

Grupo 5 Football

API: Transfers

Máster Ingeniería del Software: Cloud, Datos y
gestión TI – Fundamentos Ing. Software

Autores: Jose Manuel Luque Mendoza

Johannes Haugsgjerd Haaland

Enrique Guerrero Fernández

Curso 2019/2020



Contenido

Apartado 1: Introducción.	2
Apartado 2: Nivel de acabado.	3
Apartado 3: Requisitos del microservicio.	3
3.1 Microservicio básico.	3
3.1.1 Backend API REST	3
3.1.2 Frontend	5
3.1.3 Despliegue en la nube	6
3.1.4 Accesibilidad API REST	8
3.1.5 Ejemplos de uso de la API con Postman	8
3.1.6 Persistencia con MongoDB	10
3.1.7 Repositorio de Github usando Github Flow.	10
3.1.8 Integración continua y despliegue con Travis.ci	11
3.1.9 Imagen Docker	13
3.1.10 Pruebas unitarias de backend	14
3.1.11 Pruebas integración con BD	16
3.1.12 Mecanismo autenticación de la API	18
3.1.12 Integración con las API del grupo	19
3.2 Microservicio avanzado.	21
3.2.1 Pruebas unitarias mediante mocks.	21
3.2.2 Validación en los formularios de frontend.	22
3.2.3 API REST documentado con swagger.	24
Apartado 4: Análisis de los esfuerzos.	26

Apartado 1: Introducción.

En esta memoria presentamos la aplicación desarrollada para la asignatura de “Fundamentos de Ingeniería del Software”, del máster en “Ingeniería del Software: Cloud, Datos y Gestión TI”.

Esta aplicación se enmarca dentro del grupo 5 de trabajo, de la asignatura anteriormente mencionada. Este grupo tiene por objetivo desarrollar una aplicación basada en microservicios, que sirva para trabajar sobre datos deportivos, concretamente fútbol.

El grupo de trabajo se divide a su vez en 3 subgrupos, los cuales deben de dar soporte a distintas temáticas y funcionalidades dentro la aplicación futbolera. Así pues las temáticas dentro de la aplicación común son:

- **Gestión de torneos y partidos.**
- **Gestión de jugadores y equipos.**
- **Gestión de fichajes de futbolistas.**

Este memoria se centra en la última funcionalidad, la de fichajes de jugadores entre equipos también llamada “*Transfers*” o “*Mercado de fichajes*”. Los alumnos que componen este último grupo son: **Jose Manuel Luque Mendoza, Johannes Haugsgjerd Haaland y Enrique Guerrero Fernández.**

Estos alumnos se han encargado de realizar este microservicio basado en fichajes, para ello se han basado en los conocimientos enseñados en la asignatura, para realizar un backend usando las buenas prácticas vistas para la declaración de API REST, utilizando nodejs como lenguaje de desarrollo, MongoDB como mecanismo de persistencia de datos , JSON Web Tokens como mecanismo de autenticación en la API y React.js para la implementación de un frontend que sirva como UI.

En los apartados posteriores se detallará con mayor grado de detalle cada una de las características de la aplicación, además de mostrar el nivel de acabado al que optan.

Apartado 2: Nivel de acabado.

En este apartado procederemos a describir el nivel general de acabado que presenta nuestra aplicación. En concreto de los 4 niveles posibles (5, 7, 9 y 10 puntos) nuestro equipo se presenta al nivel de acabado de **notable 7**.

Para ello nuestra aplicación debe de cumplir cada uno los requisitos de microservicio básico y al menos 3 de las características de microservicio avanzado. De entre todas las distintas características de microservicio avanzado, hemos optado por:

- Añadir validación a los formularios de frontend.
- API REST documentado con swagger.
- Implementación de pruebas unitarias utilizando moks y/o stubs.

En el siguiente apartado se detalla con mayor profundidad qué acciones se han llevado a cabo, para cumplir tanto con los requisitos de microservicio básico, como de microservicio avanzado.

Apartado 3: Requisitos del microservicio.

3.1 Microservicio básico.

En primer lugar procedemos a justificar cómo se han ido consiguiendo cada uno de los requisitos del microservicio básico

3.1.1 Backend API REST

El backend de nuestra aplicación de transfers se trata de una API REST la cual está desarrollada con nodejs, siguiendo el conjunto de normas y buenas prácticas vistas en clase para la declaración de API REST, como por ejemplo el uso de códigos de estados adecuados.

Todos los servicios REST junto a una breve descripción y códigos de estado de los mismos son mostrados a continuación:

- **GET** /api/v1/transfers
 - Descripción: Obtiene todos los fichajes disponibles.
 - Códigos de estado: 200 OK.
- **GET** /api/v1/transfer/{id}
 - Descripción: Obtiene un fichaje por su identificador.
 - Códigos de estado: 200 OK y 404 Not Found.
- **GET** /api/v1/transfers/player/{player_id}
 - Descripción: Obtiene el histórico de fichajes de un futbolista.
 - Códigos de estado: 200 OK.
- **GET** /api/v1/transfers/team/{team_id}
 - Descripción: Obtiene todos los fichajes realizados por un equipo.

- Códigos de estado: 200 OK.
- **POST** /api/v1/transfer
 - Descripción: Registra un nuevo fichaje en la API.
 - Códigos de estado: 201 Created, 400 Bad Request, 404 Not found (No existe jugador o equipo en la API de teams y players) y 500 Internal Server Error.
- **PUT** /api/v1/transfer/{id}
 - Descripción: Obtiene todos los fichajes disponibles
 - Códigos de estado: 204 No Content, 400 Bad Request, 404 Not found y 500 Internal Server Error.
- **DELETE** /api/v1/transfer/{id}
 - Descripción: Obtiene todos los fichajes disponibles
 - Códigos de estado: 200 OK, 404 Not Found y 500 Internal Server Error.
- **DELETE** /api/v1/transfers
 - Descripción: Obtiene todos los fichajes disponibles
 - Códigos de estado: 200 OK y 500 Internal Server Error.

Por último comentar la descentralización de nuestro backend en distintos directorios y archivos, cada uno con una funcionalidad propia y diferente al resto. Así conseguimos separar y hacer más entendible nuestra aplicación, evitando la presencia de archivos con cientos de líneas de código y funcionalidades mezcladas. Los distintos directorios son:

- **/app/controllers:** Contiene toda la funcionalidad relativa a los métodos o funciones de nuestra API de transfers.
- **/app/integration:** Contiene toda la funcionalidad relacionada con las API externas, en concreto la integración con las API de players y teams del grupo de trabajo 5.
- **/app/models:** Contiene cada uno de los modelos de datos de nuestra aplicación con **mongoose** (permite definir modelos de objetos tipados que se asignan a documentos en mongo), en nuestro caso hay definido el modelo **transfer**.
- **/app/routes:** Contiene el archivo con el directorio de rutas que posee nuestra API, enlaza cada método de nuestra api con una url o end-point mediante el cual quedará accesible para peticiones externas.
- **/config:** directorio de propiedades y configuración de la aplicación, carga en variables cada una de las properties incluidas en el archivo “.env”.
- **/docs:** Documentación del proyecto, incluye tanto esta memoria como las colecciones de postman y swagger con la descripción de la API.
- **/mongo-data:** Incluye los “json” u dummies que se cargaran en BD, para comenzar a trabajar con la aplicación en local con docker.
- **/mongodb:** Incluye la imagen de docker del contenedor de mongo como su configuración.
- **/tests_jest:** Tests de los servicios REST de backend mediante el uso del framework de testing de jest.
- **/test_mocha:** Tests de integración de BD mediante el uso del framework de testing de mocha.
- **.env:** Incluye los valores de cada una de las properties de la aplicación, por ejemplo la API de teams o la url de la base de datos.

- **.travis.yml:** fichero con la secuencia de instrucciones para la integración continua con travis y despliegue en heroku.
- **db.js:** Configuración y conexión de la aplicación con la base de datos de mongo.
- **Dockerfile:** Imagen de docker de la aplicación de transfers.
- **docker-compose.yml:** Fichero docker-compose que se encarga de levantar en local los contenedores de la aplicación, base de datos y carga de datos.
- **index.js:** Punto de arranque y despliegue de nuestra aplicación.
- **server.js:** Contiene cada una de las funciones y librerías de propósito general que usa nuestra aplicación (bodyparser, securización...)

3.1.2 Frontend

También se ha desarrollado un proyecto de frontend común con el resto de los integrantes del grupo 5. Esta aplicación la cual es independiente del backend de transfers lleva por nombre **frontend**. Tiene su propio repositorio en github (<https://github.com/FIS-Equipo-5/frontend>) y cuenta de igual manera con integración continua y despliegue en heroku (<https://fis-gr5-frontend.herokuapp.com/>) mediante el uso de travis.

El frontend se ha realizado mediante el uso de **react** y al igual que el backend se ha tratado de dividir la aplicación en distintas funcionalidades y directorios:

- **/src/auth:** Contiene cada uno de los componentes e interacción con la API, en relación con el proyecto de autenticación basado en JWT (se hablará de él posteriormente)
- **/src/transfers:** Contiene cada uno de los componentes e interacción con la API, en relación con el proyecto de transfers.
- **/src/common:** Contiene cada uno de los componentes comunes de la aplicación, por ejemplo su footer o menú.
- **/src/Alert.js:** Componente general mediante el cual se muestran mensajes de alerta al usuario.
- **/src/App.js:** Componente general de la aplicación, en el se incluye cada uno de los componentes desarrollados por los miembros del equipo.
- **/src/setupProxy.js:** Fichero de proxy donde se incluyen que peticiones de la aplicación serán redirigidas a las distintas API's.
- **.travis.yml:** fichero con la secuencia de instrucciones para la integración continua con travis y despliegue en heroku.

Por último se muestran algunas capturas de la interfaz de usuario de frontend relativas a los componentes de transfers:

Football App

Transfers:

Source Team	Target Team	Player	Transfer Date	Contract Years	Cost	
Real Betis	Real Madrid	Diego	16/01/2020	3	30000000	Add Transfer
Real Betis	Real Betis Atlético de Madrid Real Madrid	Joaquín	24/01/2020	4	25000000	Edit Delete
Real Madrid	Real Betis	Karim	16/01/2020	5	30000000	Edit Delete

Football App

Transfers:

Source Team	Target Team	Player	Transfer Date	Contract Years	Cost	
			dd/mm/aaaa			Add Transfer
Real Betis	Atlético de Madrid	Joaquín	24/01/2020	4	25000000	Edit Delete
Real Madrid	Real Betis	Karim	16/01/2020	5	30000000	Save Cancel
Real Betis	Real Madrid	Diego	16/01/2020	3	30000000	Edit Delete

Football App

Transfers:

Source Team	Target Team	Player	Transfer Date	Contract Years	Cost	
			dd/mm/aaaa			Add Transfer
Real Betis	Atlético de Madrid	Joaquín	24/01/2020	4	25000000	Edit Delete
Real Madrid	Real Betis	Karim	16/01/2020	5	30000000	Edit Delete
Real Betis	Real Madrid	Diego	16/01/2020	3	30000000	Edit Delete

3.1.3 Despliegue en la nube

Otro de los puntos a cumplir por parte de nuestra aplicación era el hecho de estar desplegada en un servidor cloud , de manera que cualquier interesado en trabajar con ella dispusiese de una url pública de acceso, en lugar de tener que realizar una instalación en su ordenador personal para trabajar con ella en local, la idea es que esta aplicación también pueda ser usada por usuarios con conocimientos limitados de informática, haciendo que la posibilidad de trabajar en local sea a todas luces inviable, debido a la dificultad de la instalación.

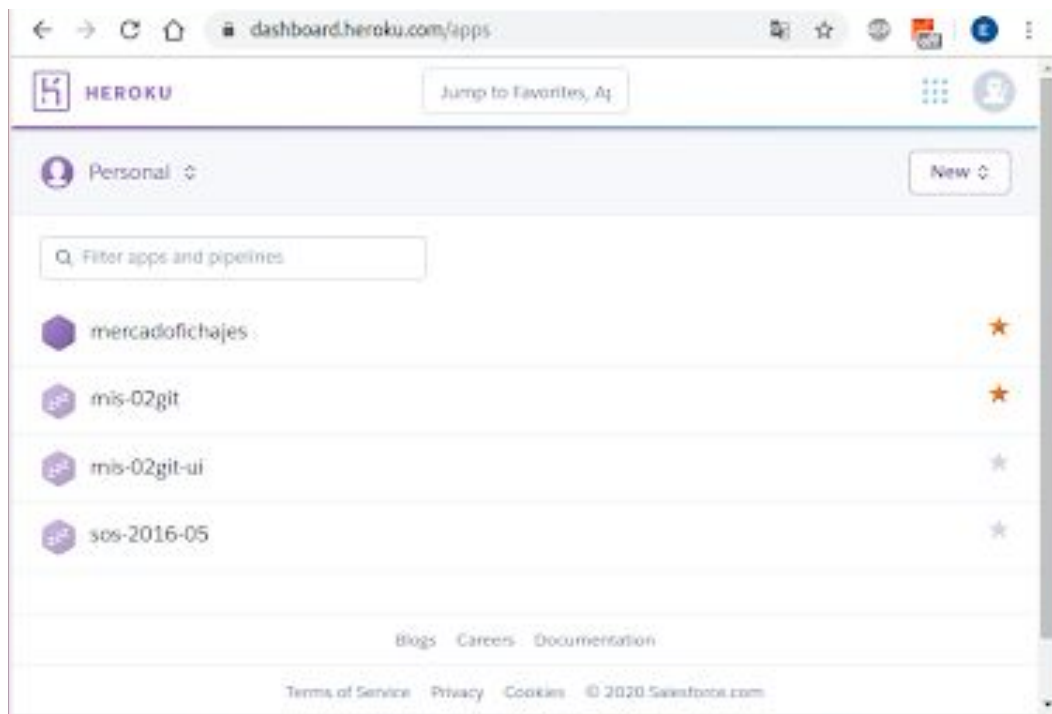
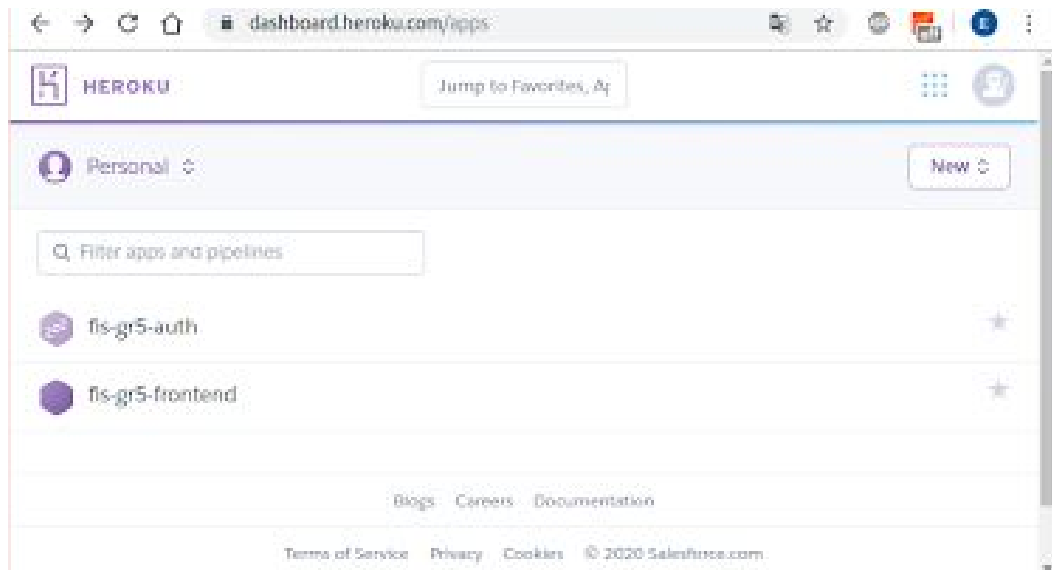
Por ello se ha hecho uso de la plataforma de computación en la nube de **Heroku**. Esta plataforma tras previo registro, nos ha permitido desplegar cómodamente cada una de las aplicaciones del grupo. Ya sea directamente por línea de comando con *"git push heroku master"*, o en combinación con travis y su archivo *".travis.yml"* del que se hablará posteriormente.

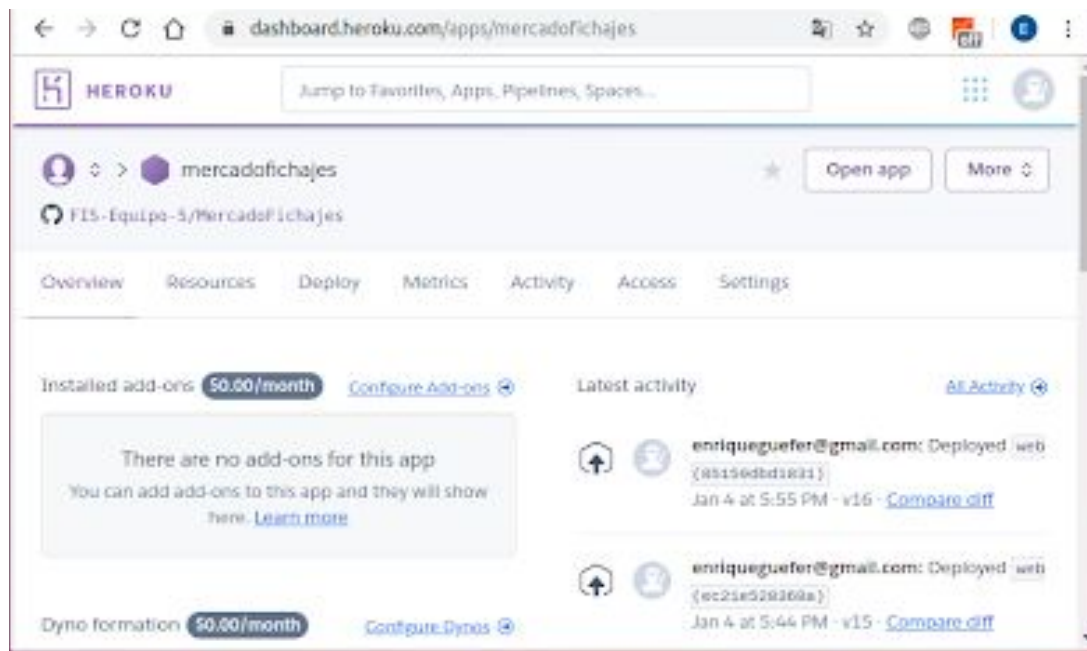
Todas las direcciones web públicas de las aplicaciones comunes del grupo 5 y del microservicio de transfers, se muestran a continuación:

- **Microservicio Transfers:** <https://mercadofichajes.herokuapp.com/>

- **Frontend:** <https://fis-gr5-frontend.herokuapp.com/>
- **Proyecto autenticación JWT:** <https://fis-g5-auth.herokuapp.com/>

Para finalizar se muestran algunas capturas de heroku con las aplicaciones anteriormente citadas:





3.1.4 Accesibilidad API REST

Otro de los requisitos que debía cumplir nuestra aplicación es que la API que gestionara el recurso de transfers, debía estar accesible en una dirección bien versionada.

Este requisito se cumple debido en primer lugar a la sintaxis clara y nítida que se ha hecho de cada recurso de la api, y en segundo lugar gracias al servidor cloud de heroku, que nos permite publicar en una dominio versionado nuestra aplicación. Así cada uno de los recursos de la API queda accesible en la siguiente dirección web completa:

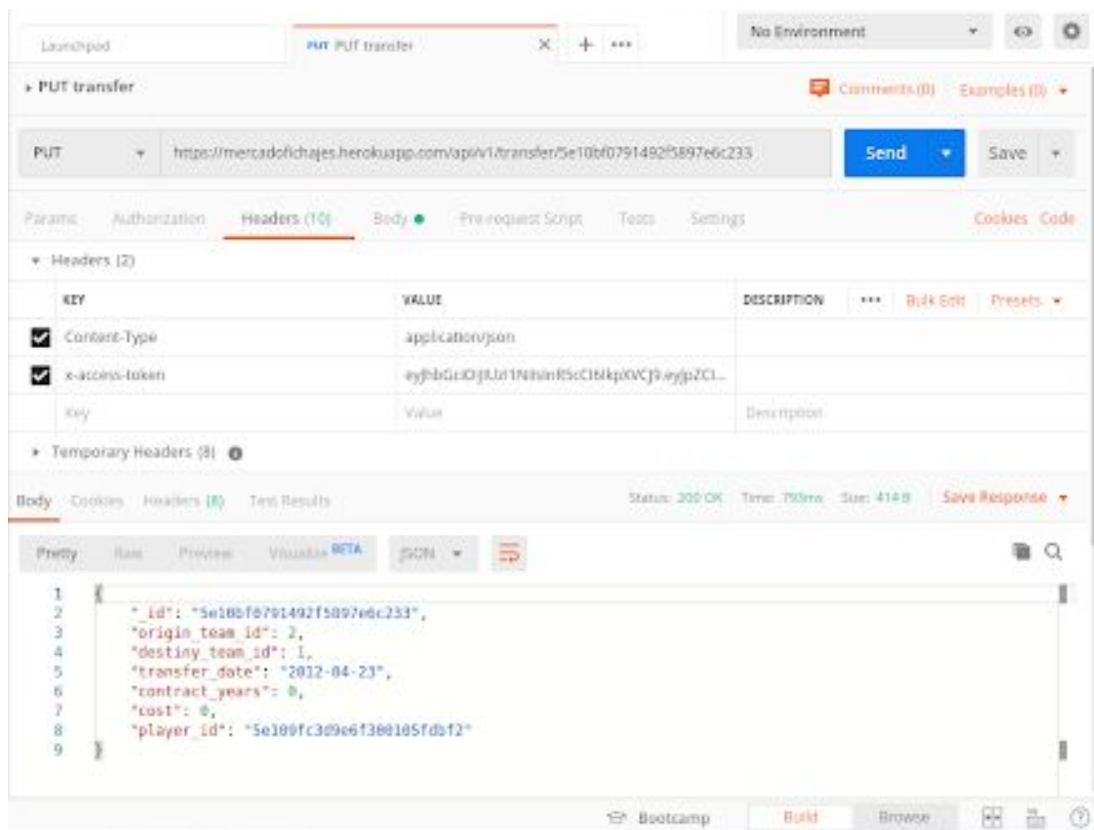
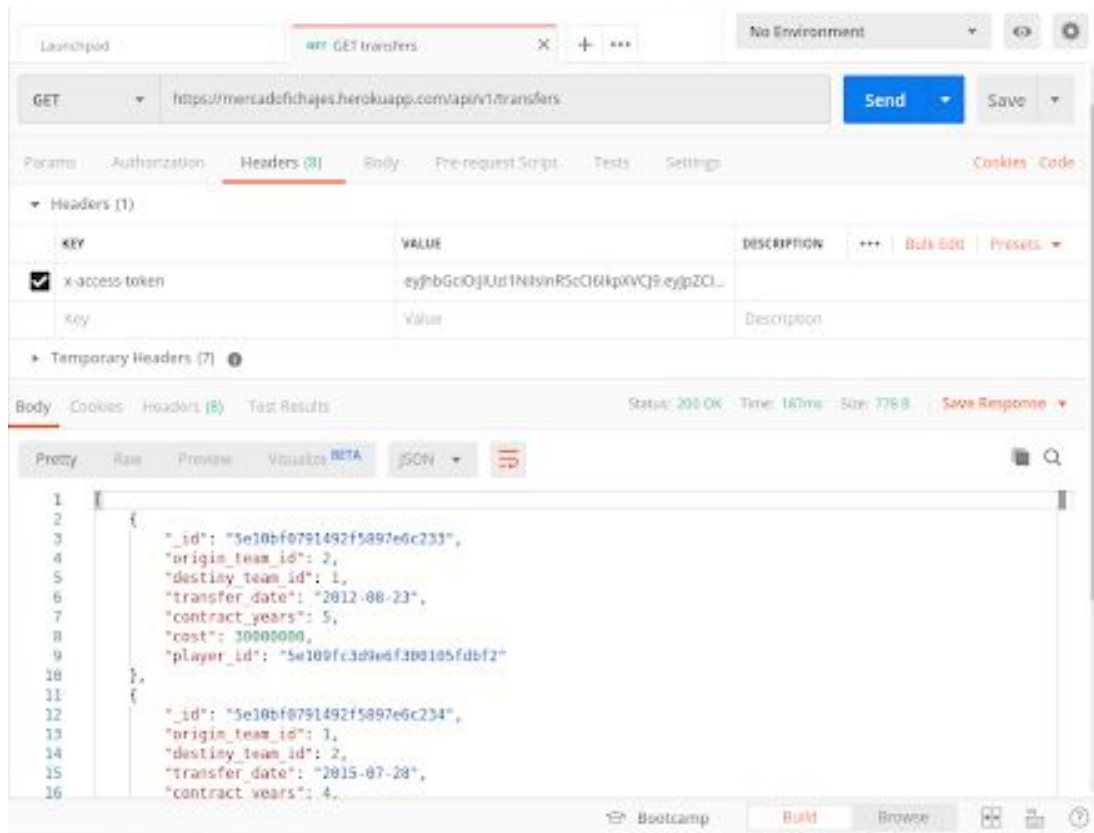
- **GET** <https://mercadofichajes.herokuapp.com/api/v1/transfers>
- **GET** <https://mercadofichajes.herokuapp.com/api/v1/transfer/{id}>
- **GET** https://mercadofichajes.herokuapp.com/api/v1/transfers/player/{player_id}
- **GET** https://mercadofichajes.herokuapp.com/api/v1/transfers/team/{team_id}
- **POST** <https://mercadofichajes.herokuapp.com/api/v1/transfer>
- **PUT** <https://mercadofichajes.herokuapp.com/api/v1/transfer/{id}>
- **DELETE** <https://mercadofichajes.herokuapp.com/api/v1/transfer/{id}>
- **DELETE** <https://mercadofichajes.herokuapp.com/api/v1/transfer/{id}>

3.1.5 Ejemplos de uso de la API con Postman

Continuamos con otro de los requisitos del microservicio básico, este trataba sobre la inclusión de una colección de ejemplos de uso de la API en Postman. En el directorio del proyecto `"/docs/MIS - HEROKU Transfers API.postman_collection.json"` se encuentra esta colección.

La colección posee una prueba de uso de cada servicio REST de la API de transfers, además de 2 peticiones con las que registrarse en el proyecto de autenticación y obtener el token JWT que debe de incluirse en el "Header" de cada petición a la API. Si no se incluye el token

se obtendrá una respuesta del tipo “JWT expired” o “JWT must be provided”. Se muestra para finalizar algunas capturas del uso de la API con Postman:



3.1.6 Persistencia con MongoDB

Con respecto a la persistencia de datos, hemos utilizado la librería “Mongoose” que nos proporciona un sistema de “schemas” fuertemente tipados para crear nuestros modelos e instancias de base de datos en MongoDB. Mediante esta librería, creamos un Schema para nuestra entidad de base de datos, definiendo los diferentes campos y sus características, dicha entidad será directamente exportada a nuestra instancia de MongoDB una vez que estemos conectados. Para la configuración de la conexión, hemos utilizado principalmente dos ficheros:

- **db.js:** En este fichero cargamos la librería de mongoose y realizamos la conexión con mongoDB a la cual se llamará en el fichero index.js para levantar el servidor con la aplicación.
- **database.config.js:** En este fichero, asignamos la configuración de la base de datos así como del servidor haciendo uso de la librería “dotenv”, la cual nos permite obtener parámetros de configuración a través de un fichero de variables de entorno “.env”, en el cual tenemos la url de la base de datos así como el puerto en el que se levantará el servidor. De este modo, externalizamos la configuración, de cara a seguir las buenas prácticas de desarrollo de la asignatura.

3.1.7 Repositorio de Github usando Github Flow.

Uno de los requisitos principales del proyecto era el de disponer de un repositorio de código en Github. La principal característica Github se encuentra en la posibilidad de alojar proyectos utilizando el sistema de control de versiones de Git, el cual permite disponer y controlar distintas versiones del código fuente, posibilitando que varios desarrolladores puedan trabajar en paralelo minimizando los conflictos entre ellos.

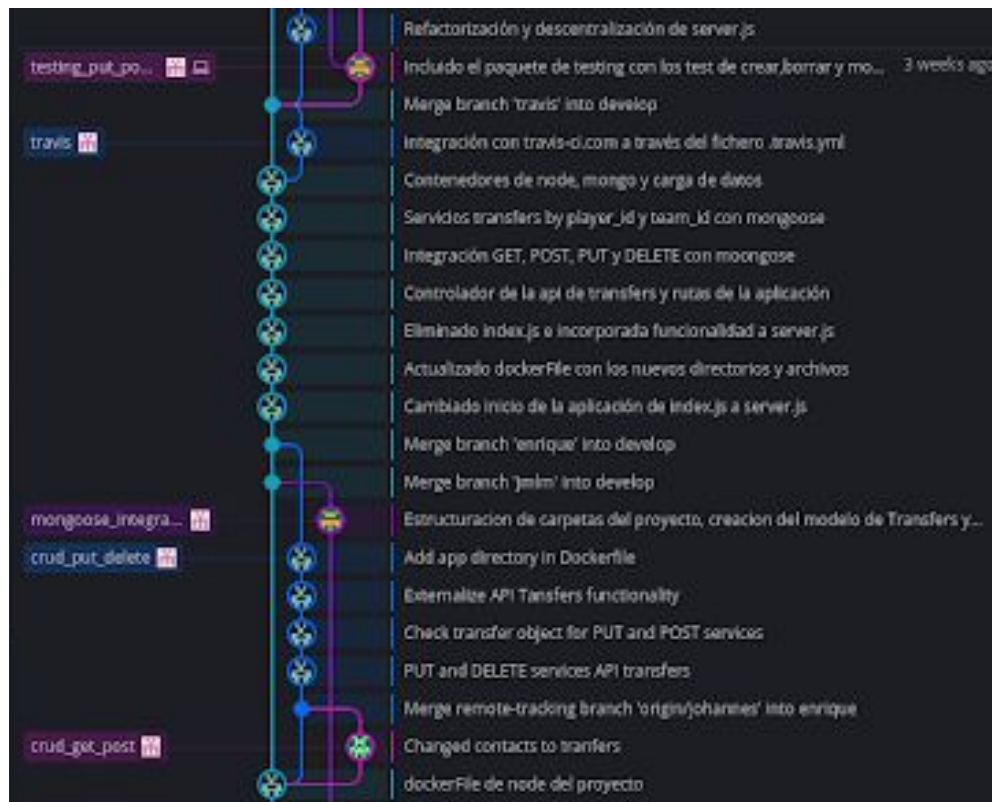
Todos los proyectos del grupo 5 de la API de fútbol, disponen de un repositorio en Github. Se muestran a continuación los comunes al grupo de desarrollo (frontend y autenticación) y el propio de transferencias:

- **Transfers:** <https://github.com/FIS-Equipo-5/MercadoFichajes>
- **Frontend:** <https://github.com/FIS-Equipo-5/frontend>
- **Proyecto autenticación JWT:** <https://github.com/FIS-Equipo-5/auth>

Junto al uso de Github como repositorio tenemos otro requisito que especificaba el uso de Github Flow para desarrollo. Github Flow se trata de un estilo de workflow que permite añadir nuevos desarrollos al proyecto de forma segura, sin comprometer la estabilidad del mismo. La solución se encuentra principalmente en el **branching** del proyecto.

El branching consiste en crear distintas ramas de desarrollo por cada nueva tarea a realizar, nombrando cada rama con un nombre que la identifique fácilmente, en definitiva asignar ramas a tareas. Durante el desarrollo se van realizando commits a la rama (salvar un conjunto de cambios), al finalizar el desarrollo, esta rama se incorpora (merge) a otra rama, donde se van añadiendo las distintas funcionalidades del proyecto.

Este estilo de programación se ha ido siguiendo en cada uno de los repositorios del proyecto. Basta con acceder a cualquier de ellos y ver las multitudes de ramas que existen. Por ejemplo en transfers: “mongoose_integration”, “travis”, “jest_testing”, “jwt_validation”... Se muestra finalmente una captura de la herramienta gráfica de “git_kraken”, que sirve para ver visualmente como funciona este estilo de programación en branches:



3.1.8 Integración continua y despliegue con Travis.ci

Otro de los aspectos relevantes del proyecto es la integración continua, la cual consiste en hacer integraciones automáticas del proyecto lo más a menudo posible, para así detectar fallos cuanto antes. Entendemos por integración la compilación y ejecución de pruebas de todo un proyecto.

Para todos y cada uno de los proyectos del grupo, se ha usado **Travis** como herramienta de integración. Esta herramienta permite a través de un archivo “.travis.yml” ubicado en la raíz del proyecto, compilar y ejecutar los tests de una aplicación. Cada vez que un desarrollador actualiza una rama remota, travis inmediatamente y tras previa configuración en <http://travis-ci.com/> y autorización en github para acceder a los repositorios, echa mano de este archivo para informar sobre el resultado de compilación y tests del desarrollo realizado. Además es posible configurar el archivo de “.travis.yml” para que en caso de ejecución satisfactoria, se despliegue la nueva versión de la aplicación en la nube.

Todos y cada uno de los **proyectos del grupo tienen definido un archivo “.travis.yml”**. Los cuales compilan la aplicación y ejecutan los tests, y despliegan las actualizaciones

satisfactorias de la rama máster en heroku. Se muestra una captura del archivo “.travis.yml” del repositorio de transfers y del resultado de la integración en la página de travis:

```

MercadoFichajes > ! .travis.yml
1  sudo: required
2  language: node_js
3  node_js:
4    - "9"
5  services:
6    - docker
7  before_install:
8    - wget -qO- https://toolbelt.heroku.com/install-ubuntu.sh | sh
9  install:
10   - npm install
11   - npm test
12  script:
13    docker build -t transfers .
14  deploy:
15    providers: script
16    skip_cleanup: true
17    script:
18      heroku container:login;
19      heroku container:push web -a $HEROKU_APP_NAME;
20      heroku container:release web -a $HEROKU_APP_NAME;
21  branch: master

```

FIS-Equipo-5 / MercadoFichajes build passing

Current Branches Build History Pull Requests > Build #53 Job #53.1 More options

✓ master Merge branch 'develop' 453.1 passed Restart job

→ Commit 1d33efc 🔗 Ran for 2 min

🔗 Compare 9a18ad1...1d33efc 🔗 about an hour ago

🔗 Branch master 🔗

👤 Enrique Guerrero Fernández

🔗 Node.js: 9

🔗 AMD64

[Job log](#) [View config](#)

```

1 Worker information
2
3 Build system information
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

3.1.9 Imagen Docker

Continuamos con el requisito de incluir al menos una imagen Docker en el proyecto. Docker se caracteriza por el despliegue de aplicaciones dentro de contenedores software, permitiendo así automatizar la ejecución y despliegue de múltiples aplicaciones independientemente del sistema operativo anfitrión.

En nuestro proyecto hemos realizado al menos una imagen docker por aplicación (a excepción del proyecto de frontend que no posee ninguna). El repositorio común de auth dispone de un archivo Dockerfile en la raíz del proyecto, y **nuestro proyecto de transfers dispone** de hasta un total de **3 imágenes Docker**, combinadas con un archivo **docker-compose**, que posibilita el despliegue simultáneo de los 3 contenedores de transfers que son:

- **Server:** Nuestra aplicación backend con la API rest de transfers
- **DB:** Nuestra base de datos local de MongoDB para pruebas.
- **mongo-data:** La carga inicial de dummies en la base de datos.

Para finalizar se muestran las capturas relativas a los archivos de docker-compose y Dockerfile de la API REST y de la base de datos:



```
1 version: '3'
2
3 services:
4   server:
5     container_name: server
6     build:
7       context: .
8       dockerfile: Dockerfile
9     depends_on:
10      - db
11     ports:
12      - "3000:3000"
13     restart: always
14
15   db:
16     container_name: db
17     build:
18       context: ./mongodb
19       dockerfile: Dockerfile
20     ports:
21      - "27017:27017"
22     restart: always
23
24   mongo-data:
25     build: ./mongo-data
26     depends_on:
27      - db
```

```
Untitled-2 •  Untitled-3 •  docker-compose.yml  Dockerfile MercadoFichajes X
MercadoFichajes > Dockerfile
1  FROM node:9-alpine
2
3  #Establecer directorio de trabajo
4  WORKDIR /app
5
6  #Instala los paquetes existentes en el package.json
7  COPY package.json .
8  RUN npm install --quiet
9
10 COPY server.js .
11 COPY db.js .
12 COPY index.js .
13 COPY .env .
14 COPY app ./app
15 COPY config ./config
16 COPY docs/swagger ./docs/swagger
17
18 EXPOSE 3000
19 CMD ["npm", "start"]
```

```
Untitled-2 •  Untitled-3 •  docker-compose.yml  Dockerfile
MercadoFichajes > mongodb > Dockerfile
1  FROM mongo:3.4-jessie
2
3  # Copy config file
4  COPY mongod.conf /etc/mongod.conf
5
6  # Expose MongoDB port
7  EXPOSE 27017
8
9  # Run application
10 CMD ["mongod", "--config", "/etc/mongod.conf"]
```

3.1.10 Pruebas unitarias de backend

Con respecto a las pruebas de integración de backend, hemos generado un directorio “test_jest” en la raíz del proyecto, que contiene un único archivo (“server.test.js”) con todas las pruebas de la API REST de transfers. Conviene citar que se han realizado tests no solo de cada servicio REST, si no también de cada una de las distintas respuestas (códigos de estado) con las que se informa al usuario del resultado de su petición sobre el servicio REST.

Para realizar dichas pruebas, hemos utilizado el framework de “Jest”, el cual está desarrollado por el equipo de facebook, y aunque nace en el contexto de React, es un framework de testing generalista en javascript que destaca por sus funcionalidades potentes e innovadoras.

En la siguiente imagen podemos ver cómo se estructura un test de pruebas unitarias de la API REST en nuestra aplicación:


```

9
10 describe("Transfer API", () => {
11
12     describe("GET /transfers", () => {
13
14         let transfers_temp = [
15             new Transfer({origin_team_id: 1, "destiny_team_id": 2, "transfer_date": "2012-08-23"},
16             new Transfer({origin_team_id: 2, "destiny_team_id": 3, "transfer_date": "2015-07-28"},
17         ];
18
19         beforeEach(() => {
20             dbFind = jest.spyOn(Transfer, "find");
21             token = jest.spyOn(jwt, "verify");
22         });
23
24         it("Should return all transfers", async () => {
25
26             dbFind.mockImplementation((query, callback) => {
27                 callback(null, transfers_temp);
28             });
29
30             token.mockImplementation((token, secretOrPublicKey, callback) => {
31                 callback(false, "id");
32             });
33
34             return request(app).get(BASE_API_PATH + "/transfers").then((response) => {
35                 expect(response.statusCode).toBe(200);
36                 expect(response.body.length).toBe(transfers_temp.length);
37                 expect(dbFind).toBeCalledWith({}, expect.any(Function));
38             });
39         });
40     });
41
42 });

```

Los pasos a seguir son los siguientes:

1. Primero importamos la dependencia de *"request"* de supertest para poder realizar peticiones sobre la aplicación, en concreto request debe de recibir un archivo de entrada que en nuestro caso será el archivo *"server.js"*, el cual se trata del archivo de entrada a nuestra API. Además es necesario importar la entidad del modelo, como el resto de archivos que son llamados durante la ejecución del test.
2. Mediante la llamada a *"describe()"*, podemos agrupar las acciones que vamos a realizar sobre la API en la colección de tests de este fichero, en este caso, vamos a hacer una llamada al servicio REST que devuelve todas las transferencias de la API.
3. Toma especial relevancia la llamada *"beforeAll()"* que permite especificar qué operaciones se realizarán antes de ejecutar los test. En este caso se monitoriza mediante *"spyOn"* los métodos *"find"* y *"verify"* que serán llamados posteriormente durante la ejecución del test.
4. Mediante las llamadas a *"it()"*, definimos y ejecutamos el test en sí, primero debemos describir **qué valores devolverán (mocks)** los métodos descritos anteriormente en el *"beforeAll()"* y a continuación escribimos la petición REST que se realizará a la API, en este caso una petición GET al recurso *"transfers"*. Finalmente mediante expects, comprobamos que la respuesta es la esperada con una condición, si dicha condición se

cumple, el test nos indicará que todo está correcto, de lo contrario, el test fallará y por tanto tendremos errores en nuestro código.

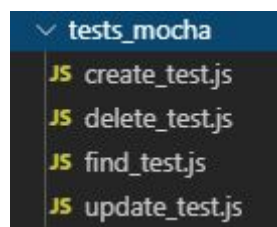
5. Para ejecutar los tests es necesario situarnos en la carpeta “tests_jest” y ejecutar en la consola el siguiente comando: “npm test”. Tras ejecutar dicho comando, jest ejecutará todos los tests y nos indicará el estado de los mismos por consola:

```
PROBLEMS  OUTPUT  DEBUGCONSOLE  TERMINAL
✓ Should return an Error Object (18ms)
GET /transfers/player/:player_id
✓ Should return all transfers of a player (19ms)
✓ Should return an Error Object (16ms)
GET /transfers/team/:destiny_team_id
✓ Should return all transfers of a player (14ms)
✓ Should return an Error Object (19ms)
POST /transfer
✓ Should add a new transfer if everything is fine (93ms)
✓ Should return a 400 Bad Request for the input Transfer (18ms)
✓ Should return 500 if there is a problem with the DB (14ms)
PUT /transfer/:transfer_id
✓ Should update a transfer if everything is fine (24ms)
✓ Should return a null Transfer (27ms)
✓ Should return an Error Object of kind ObjectId (21ms)
✓ Should return an Error Object (29ms)
DELETE /transfer/:transfer_id
✓ Should delete a transfer by Id (18ms)
✓ Should return a null Transfer (12ms)
✓ Should return an Error Object of kind ObjectId (17ms)
✓ Should return an Error Object (17ms)
DELETE /transfers
✓ Should delete all transfers (13ms)
✓ Should return an Error Object (18ms)

Test Suites: 1 passed, 1 total
Tests: 23 passed, 23 total
Snapshots: 0 total
Time: 7.900s
Ran all test suites.
console.log app/controllers/transfer/controller.js:254
```

3.1.11 Pruebas integración con BD

Con respecto a las pruebas de integración con BD, hemos generado un fichero para cada tipo de operación realizable sobre la misma.



Para realizar dichas pruebas, hemos utilizado el framework “Mocha”, el cual está diseñado para NodeJS y nos proporciona numerosas funcionalidades con respecto al diseño y reporte de tests.

En la siguiente imagen podemos ver cómo se estructura un test de integración en nuestra aplicación:

```

1  const assert = require('assert');
2  const Transfer = require('../app/models/transfer.model'); //imports the Transfer model.
3
4  let transferDummy;
5
6  beforeEach( async () => {
7      transferDummy = await new Transfer({ origin_team_id: '1',destiny_team_id: '2',transfer_date: ne
8      transferDummy = await new Transfer({ origin_team_id: '1',destiny_team_id: '2',transfer_date: ne
9
10     transferDummy.save()
11     .then(() => done());
12 });
13 describe('Finding documents',() => {
14     it('Finds all transfers', async () => {
15         Transfer.find()
16         .then((transfer) => {
17             assert(transfer !== null);
18             done();
19         });
20     });
21
22     it('Finds transfer on ID', async () => {
23         Transfer.findOne(transferDummy)
24         .then(() => Transfer.findById(transfer._objectId)
25         .then((transferById) => {
26             assert(transferById === transferDummy);
27             done();
28         }));
29     });
30

```

Los pasos a seguir son los siguientes:

1. Primero importamos la dependencia de “assert” para las assertions, así como nuestra entidad del modelo. Mediante el método “beforeEach” definimos los requisitos de entrada del test, en este caso, estamos testeando un findAll de todas nuestras Transfers por tanto necesitamos crear al menos un par de instancias de la misma para poder comprobar que se obtienen correctamente al ejecutar nuestro “find()”.
2. Mediante la llamada a describe() podemos describir las acciones que vamos a realizar en la colección de tests de este fichero, en este caso, vamos a hacer operaciones de búsqueda(find) sobre las instancias de nuestra entidad.
3. Mediante las llamadas a it(), definimos y ejecutamos el test en sí, primero debemos describir qué operación vamos a testear y a continuación escribimos las operaciones que se realizan en la funcionalidad que estamos testeando, finalmente mediante asserts, comprobamos que la respuesta es la esperada con una condición, si dicha condición se cumple, el test nos indicará que todo está correcto, de lo contrario, el test fallará y por tanto tendremos errores en nuestro código.
4. Para ejecutar los test es necesario situarnos en la carpeta “tests_mocha” y ejecutar en la consola el siguiente comando: “npm test”. Tras ejecutar dicho comando, mocha ejecutará todos los tests y nos indicará el estado de los mismos por consola:

```
Finding documents
✓ Finds all transfers
✓ Finds transfer on ID
✓ Finds all transfers on teamID
✓ Finds transfers on playerID
```

Este framework nos permite también visualizar los reportes de tests mediante una interfaz gráfica, aunque para nuestro microservicio no hemos visto necesario su utilización.

3.1.12 Mecanismo autenticación de la API

Para la autenticación, hemos utilizado la estrategia de autenticación mediante JWT(JSON Web Token). Para ello, hemos desarrollado un microservicio que se encarga exclusivamente de la gestión, autenticación y autorización de usuarios. En dicho microservicio, mediante la librería “jsonwebtoken” generamos el token que utilizarán el resto de microservicios de cara a que sus operaciones sean autorizadas. Dicha librería nos proporciona herramientas para poder implementar la autenticación por token de forma sencilla y segura sin tener que realizar una implementación totalmente “ad-hoc”.

El flujo de autenticación/autorización es el siguiente:

1. El usuario, entra en la app e introduce sus credenciales de usuario (email y password), dichas credenciales son enviadas mediante una petición de tipo “POST” al microservicio de auth, el microservicio de auth comprobará que dicho usuario existe en la base de datos (ha debido registrarse previamente, dicho registro también es realizado por el microservicio de auth) y si es así, devolverá la siguiente información:
 - a. status : Devuelve el estado del intento de autenticación, en este caso “success”.
 - b. message: Devuelve un mensaje a modo de respuesta con respecto al intento de autenticación. En este caso, “user found!!!”
 - c. data: Devuelve dos elementos
 - i. La información del usuario(user):
 1. _id: El id de usuario.
 2. name: El nombre con el que se registró el usuario.
 3. email: El email con el que se registró el usuario.
 4. password: la contraseña cifrada del usuario.
 5. __v: Parámetro que indica la versión interna del usuario creado por mongoose.
 - ii. El token de autorización(token): Un token cifrado en función del secreto usado que contiene la información del usuario entre otras cosas.
2. El frontend almacena el token en el localStorage y permite al usuario navegar por la aplicación.
3. El token es inyectado como valor de la cabecera “x-access-token” en todas las peticiones que se realizan desde la aplicación(viajan desde el frontend hacia el backend).

4. El backend mediante el método `jwt.verify()` valida el token recibido antes de realizar cada operación, en dicha validación se descifra el token en base al secreto (el cual debe ser y es el mismo en todos los microservicios) y se obtiene si el token es válido y no está caducado, en caso afirmativo, se realiza la operación que se ha solicitado y en caso negativo, se devolverá un error y no se realizará la operación.

Para cada intento de autenticación, se genera un token válido en caso de que todos los pasos descritos anteriormente se desarrollen correctamente. Dichos tokens expirarán en base al tiempo configurado en la aplicación.

3.1.13 Integración con las API del grupo

Llegamos a unos de los requisitos más importantes del trabajo y es la tarea de integración con el resto de las API's del grupo. Como se comentó anteriormente nuestra api trata sobre fútbol y entre los distintos recursos que se gestionan tenemos: torneos y partidos, jugadores y equipos y fichajes.

Nuestro subgrupo posee la API de fichajes, la cual puede definirse como transacciones de futbolistas entre equipos. Cada fichaje quedaría registrado por:

1. **Clubes implicados:** El equipo que vende y el equipo que compra a un futbolista. De los cuales se almacenan los id's. Llamados a partir de ahora por *"origin_team_id"* y *"destiny_team_id"*.
2. **El futbolista:** jugador que protagoniza la transferencia. Del cual se almacenará también su id. Llamado a partir de ahora por *"player_id"*.
3. **Fecha del traspaso:** *"transfer_date"*
4. **Años de contrato:** *"contract_years"*
5. **Coste de la transferencia:** *"cost"*

Como puede observarse por la definición de nuestro modelo de datos, nos vemos obligados a realizar una integración con las API's de *"teams"* y *"players"*. Esta integración sea ha realizado en el backend de nuestra API REST de Transfers, concretamente en el archivo *"app/controllers/transfer.controller.js"*. Los archivos *"player.integration.js"* y *"teams.integration.js"* del directorio *"app/integration"* recogen cada una de las operaciones (GET's o PUT's) que se realizan sobre estas 2 API's como resultado de la integración con la API de *"transfers"*.

Un resumen de la integración que se ha realizado sería:

1. **Registro de un fichaje (POST):** Al registrar un fichaje en la base de datos, se recuperan de la API de teams y player los datos relativos a dichos equipos y jugador. Por medio de las peticiones REST **GET**:

- https://fis2019-teams.herokuapp.com/api/v1/player/{player_id}
- https://fis2019-teams.herokuapp.com/api/v1/teams/team/{origin_team_id}
- https://fis2019-teams.herokuapp.com/api/v1/teams/team/{destiny_team_id}

Tras cambiar al futbolista el valor del atributo *“team_id”* por el identificador del nuevo equipo que le acaba de fichar, se realiza una petición **PUT** para actualizar al jugador a:

- <https://fis2019-teams.herokuapp.com/api/v1/player>

En el cuerpo de la petición viaja el objeto al completo del futbolista actualizado. Por último toca actualizar los campos presupuesto y valor de los equipos implicados. Para ello al equipo origen o el equipo que ha realizado la venta, se le suma en su atributo presupuesto (*“budget”*) el coste de la transferencia y se le resta en el atributo valor (*“value”*) el valor que tenía el futbolista que acaba de abandonar el club.

Respecto al equipo destino o el equipo que ficha al futbolista, se le resta en su atributo presupuesto el coste de la transferencia y se le suma en el atributo valor el valor del futbolista que acaba de adquirir. Tras estas 2 modificaciones se realizan dos peticiones **PUT** para actualizar a ambos equipos al servicio REST:

- https://fis2019-teams.herokuapp.com/api/v1/teams/{team_name}

- 2. Actualización de un fichaje (PUT):** La actualización de un fichaje solo permite la actualización del tiempo del contrato (*“contract_years”*) y del coste de la transferencia (*“cost”*). No tiene sentido en el ámbito de una transferencia de deportistas, poder modificar los equipos que han vendido/fichado ni el futbolista que ha sido transferido.

Por ello tras actualizar satisfactoriamente la transferencia en BD, en el caso de que se haya cambiado el coste de la transferencia se recuperan mediante 2 llamadas **GET** los datos de los equipos implicados con :

- https://fis2019-teams.herokuapp.com/api/v1/teams/team/{origin_team_id}
- https://fis2019-teams.herokuapp.com/api/v1/teams/team/{destiny_team_id}

En el atributo presupuesto (*“budget”*) de ambos equipos, se le aplica la diferencia de precio entre el antiguo valor de la transferencia y el nuevo. Tras ello se actualizan los equipos con 2 **PUT** a:

- https://fis2019-teams.herokuapp.com/api/v1/teams/{team_name}

3.2 Microservicio avanzado.

Para finalizar procedemos a justificar cómo se han superado cada uno de los 3 requisitos de microservicio avanzado. Recordemos que la entrega permitía elegir 3 opciones o más de entre un total de 8 requisitos, para alcanzar el nivel de notable 7. Los 3 requisitos escogidos por nuestra parte han sido “pruebas unitarias mediante mocks”, “validación en formularios” y “API REST documentado con Swagger”.

3.2.1 Pruebas unitarias mediante mocks.

Este primer requisito consiste en desarrollar pruebas unitarias que implemente mocks. Un mock en el ámbito de testing puede definirse como un “doble” de una función, que permite verificar si un método concreto ha sido llamado, qué parámetros ha recibido o cuántas veces ha sido invocado. Además también permite devolver un objeto de respuesta determinado.

En nuestra API de transfers hemos implementado mocks en las pruebas de integración de BD. Basta con recorrer estos tests para echar un vistazo y comprobar su uso. Específicamente se han usado mocks para:

1. Mockear la respuesta de la función *“verify”* de la librería *“jsonwebtoken”* que comprueba la validez del token.
2. Mockear las respuestas de la BD cuando se realizan operaciones sobre ella.
3. Mockear las respuestas de las APIS de *“teams”* y *“player”* cuando se realiza alguna petición para la integración.

Se muestra una captura del uso de estos mocks en el test de **POST** transfer:

```
192
193   beforeEach(() => {
194     dbInsert = jest.spyOn(Transfer, "create");
195   });
196
197   it('Should add a new transfer if everything is fine', async () => {
198     dbInsert.mockImplementation((c, callback) => {
199       callback(false, new Transfer(transfer_post));
200     });
201
202     getPlayer = jest.spyOn(playersApi, "getPlayerById");
203     updatePlayer = jest.spyOn(playersApi, "updatePlayer");
204
205     getTeam = jest.spyOn(teamsApi, "getTeamById");
206     updateTeam = jest.spyOn(teamsApi, "updateTeam");
207
208     getPlayer.mockImplementation((id) => {
209       return player;
210     });
211
212     updatePlayer.mockImplementation((obj) => {
213       return true;
214     });
215
216     getTeam.mockImplementation((id) => {
217       return team;
218     });
219
220     updateTeam.mockImplementation((obj) => {
221       return true;
222     });
223
224   });
```

3.2.2 Validación en los formularios de frontend.

Este requisito trata sobre validar los campos del formulario del frontend de transfer, de manera que informe al usuario de algún dato que ha obviado introducir, o un dato que sea incorrecto o bien limite de alguna manera el tipo de entrada a un formulario, o la longitud de los datos que puede escribir en el mismo.

En el frontend de transfer se ha optado por:

1. Informar al usuario que está obligado a insertar todos los datos para registrar una transferencia:

Football App

Transfers:

Error! You must write all the transfer fields ×

Source Team	Target Team	Player	Transfer Date	Contract Years	Cost	
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text" value="dd/mm/yyyy"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Add Transfer"/>
Real Madrid	Real Betis		23/08/2012	5	30000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Real Betis	Real Madrid		28/07/2015	4	60000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Real Betis	Real Madrid		12/08/2016	5	50000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

2. Informar al usuario que está obligado a escribir el coste y años de contrato cuando edita una transferencia:

Football App

Transfers:

Error! You must write the cost and the contract years ×

Source Team	Target Team	Player	Transfer Date	Contract Years	Cost	
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text" value="dd/mm/yyyy"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Add Transfer"/>
Real Madrid	Real Betis		23/08/2012	<input type="text"/>	30000000	<input type="button" value="Save"/> <input type="button" value="Cancel"/>
Real Betis	Real Madrid		28/07/2015	4	60000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Real Betis	Real Madrid		12/08/2016	5	50000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

3. Tanto en el registro como en la edición, los atributos “cost” y “contract_years” están limitados a valores numéricos de 9 y 1 caracteres de longitud:

Football App

Transfers:

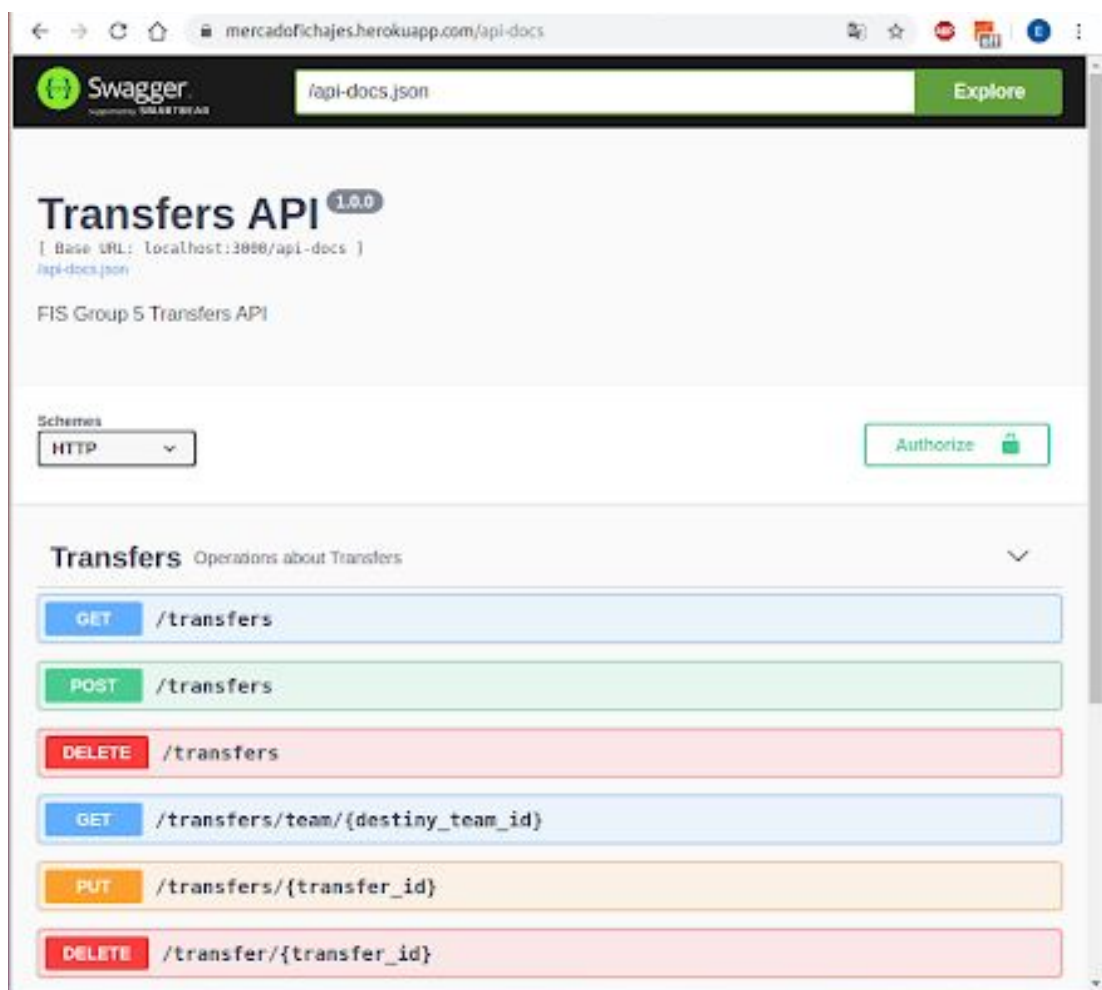
Source Team	Target Team	Player	Transfer Date	Contract Years	Cost	
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text" value="dd/mm/yyyy"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Add Transfer"/>
Real Madrid	Real Betis		23/08/2012	6	345354354	<input type="button" value="Save"/> <input type="button" value="Cancel"/>
Real Betis	Real Madrid		28/07/2015	4	60000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Real Betis	Real Madrid		12/08/2016	5	50000000	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

3.2.3 API REST documentado con Swagger.

Llegamos al último requisito, este trata sobre documentar la API REST de Transfers mediante el uso de Swagger. Swagger es una plataforma software que permite a los desarrolladores diseñar, construir, documentar y consumir servicios web REST.

Las APIs permiten conectar y compartir datos, pero surge un gran problema y es que las APIs no cuentan con un estándar de diseño común y tampoco con unos parámetros comunes de documentación. Swagger permite solventar estos problemas gracias a la definición de una serie de reglas, especificaciones y herramientas que nos ayudan a documentar nuestras APIs.

Para la definición de nuestra API de Transfers con swagger, hemos hecho uso de la dependencia de npm de [“express-swagger-generator”](#). Esta dependencia por medio de unas líneas de código en el archivo `“index.js”` y mediante la declaración de los servicios de la API REST y modelo de datos en un archivo `“.js”` (`/docs/swagger/transfersSwagger.js`), nos permite de manera cómoda documentar nuestra API y publicarla en la dirección `“/api-docs”` para su visualización. Así si accedemos a <https://mercadofichajes.herokuapp.com/api-docs>. Podemos observar la documentación swagger de nuestra API:



mercadofichajes.herokuapp.com/api-docs#/Transfers/get_transfers_player_...

GET /transfers/player/{player_id}

This function content is parsed by doctrine

Parameters Try it out

Name	Description
player_id * required string (path)	player id - eg: 23

Responses Response content type: application/json

Code	Description
200	<p>Requested Transfer</p> <p>Example Value / Model</p> <pre>{ "transfer_id": "5dee74accf9a95284ba5b323", "transfer_date": "2019-08-31", "contract_years": "3", "origin_team_id": "213", "destiny_team_id": "212", "cost": 8, "player_id": 0 }</pre>
404	Not Found

Models

```
Transfer {
  transfer_id* string
    example: 5dee74accf9a95284ba5b323
    transfer id
  transfer_date* string
    example: 2019-08-31
    transfer date
  contract_years* number
    example: 3
    contract years
  origin_team_id* number
    example: 213
    player's original team id
  destiny_team_id* number
    example: 212
    player's destiny team id
  cost* number
    player cost eg: 234230
  player_id* number
    player id eg: 20
}
```

VALID {-}

Apartado 4: Análisis de los esfuerzos.

En el siguiente apartado se muestra una tabla con todas las tareas realizadas para llevar a cabo el proyecto, cada tarea se encuentra desgranada por nombre, autor, fecha en la que se comenzó a realizar y horas de esfuerzo que ha supuesto para cada componente del grupo:

Nombre tarea	Fecha de comienzo	Autor	Horas de esfuerzo
Inicialización del proyecto de backend y repositorio	4/12/2019	Todos	30 min
CRUD GET y POST	5/12/2019	Johannes	5 h
CRUD PUT y DELETE	6/12/2019	Enrique	6 h
Integración con Mongo	5/12/2019	Jose Manuel	5 h
Imágenes de docker del proyecto y base de datos	5/12/2019	Enrique	1 h
Integración con travis	13/12/2019	Enrique	1 h
Testing integración BD PUT, POST y DELETE (Mocha)	14/12/2019	Jose Manuel	15 h
Testing integración BD GETs (Mocha)	17/12/2019	Johannes	5 h
Testing backend (Jest)	16/12/2019	Enrique	8 h
Inicialización y desarrollo del Proyecto auth	27/12/2019	Jose Manuel	20 h
Autenticación JWT en el proyecto de transfers	27/12/2019	Jose Manuel	8 h
Colección de postman	23/12/2019	Enrique	1 h
Integración con las API de teams y players	23/12/2019	Enrique	28 h
Documentación API con Swagger	23/12/2019	Johannes	11 h
Inicialización y desarrollo del frontend de transfers	2/01/2020	Enrique	24 h
Desarrollo del frontend de Auth	4/01/2020	Jose Manuel	12 h
Desarrollo de la memoria	5/01/2020	Todos	

