



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
3.0.1. Validación Software	11
3.1. Tecnicas Usadas	12
4. La Solución	13
4.1. El Generador	13
4.2. Tecnologías Utilizadas	13
4.3. Especificación de la Solución	14
4.4. Pruebas	16
5. Entrega	25
5.1. Generador	25
5.2. Makefile	25
5.3. TI.CesarHugoBarzanoCruz.pdf	25
5.4. Directorio DOC	25
5.5. Directorio src/testgenerator/input	26
5.6. Directorio src/testgenerator/output	26
6. Anexo	29
6.1. makefile	29

Índice de figuras

3.1. Método V	11
4.1. Método V	14
4.2. TestCase1	17
4.3. RUN TestCase1	17
4.4. RUN TestCase2	18

Capítulo 1

Resumen

En la actualidad, 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 incubiertas 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, siguiendo la metodología de desarrollo en V en la cual profundizaremos más adelante. El desarrollo de estos proyectos conlleva la ejecución de lo denominado como "Plan de Validación" que si mismo es un proyecto de grandes proporciones que acompaña al proyecto principal durante todas sus etapas. El plan de validación de un proyecto establece el camino o manera en la que se valida el correcto funcionamiento

del proyecto final, es decir, asegurar al cliente de que todos los requisitos planteados en fase de diseño se cumplen, produciendo así un sistema final fiable.

El plan de validación en terminos de coste y tiempo tiene una proporciones considerables principalmente por que el proceso de refinamiento es lento, por lo que toman vital inportancia la automatización de pruebas y procedimiento, obteniendo una ganacia en hombre/tiempo significativa. Si quisieramos aplicar procedimientos similares al desarrollo de aplicaciones web basadas en metodologías ágiles, seria necesario encontrar el termino medio entre un desarrollo ágil y un plan de validación que asegure el correcto funcionamiento de estos servicios de información. Debido a esto, el objetivo de esta investigación es el de intentar aplicar técnicas de validación tradicionales a procesos de desarrollo ágil, donde la generación automática de código desempeña un papel fundamental para la automatización de dichas pruebas.

El resultado de esta investigación serán la generación automática de las pruebas necesarias para validar que aplicaciones de caracter web desarrolladas de manera ágil funcionan de la manera esperada, sin que dicho proceso de testeo suponga un pronblema en terminos de tiempo o dinero.

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. Esta demanda en ocasiones puede ocasionar ciertos problemas a la hora de ofrecer servicios de información 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. Disponibilizar servicios no validados puede suponer una mala toma de decisiones pues el usuario medio asume la disponibilidad de estos o incluso la veracidad de la información proporcionada.

El problema radica en que las metodologías de desarrollo para este tipo de servicios normalmente son de carácter ágil y los servicios expuestos normalmente no han superado un plan de validación adecuado ya que esto supone un incremento económico en el proceso de desarrollo y una notable dilatación en el tiempo de producción de nuevos releases debido a que la validación y verificación software mediante un enfoque tradicional es un proceso lento y pesado.

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

En el desarrollo de servicios web y sus interfaces, normalmente mediante REST y SOAP, es muy común encontrar problemas relacionados con la pérdida de servicio o la seguridad. Muchos de estos problemas son fruto de un desarrollo apresurado o puesta en producción de servicios de la información que no han pasado por etapas de revisión o certificación. Esto produce que los consumidores de estos servicios, normalmente otros servicios o frontales utilizados por cliente sufran fallos, ya que el canal o flujo de datos que alimenta todo la cadena no cumple lo esperado o las interfaces no son compliant con lo prometido a terceros. Estos problemas surgen a raíz de una mala validación de estos servicios.

3.0.1. Validación Software

La validación[4] software es un proceso de refinamiento en el que se garantiza el correcto funcionamiento del sistema validado, garantizando el correcto diseño e implementación de los requisitos planteados para el proyecto. Es un proceso largo y costoso con las dimensiones de un proyecto en si mismo donde se establece un plan de validación formado de campañas de verificación donde se establece la manera en la que se certifica el funcionamiento del software validado en cada una de las etapas del proyecto. La validación software normalmente sigue la metodología en V, donde a cada etapa del proyecto, le acompaña una etapa del plan de validación:

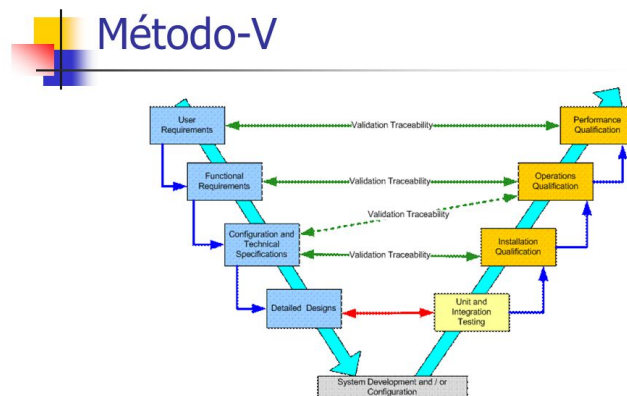


Figura 3.1: Método V

En terminos de tiempo y costes, este proceso es muy costoso por lo que se intenta automatizar en la medida de lo posible todas las tareas de testing asociadas a la etapa del desarrollo en la que el proyecto se encuentra.

El plan de validación en si, trata de establecer estas etapas del proyecto, definiendo los aspectos a cubrir en cada una de las entregas software establecidas por calendario. Cada entrega esta formada por una serie de casos de prueba o TestCases que aglutinan un conjunto de requisitos o funcionalidades a validar. Normalmente este conjunto es de alto nivel, por lo que los casos de prueba a su vez estan formados por una serie de TestSteps o pasos para validar cada uno de los requisitos de alto nivel definidos para el sistema. Cada TestSteps esta formado por uno o varios criterios PASS/FAIL. Esta cadena puede extenderse hacia arriba, tantos niveles como complejidad tenga el software que se intenta validar. Si trasladamos estas tecnicas al proceso de generación de servicios web, el coste en terminos de tiempo y dinere para el desarrollo ágil de servicios se vería altamente encarecido.

3.1. Tecnicas Usadas

Actualmente, para solventar este tipo de problemas aplicados a los servicios de la información existen metodologias como el test driven development o desarrollo basado en test donde primero se desarrollan los casos de prueba y despues el software necesario para pasar dichas pruebas. Tambien existen herramientas como Postman[5] para testear las interfaces expuestas a los frontales u otros servicios o SeleniumHQ[4] para la automatización desde navegador. El problema de usar tecnologías como estas radica en la lentitud de las mismas, ya que apesar de ser muy potentes y perseguir la automatiza-

ción, el tiempo de especificación de lo que se quiere ejecutar es muy elevado, incluyendo restricciones propias que por contrato y base-line tecnológico imponen los diversos proyectos, limitando las herramientas aceptadas para la validación de los mismos.

Capítulo 4

La Solución

En vista de los problemas ocasionados por una mala validación de los servicios de información expuestos y debido a las carencias que tienen las herramientas actuales para el testing de servicios web, se propone la creación de un generador de pruebas para las interfaces de ciertos servicios web con sus consumidores o aplicaciones cliente.

4.1. El Generador

Se propone la implementación de un generador cuyas entradas sean ficheros de especificación para API-REST como RAML, JSON, XML y su salida sean los distintos TestCases que se especifiquen como necesarios para validar el correcto funcionamiento del servicio web en cuestión. Dichos TestCases estaran formados por una serie de pasos o TestSteps con la especificación atómica que se ha de validar en ese paso, donde el resultado de la ejecución de este paso sea del criterio tipo pasa/no pasa.

El generador resultante ha de entenderse como una herramienta propia para el desarrollo, ya que permitirá al desarrollador validar de forma rápida que el servicio en cuestión esta siendo construido de manera adecuada y que las interfaces expuestas efectivamente cumplen con lo deseado.

4.2. Tecnologías Utilizadas

Como tecnología base, se ha decidido utilizar el lenguaje de programación GO[1], versión 1.10 debido a la eficiencia y facilidad que dicho lenguaje proporciona para el procesamiento de cadenas de caracteres. Como tecnología de salida, el generador es capaz de producir casos de prueba en bash[3] shell script. Se ha decido generar shell script debido a su alta eficiencia, ya que todo lo construido son sentencias interpretadas por el propio sistema operativo. Esto es altamente beneficioso a la hora de ejecutar grandes conjuntos de pruebas. Al generar código capaz de ser interpretado por el sistema

operativo, no es necesaria la instalación de librerías de terceros o software adicional, lo cual es ideal para mantener la integridad de los entornos de desarrollo. Al tratarse de pruebas para las interfaces expuestas por los servicios de la información, si contamos con una máquina de pruebas con salida a internet capaz de consumir dichos servicios web, las pruebas también podrían ser ejecutadas de manera remota y así obtener métricas sobre el estado de la red.

El sistema operativo base sobre el que se han realizado las pruebas es Ubuntu[2] 18.04 LTS atacando a una serie de servicios web que se ejecutan en el mismo sistema operativo pero dentro de contenedores o dockers.

Adicionalmente 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

4.3. Especificación de la Solución

Al hablar de servicios web, es necesario tener en cuenta que estos hacen uso del protocolo HTTP por lo que los códigos de estado devueltos por el protocolo son los que marcan el comportamiento de las peticiones atendidas por el servicio web.

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

Figura 4.1: Método V

Por ello, el generador recibirá como entrada un fichero de especificación de api formado por entradas como la siguiente, donde establecer los atributos necesarios para validar que esta interfaz del servicio web es correcta.

Listing 4.1: Entrada Unitaria Configuración Generador

```

1 [{
2   "name": "basic web app",
3   "route": "/",
4   "http_verb": "GET",

```



```
5  "url": "http://localhost",
6  "port": ":8080",
7  "body": "",
8  "expected_code": "200"
9  ]]
```

Para cada uno de estas entradas, el generador producirá el siguiente TestStep:

Listing 4.2: Test Step

```
1 TestStep_0() {
2 echo "----- Test Step - 0 -----"
3 echo "TEST STEP - 0 "
4 echo "API NAME: basic web app"
5 echo "URL: http://localhost:8080/"
6
7 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
8   dev/null http://localhost:8080/)
9
10 if [ $response_code = "200" ]; then
11   echo "STEP - 0: PASS"
12   echo "----- --- ----"
13   return 0
14 else
15   echo "STEP - 0: FAIL"
16   return 1
17 fi
18 }
```

Con el conjunto de TestSteps que el generador produzca, se creará un TestCase como el que se muestra a continuación:

Listing 4.3: Test Case

```
1  #!/usr/bin/env bash
2
3  echo
4  TestStep_0() {
5  echo "----- Test Step - 0 -----"
6  echo "TEST STEP - 0 "
7  echo "API NAME: basic web app"
8  echo "URL: http://localhost:8080/"
9
10 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
11   dev/null http://localhost:8080/)
12
13 if [ $response_code = "200" ]; then
14   echo "STEP - 0: PASS"
15   echo "----- --- ----"
16   return 0
17 else
18   echo "STEP - 0: FAIL"
19   return 1
20 fi
21 }
22 echo "----- --- ----"
23 echo
```

```

24 echo
25 TestStep_1() {
26 echo "----- Test Step - 1 -----"
27 echo "TEST STEP - 1 "
28 echo "API NAME: basic web app"
29 echo "URL: localhost:8080/users"
30
31 response_code=$(curl -XPOST -i -k --write-out %{http_code} --output /
    dev/null localhost:8080/users)
32
33 if [ $response_code = "" ]; then
34     echo "STEP - 1: PASS"
35     echo "----- --- -----"
36     return 0
37 else
38     echo "STEP - 1: FAIL"
39     return 1
40 fi
41 }
42 echo "----- --- -----"
43 echo
44
45 TEST_PASS=0
46 TEST_FAIL=0
47 TOTAL_TEST=0
48 declare -a arr=("TestStep_0" "TestStep_1" )
49
50 for i in "${arr[@]}"
51 do
52     if $i; then
53         TEST_PASS=$((TEST_PASS+1));
54     else
55         TEST_FAIL=$((TEST_FAIL+1));
56     fi
57 done
58
59 echo
60 echo
61 echo "--- TEST CASE REPORT ---"
62 echo "TEST PASS: " $TEST_PASS
63 echo "TEST FAIL: " $TEST_FAIL
64 echo "TOTAL EXECUTED: " ${#arr[@]}
65 echo "----- --- -----"

```

Este TestCase será el encargado de ejecutar todos los Test Steps, dando un informe del estado de la API, o APIS que se esten probando con la configuración de entrada.

4.4. Pruebas

Con el Objetivo de mostrar la potencia del generador, se van a realizar una serie de ejecuciones de prueba para mostrar los posibles escenarios del generador. Para ello es necesario disponer de un servicio web sobre el que generar y ejecutar los casos de prueba generados. En este caso, disponemos de un servicio de web basado contenedores ejecutalo localmente, los fuentes del servicio web usado para testear pueden encontrarse aquí

<https://github.com/hugobarzano/DataRestful>

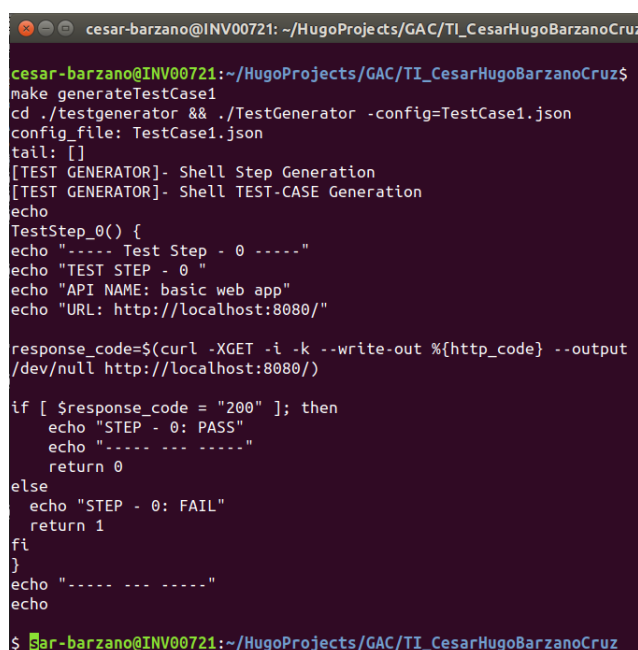
TestCase1

El primer caso de prueba, utilizará como configuracion un solo end-point de api y puede ser generado mediante **make generateTestCase1**

Listing 4.4: TestCase1.json

```
1 [{
2   "name": "basic web app",
3   "route": "/",
4   "http_verb": "GET",
5   "url": "http://localhost",
6   "port": ":8080",
7   "body": "",
8   "expected_code": "200"
9 }
10 ]
```

El generador producirá la siguiente salida:



```
cesar-barzano@INV00721: ~/HugoProjects/GAC/TI_CesarHugoBarzanoCruz
cesar-barzano@INV00721:~/HugoProjects/GAC/TI_CesarHugoBarzanoCruz$
make generateTestCase1
cd ./testgenerator && ./TestGenerator -config=TestCase1.json
config_file: TestCase1.json
tail: []
[TEST GENERATOR]- Shell Step Generation
[TEST GENERATOR]- Shell TEST-CASE Generation
echo
TestStep_0() {
echo "----- Test Step - 0 -----"
echo "TEST STEP - 0 "
echo "API NAME: basic web app"
echo "URL: http://localhost:8080/"

response_code=$(curl -XGET -i -k --write-out %{http_code} --output
/dev/null http://localhost:8080/)

if [ $response_code = "200" ]; then
echo "STEP - 0: PASS"
echo "-----"
return 0
else
echo "STEP - 0: FAIL"
return 1
fi
}
echo "-----"
echo
```

Figura 4.2: TestCase1

Una vez generado el primer caso de prueba y con el servicio web corriendo, podemos ejecutarlo mediante **make runTestCase1** produciendo el siguiente resultado:

El generador producirá la siguiente salida:

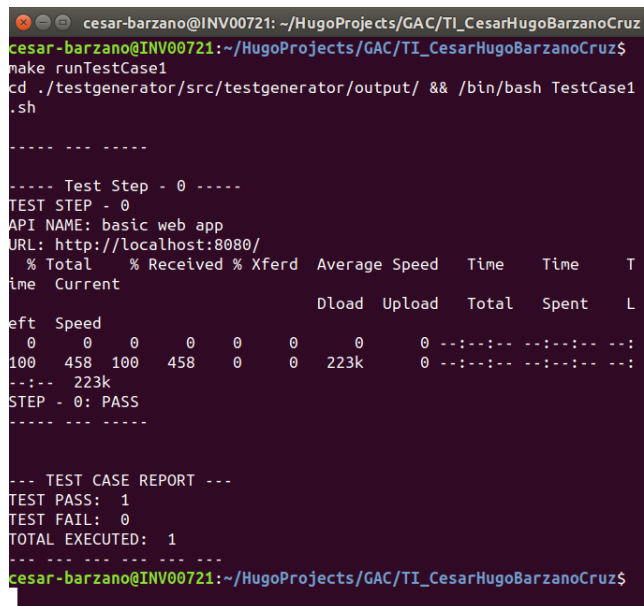


Figura 4.3: RUN TestCase1

Test Case 2

El segundo caso de prueba, utilizará como configuracion un 2 end-point de api para disintos recursos y puede ser generado mediante **make generateTestCase2**

Listing 4.5: TestCase2.json

```
1 [{
2   "name": "Datarestful - Index",
3   "route": "/",
4   "http_verb": "GET",
5   "url": "http://localhost",
6   "port": ":8080",
7   "body": "",
8   "expected_code": "200"
9 },
10 {
11   "name": "Datarestful",
12   "route": "/Services/",
13   "http_verb": "GET",
14   "url": "http://localhost",
15   "port": ":8080",
16   "body": "",
17   "expected_code": "200"
18 }
19 ]
```

Una vez generado el segundo caso de prueba y con el servicio web corriendo, podemos ejecutarlo mediante **make runTestCase2** produciendo el siguiente resultado:

```

cesar-barzano@INV00721: ~/HugoProjects/GAC/TI_CesarHugoBarzanoCruz
API NAME: Datarestful - Index
URL: http://localhost:8080/
% Total % Received % Xferd Average Speed Time Time T
ime Current
Dload Upload Total Spent L
eft Speed
0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:
100 458 100 458 0 0 223k 0 --:--:-- --:--:-- --:
--- -- 223k
STEP - 0: PASS
-----
----- Test Step - 1 -----
TEST STEP - 1
API NAME: Datarestful
URL: http://localhost:8080/Services/
% Total % Received % Xferd Average Speed Time Time T
ime Current
Dload Upload Total Spent L
eft Speed
0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:
100 361 100 361 0 0 176k 0 --:--:-- --:--:-- --:
--- -- 176k
STEP - 1: PASS
-----

--- TEST CASE REPORT ---
TEST PASS: 2
TEST FAIL: 0
TOTAL EXECUTED: 2
-----
cesar-barzano@INV00721:~/HugoProjects/GAC/TI_CesarHugoBarzanoCruz$

```

Figura 4.4: RUN TestCase2

TestCase3

El tercer caso de prueba, utilizará como configuracion un muchos end-point de api para disintos recursos y puede ser generado mediante **make generateTestCase3**

Listing 4.6: TestCase3.json

```

1 [{
2   "name": "Datarestful - Index",
3   "route": "/",
4   "http_verb": "GET",
5   "url": "http://localhost",
6   "port": ":8080",
7   "body": "",
8   "expected_code": "200"
9 },
10 {
11   "name": "Datarestful",
12   "route": "/Services/",
13   "http_verb": "GET",
14   "url": "http://localhost",
15   "port": ":8080",
16   "body": "",
17   "expected_code": "200"
18 },
19 {
20   "name": "Datarestful - Index",
21   "route": "/",
22   "http_verb": "GET",

```

```

23     "url": "http://localhost",
24     "port": ":8080",
25     "body": "",
26     "expected_code": "200"
27 },
28 {
29     "name": "Datarestful",
30     "route": "/Services/",
31     "http_verb": "GET",
32     "url": "http://localhost",
33     "port": ":8080",
34     "body": "",
35     "expected_code": "200"
36 },{
37     "name": "Datarestful - Index",
38     "route": "/",
39     "http_verb": "GET",
40     "url": "http://localhost",
41     "port": ":8080",
42     "body": "",
43     "expected_code": "200"
44 },
45 {
46     "name": "Datarestful",
47     "route": "/Services/",
48     "http_verb": "GET",
49     "url": "http://localhost",
50     "port": ":8080",
51     "body": "",
52     "expected_code": "200"
53 },{
54     "name": "Datarestful - Index",
55     "route": "/",
56     "http_verb": "GET",
57     "url": "http://localhost",
58     "port": ":8080",
59     "body": "",
60     "expected_code": "200"
61 },
62 {
63     "name": "Datarestful",
64     "route": "/Services/",
65     "http_verb": "GET",
66     "url": "http://localhost",
67     "port": ":8080",
68     "body": "",
69     "expected_code": "200"
70 }
71 ]

```

Una vez generado el tercer caso de prueba usando **make generateTest-Case3** y con el servicio web corriendo, podemos ejecutarlo mediante **make runTestCase3**. Al tratarse de 7 end-points de api, lo generado y ejecutado mediante make es:

Listing 4.7: TestCase3.sh

```

1 #!/usr/bin/env bash
2
3 echo

```

```
4 TestStep_0() {
5 echo "----- Test Step - 0 -----"
6 echo "TEST STEP - 0 "
7 echo "API NAME: Datarestful - Index"
8 echo "URL: http://localhost:8080/"
9
10 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
    dev/null http://localhost:8080/)
11
12 if [ $response_code = "200" ]; then
13     echo "STEP - 0: PASS"
14     echo "----- --- -----"
15     return 0
16 else
17     echo "STEP - 0: FAIL"
18     return 1
19 fi
20 }
21 echo "----- --- -----"
22 echo
23
24 echo
25 TestStep_1() {
26 echo "----- Test Step - 1 -----"
27 echo "TEST STEP - 1 "
28 echo "API NAME: Datarestful"
29 echo "URL: http://localhost:8080/Services/"
30
31 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
    dev/null http://localhost:8080/Services/)
32
33 if [ $response_code = "200" ]; then
34     echo "STEP - 1: PASS"
35     echo "----- --- -----"
36     return 0
37 else
38     echo "STEP - 1: FAIL"
39     return 1
40 fi
41 }
42 echo "----- --- -----"
43 echo
44
45 echo
46 TestStep_2() {
47 echo "----- Test Step - 2 -----"
48 echo "TEST STEP - 2 "
49 echo "API NAME: Datarestful - Index"
50 echo "URL: http://localhost:8080/"
51
52 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
    dev/null http://localhost:8080/)
53
54 if [ $response_code = "200" ]; then
55     echo "STEP - 2: PASS"
56     echo "----- --- -----"
57     return 0
58 else
59     echo "STEP - 2: FAIL"
60     return 1
61 fi
62 }
```

```

63 echo "----- --- ----"
64 echo
65
66 echo
67 TestStep_3() {
68 echo "----- Test Step - 3 ----"
69 echo "TEST STEP - 3 "
70 echo "API NAME: Datarestful"
71 echo "URL: http://localhost:8080/Services/"
72
73 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
74 dev/null http://localhost:8080/Services/)
75
76 if [ $response_code = "200" ]; then
77     echo "STEP - 3: PASS"
78     echo "----- --- ----"
79     return 0
80 else
81     echo "STEP - 3: FAIL"
82     return 1
83 fi
84 echo "----- --- ----"
85 echo
86
87 echo
88 TestStep_4() {
89 echo "----- Test Step - 4 ----"
90 echo "TEST STEP - 4 "
91 echo "API NAME: Datarestful - Index"
92 echo "URL: http://localhost:8080/"
93
94 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
95 dev/null http://localhost:8080/)
96
97 if [ $response_code = "200" ]; then
98     echo "STEP - 4: PASS"
99     echo "----- --- ----"
100     return 0
101 else
102     echo "STEP - 4: FAIL"
103     return 1
104 fi
105 echo "----- --- ----"
106 echo
107
108 echo
109 TestStep_5() {
110 echo "----- Test Step - 5 ----"
111 echo "TEST STEP - 5 "
112 echo "API NAME: Datarestful"
113 echo "URL: http://localhost:8080/Services/"
114
115 response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
116 dev/null http://localhost:8080/Services/)
117
118 if [ $response_code = "200" ]; then
119     echo "STEP - 5: PASS"
120     echo "----- --- ----"
121     return 0
122 else

```



```
122     echo "STEP - 5: FAIL"
123     return 1
124 fi
125 }
126 echo "----- --- ----"
127 echo
128
129 echo
130 TestStep_6() {
131     echo "----- Test Step - 6 ----"
132     echo "TEST STEP - 6 "
133     echo "API NAME: Datarestful - Index"
134     echo "URL: http://localhost:8080/"
135
136     response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
137         dev/null http://localhost:8080/)
138
139     if [ $response_code = "200" ]; then
140         echo "STEP - 6: PASS"
141         echo "----- --- ----"
142         return 0
143     else
144         echo "STEP - 6: FAIL"
145         return 1
146     fi
147 }
148 echo "----- --- ----"
149 echo
150
151 echo
152 TestStep_7() {
153     echo "----- Test Step - 7 ----"
154     echo "TEST STEP - 7 "
155     echo "API NAME: Datarestful"
156     echo "URL: http://localhost:8080/Services/"
157
158     response_code=$(curl -XGET -i -k --write-out %{http_code} --output /
159         dev/null http://localhost:8080/Services/)
160
161     if [ $response_code = "200" ]; then
162         echo "STEP - 7: PASS"
163         echo "----- --- ----"
164         return 0
165     else
166         echo "STEP - 7: FAIL"
167         return 1
168     fi
169 }
170 echo "----- --- ----"
171 echo
172
173 TEST_PASS=0
174 TEST_FAIL=0
175 TOTAL_TEST=0
176
177 declare -a arr=("TestStep_0" "TestStep_1" "TestStep_2" "TestStep_3" "
178     TestStep_4" "TestStep_5" "TestStep_6" "TestStep_7" )
179
180 for i in "${arr[@]}"
181 do
182     if $i; then
183         TEST_PASS=$((TEST_PASS+1));
```

```
181     else
182         TEST_FAIL=$((TEST_FAIL+1));
183     fi
184 done
185
186 echo
187 echo
188
189 echo "--- TEST CASE REPORT ---"
190 echo "TEST PASS: " $TEST_PASS
191 echo "TEST FAIL: " $TEST_FAIL
192 echo "TOTAL EXECUTED: " ${#arr[@]}
193 echo "--- --- --- --- --- ---"
```

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] UBUNTU, *Ubuntu 18 TLS* <https://www.ubuntu.com/>
- [3] BASH, *Shell BASH* <https://linux.die.net/man/1/bash>
- [4] VALIDATION PLAN, *Ofni Systems* <http://www.ofnisystems.com/services/validation/validation-plans/>
- [5] POSTMAN, *POSTMAN API DEVELOPMENT* <https://www.getpostman.com/>
- [4] SELENIUM, *Selenium Browser Automation* <https://www.seleniumhq.org/>

Capítulo 6

Anexo

6.1. makefile

Listing 6.1: makefile

```
1 #Makefile
2
3 install:
4     sh ./install.sh
5
6 build:
7     go build generadorP3.go
8
9 generateTestCase1:
10     cd ./testgenerator && ./TestGenerator -config=TestCase1.json
11 runTestCase1:
12     cd ./testgenerator/src/testgenerator/output/ && /bin/bash
13     TestCase1.sh
14 generateTestCase2:
15     cd ./testgenerator && ./TestGenerator -config=TestCase2.json
16 runTestCase2:
17     cd ./testgenerator/src/testgenerator/output/ && /bin/bash
18     TestCase2.sh
19 generateTestCase3:
20     cd ./testgenerator && ./TestGenerator -config=TestCase3.json
21 runTestCase3:
22     cd ./testgenerator/src/testgenerator/output/ && /bin/bash
23     TestCase3.sh
```