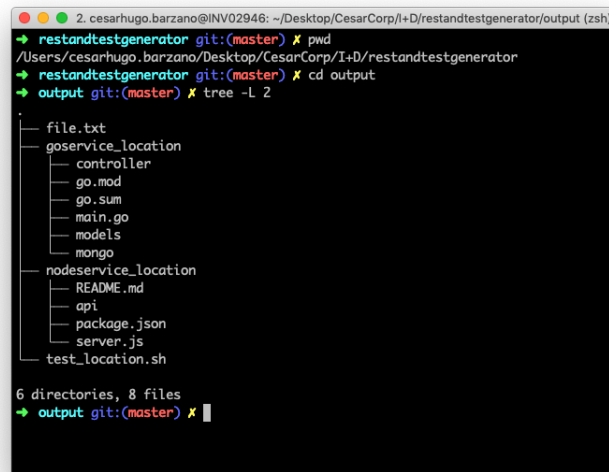
```
1 [{  
2   "name": "Players location",  
3   "service": "/location",  
4   "url": "http://localhost",  
5   "port": ":8080",  
6   "body": {  
7     "player": "suave93",  
8     "x": 67.33,  
9     "y": 45.66,  
10    "z": 5.55  
11   }  
12 }]
```

Para ello, es necesario guardar esta especificación de servicio en un fichero denominado `location_api.json` dentro del directorio `input` y ejecutar el generador:

Listing 4.8: Ejemplo Ejecución Generador: `location_api.json`

```
1 \ $ ./restandtestgenerator.darwin
```

Tras la ejecución, en el directorio `output/` se ha generado el siguiente contenido:



```

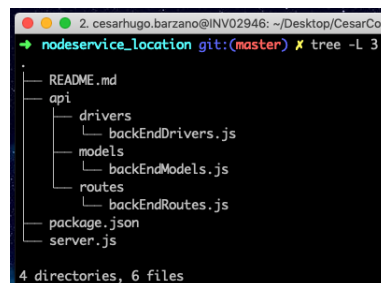
cesarhugo.barzano@INV02946: ~/Desktop/CesarCorp/I+D/restandtestgenerator/output (zsh)
→ restandtestgenerator git:(master) ✗ pwd
/Users/cesarhugo.barzano/Desktop/CesarCorp/I+D/restandtestgenerator
→ restandtestgenerator git:(master) ✗ cd output
→ output git:(master) ✗ tree -L 2
.
├── file.txt
├── goservice_location
│   ├── controller
│   ├── go.mod
│   ├── go.sum
│   ├── main.go
│   ├── models
│   └── mongo
├── nodeservice_location
│   ├── README.md
│   ├── api
│   ├── package.json
│   ├── server.js
│   └── test_location.sh
└── test_location.sh

6 directories, 8 files
→ output git:(master) ✗

```

Figura 4.5: Servicio generado location_api.json

Como se puede observar, se ha generado el servicio Go, el servicio NODE y el test_case.sh. En este escenario se va a poner el foco en nodeservice_location. La estructura generada es la siguiente:



```

cesarhugo.barzano@INV02946: ~/Desktop/CesarCorp/I+D/restandtestgenerator/output (zsh)
→ nodeservice_location git:(master) ✗ tree -L 3
.
├── README.md
├── api
│   ├── drivers
│   │   └── backEndDrivers.js
│   ├── models
│   │   └── backEndModels.js
│   └── routes
│       └── backEndRoutes.js
├── package.json
├── server.js
└── test_location.sh

4 directories, 6 files

```

Figura 4.6: Servicio generado location_api.json

Se pueden apreciar los siguientes modulos:

1. api/drivers: Contiene el código necesario para la comunicación con la capa de persistencia
2. api/models: Contiene el modelo de negocio generado para el servicio
3. routes: Contiene el código de la API del servicio
4. server.js: Fichero principal del servidor de aplicaciones
5. package.json: Ficho de dependencias. Pendiente de inicializar.

Para inicializar el servicio NODE /location seria necesario ejecutar y contestar a las preguntas de npm:

Listing 4.9: Ejemplo inicialización /location

```
1 \ $ npm init
2 {
3   "dependencies": {
4     "body-parser": "~1.18.2",
5     "express": "~4.16.2",
6     "mongoose": "~5.4.15",
7     "package.json": "~2.0.1"
8   },
9   "devDependencies": {
10    "nodemon": "~1.18.10"
11  },
12  "scripts": {
13    "start": "node server.js"
14  },
15  "name": "nodeservice_location",
16  "description": "",
17  "version": "1.0.0",
18  "main": "server.js",
19  "author": "",
20  "license": "ISC"
21 }
```

Para instalar las dependencias que el servicio NODE necesita:

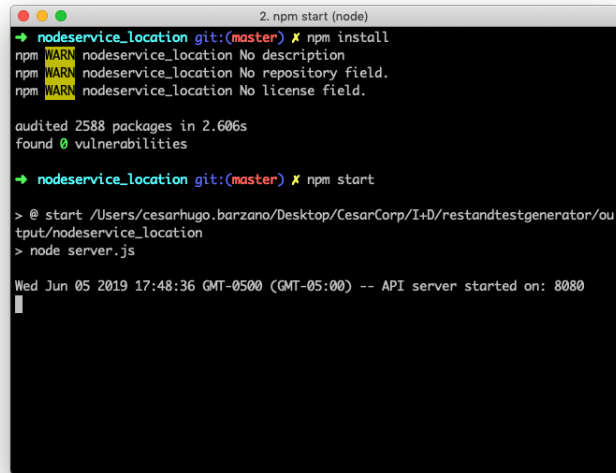
Listing 4.10: Ejemplo instalación /location

```
1 $ npm install
```

Para ejecutar el servicio NODE:

Listing 4.11: Ejemplo Ejecución /location

```
1 $ npm start
```



```

2. npm start (node)
→ nodeservice_location git:(master) ✗ npm install
npm WARN nodeservice_location No description
npm WARN nodeservice_location No repository field.
npm WARN nodeservice_location No license field.

audited 2588 packages in 2.606s
found 0 vulnerabilities

→ nodeservice_location git:(master) ✗ npm start

> @ start /Users/cesarhugo.barzana/Desktop/CesarCorp/I+D/restandtestgenerator/ou
tput/nodeservice_location
> node server.js

Wed Jun 05 2019 17:48:36 GMT-0500 (GMT-05:00) -- API server started on: 8080

```

Figura 4.7: Servicio corriendo /location

De manera adicional, el generador ha producido el siguiente test_location.sh para el escenario:

Listing 4.12: test_location.sh

```

1 #!/usr/bin/env bash
2
3 TestStep_1() {
4 echo "----- Test Step - 1 -----"
5 echo "TEST STEP - 1 "
6 echo "API NAME: Players location"
7 echo "URL: http://localhost:8080/location"
8 echo "Scripting field..."
9 #<<SCRIPT_PLACEHOLDER>>
10
11 response_code=$(curl -XPOST -i -k "http://localhost:8080/location" --
   write-out %{http_code} --output /dev/null -d '{"player":"suave93
   ","x":67.33,"y":45.66,"z":5.55}' )
12
13 if [ $response_code = "201" ]; then
14     echo "STEP - 1: PASS"
15     echo "----- --- -----"
16     return 0
17 else
18     echo "STEP - 1: FAIL"
19     return 1
20 fi
21
22 }
23
24 TestStep_2() {
25 echo "----- Test Step - 2 -----"
26 echo "TEST STEP - 2 "
27 echo "API NAME: Players location"

```

```
28 echo "URL: http://localhost:8080/location"
29 echo "Scripting field..."
30 get=$(curl -sb -H "Accept: application/json" "http://localhost:8080/
    location" | jq '.[0]._id')
31 export ID=$get
32
33 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
    dev/null http://localhost:8080/location)
34
35 if [ $response_code = "200" ]; then
36     echo "STEP - 2: PASS"
37     echo "-----"
38     return 0
39 else
40     echo "STEP - 2: FAIL"
41     return 1
42 fi
43
44 }
45
46 TestStep_3() {
47     echo "----- Test Step - 3 -----"
48     echo "TEST STEP - 3 "
49     echo "API NAME: Players location"
50     echo "URL: http://localhost:8080/location"
51     echo "Scripting field..."
52     #<<SCRIPT_PLACEHOLDER>>
53
54     response_code=$(curl -XPUT -i -k "http://localhost:8080/location/${ID}
        /\\" --write-out %{http_code} --output /dev/null -d '{"player
        ":"suave93_Update","x":68.33,"y":46.66,"z":6.55}' )
55
56     if [ $response_code = "200" ]; then
57         echo "STEP - 3: PASS"
58         echo "-----"
59         return 0
60     else
61         echo "STEP - 3: FAIL"
62         return 1
63     fi
64
65 }
66
67 TestStep_4() {
68     echo "----- Test Step - 4 -----"
69     echo "TEST STEP - 4 "
70     echo "API NAME: Players location"
71     echo "URL: http://localhost:8080/location"
72     echo "Scripting field..."
73     echo $ID
74
75     response_code=$(curl -XDELETE -i -k --write-out %{http_code} --output
        /dev/null http://localhost:8080/location/${ID}/\")
76
77     if [ $response_code = "200" ]; then
78         echo "STEP - 4: PASS"
79         echo "-----"
80         return 0
81     else
82         echo "STEP - 4: FAIL"
83         return 1
84     fi
85 }
```

```

85 }
86 }
87
88 TEST_PASS=0
89 TEST_FAIL=0
90 TOTAL_TEST=0
91
92 declare -a arr=("TestStep_1" "TestStep_2" "TestStep_3" "TestStep_4" )
93
94 for i in "${arr[@]}"
95 do
96     if $i; then
97         TEST_PASS=$((TEST_PASS+1));
98     else
99         TEST_FAIL=$((TEST_FAIL+1));
100     fi
101 done
102
103 echo
104 echo
105
106 echo "--- TEST CASE REPORT ---"
107 echo "TEST PASS: " $TEST_PASS
108 echo "TEST FAIL: " $TEST_FAIL
109 echo "TOTAL EXECUTED: " ${#arr[@]}
110 echo "----"

```

Dicho test puede ser ejecutado contra el actual servicio en ejecución de la siguiente manera:

Listing 4.13: exec test_location.sh

```

1 chmod +x test_location.sh
2 ./test_location.sh

```

El resultado de dicha ejecución valida que el servicio generado funciona correctamente:

The figure consists of two side-by-side terminal windows. The left window shows the process of installing a Node.js package and starting a server. The right window shows the output of a test script, which includes a table of test results and a final report.

Left Terminal Output:

```

2. npm start (node)
→ node-service_location git:(master) ✗ npm install
npm WARN node-service_location No description
npm WARN node-service_location No repository field.
npm WARN node-service_location No license field.
audited 2588 packages in 2.048s
found 0 vulnerabilities
→ node-service_location git:(master) ✗ npm start
> @ start /Users/cesarhugo.barzano/Desktop/CesarCorp/I+D/restandtestgenerator/ou
tput/node-service_location
> node server.js

Wed Jun 05 2019 17:55:19 GMT-0500 (GMT-05:00) -- API server started on: 8080
POST MODEL::
LIST MODELS::
LIST MODELS::
PUT MODEL::
DELETE MODEL::

```

Right Terminal Output:

```

3. cesarhugo.barzano@INV02946: ~/Desktop/CesarCorp/I+D/restandtestgenerator/output (zsh)
Scripting field...
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 112 100 56 100 56 687 687 ---:---:---:---: 691
STEP - 3: PASS
----- Test Step - 4 -----
TEST STEP - 4
API NAME: Players location
URL: http://localhost:8080/location
Scripting field...
"Scd7c894d154b82f635186f7"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 40 100 40 0 0 550 0 ---:---:---:---: 555
STEP - 4: PASS
-----
--- TEST CASE REPORT ---
TEST PASS: 4
TEST FAIL: 0
TOTAL EXECUTED: 4
-----
→ output git:(master) ✗

```

Figura 4.8: test_location.sh result

Es interesante tener en cuenta que los códigos de estado devueltos por el protocolo HTTP son los que marcan el comportamiento de las peticiones

atendidas por el servicio web y son usados por el test case para validar el resultado.


HTTP Status Codes		
	1xx - Informational	>
	2xx - Success	>
	3xx - Redirection	>
	4xx - Client Error	>
	5xx - Server Error	>

Figura 4.9: HTTP Status Codes

4.4.3. Escenario 3: Miauuu as a service

En este escenario se va a generar el siguiente servicio:

Listing 4.14: cats_cloud_service.json

```
1 [{  
2   "name": "Miauuu as a service",  
3   "service": "/cats",  
4   "url": "http://localhost",  
5   "port": "8080",  
6   "body": {  
7     "name": "Wurst",  
8     "alias": "Mi bolita gatita bonita, la reina",  
9     "reina": true,  
10    "age": 1.5,  
11    "color": "Negro como mi alma"  
12  }  
13 }]
```

El objetivo de este escenario es mostrar el modelo de negocio generado tanto en GO como en NODE. Para generar el servicio es necesario guardar en el directorio input/ la anterior especificación en un fichero denominado cats_cloud_service.json y ejecutar:

Listing 4.15: Ejemplo Ejecución Generador: cats_cloud_service.json

```
1 \ $ ./restandtestgenerator.darwin
```

Dicha ejecución produce el siguiente código:

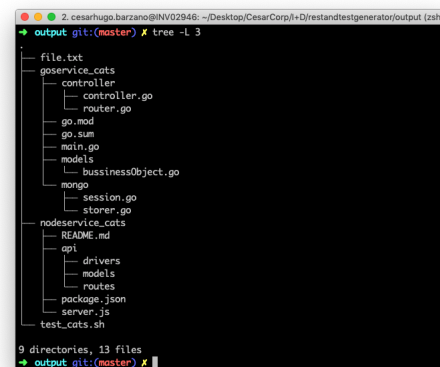


Figura 4.10: Servicios /cats generados

Navegando hasta `output/goservice_cats/models/bussinessObject.go` se puede observar el modelo de negocio generado para el servicio en GO:

Listing 4.16: Ejemplo Modelo de negocio GO

```

1 type BusinessObject struct {
2     ID      string      'bson:"_id" json:"_id,omitempty"'
3     Alias   string      'bson:"alias" json:"alias,omitempty"'
4     Reina   bool        'bson:"reina" json:"reina,omitempty"'
5     Age     float64     'bson:"age" json:"age,omitempty"'
6     Color   string      'bson:"color" json:"color,omitempty"'
7     Name    string      'bson:"name" json:"name,omitempty"'
8 }

```

Navegando hasta `output/nodeservice_cats/api/models/backEndModels.js` se puede observar el modelo de negocio generado para el servicio en NODE:

Listing 4.17: Ejemplo Modelo de negocio NODE

```

1 var ModelSchema = new Schema ({
2   alias : String,
3   age   : Number,
4   color : String,
5   name  : String,
6 });

```

4.4.4. Escenario 4: Múltiples Servicios

En este escenario se va a mostrar la capacidad del generador para producir múltiples servicios. Para ello, dentro del directorio `input/` se van a guardar las especificaciones de servicio usadas en los ejemplos anteriores junto con la siguiente servicio. Dicho servicio ejemplifica la posibilidad de usar un servicio web para controlar la asistencia de alumnos a cierta asignatura y el uso del tipo de dato boolean.

Listing 4.18: Ejemplo definición servicio class_attendance.json

```
1 [{
2   "name": "Attendance service",
3   "service": "/student",
4   "url": "http://localhost",
5   "port": ":8080",
6   "body": {
7     "name": "Cesar Hugo",
8     "attendance": true
9   }
10 }]
```

El directorio de entrada quedaría con el siguiente ficheros de especificación:

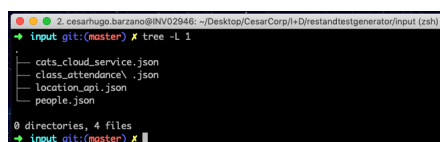


Figura 4.11: Múltiples Inputs

Ejecutar el generador:

Listing 4.19: Ejemplo Ejecución Generador: múltiples inputs

```
1 \ $ ./restandtestgenerator.darwin
```

Finalmente se genera un servicio NODE, un servicio GO y un caso de prueba.sh para cada uno de los servicios.



Figura 4.12: Múltiples Outputs

Capítulo 5

Entrega

En este capítulo se detallan cada uno de los ficheros/directorios que forman parte de la entrega.

5.1. Generador

El generador propuesto por el enunciado se compone de una serie de paquetes GO alojados dentro de testgenerator/src. El generador ha sido desarrollado sobre Ubuntu.18 y hace uso de la versión Go 1.10 pero al ser un lenguaje compilado, podemos elegir entre ejecutar el código fuente (necesario tener GO instalado en el sistema o utilizar el ejecutable generado con la compilación).

5.2. Makefile

El fichero "makefile" establece las distintas ejecuciones de pruebas para el generador.

5.3. TI_CesarHugoBarzanoCruz.pdf

Memoria de la práctica, referencia a este documento en si mismo, alojado en el directorio raíz de la entrega.

5.4. Directorio DOC

Directorio donde se almacenan todos los fuentes usados para generar esta documentación utilizando LaTeX. Incluye tambien las imagenes usadas en la memoria.

5.5. Directorio src/testgenerator/input

Directorio donde se almacenan todos los ficheros usados como configuración para el generador.

5.6. Directorio src/testgenerator/output

Directorio donde se almacenan todos los ficheros resultados de la ejecución de la pruebas.

Bibliografía

- [1] GO, *The GO Programming Language* <https://golang.org/>
- [2] POSTMAN, *POSTMAN API DEVELOPMENT* <https://www.getpostman.com/>
- [3] SELENIUM, *Selenium Browser Automation* <https://www.seleniumhq.org/>
- [4] SCRUM, *Scrum Process* <https://winred.es/management/metodologia-scrum-que-es/gmx-niv116-con24594.htm>
- [5] KANBAN, *Kamban Table* <http://www.itmplatform.com/es/blog/kanban-por-que-es-agil-y-en-que-aventaja-a-scrum>
- [6] LEAN, *Lean Process* <https://centricconsulting.com/business-consulting/improve-operational-performance/business-process-improvement-lean-six-sigma>
- [7] KENT BECK , *Kent Beck - Extreme programming explained: embrace change -Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©2000 ISBN:0-201-61641-6* <https://dl.acm.org/citation.cfm?id=318762>
- [8] XP, *Basic Extreme Programming Practices* <https://ronjeffries.com/xprog/what-is-extreme-programming>
- [9] MICRO-SERVICES ARC, *Arquitectura basada en microservicios* <https://openwebinars.net/blog/diferencia-entre-arquitectura-monolitica-y-microservicios>
- [10] CHARLES ANTONY HOARE, *Communicating sequential processes* <https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf>
- [11] JAVASCRIPT, <https://www.javascript.com>
- [12] NODE JS, *JavaScript runtime built on Chrome's V8 JavaScript engine.* <https://nodejs.org/es>

- [13] SHELL BASH, *Bash Reference Manual* <https://www.gnu.org/software/bash/manual/bash.html>
- [14] CURL, *command line tool and library for transferring data with URLs* <https://curl.haxx.se/>

Capítulo 6

Anexo