CodigoDelSur

Challenge Node.js

MovieAPI

Agustín Ferrari Luciano

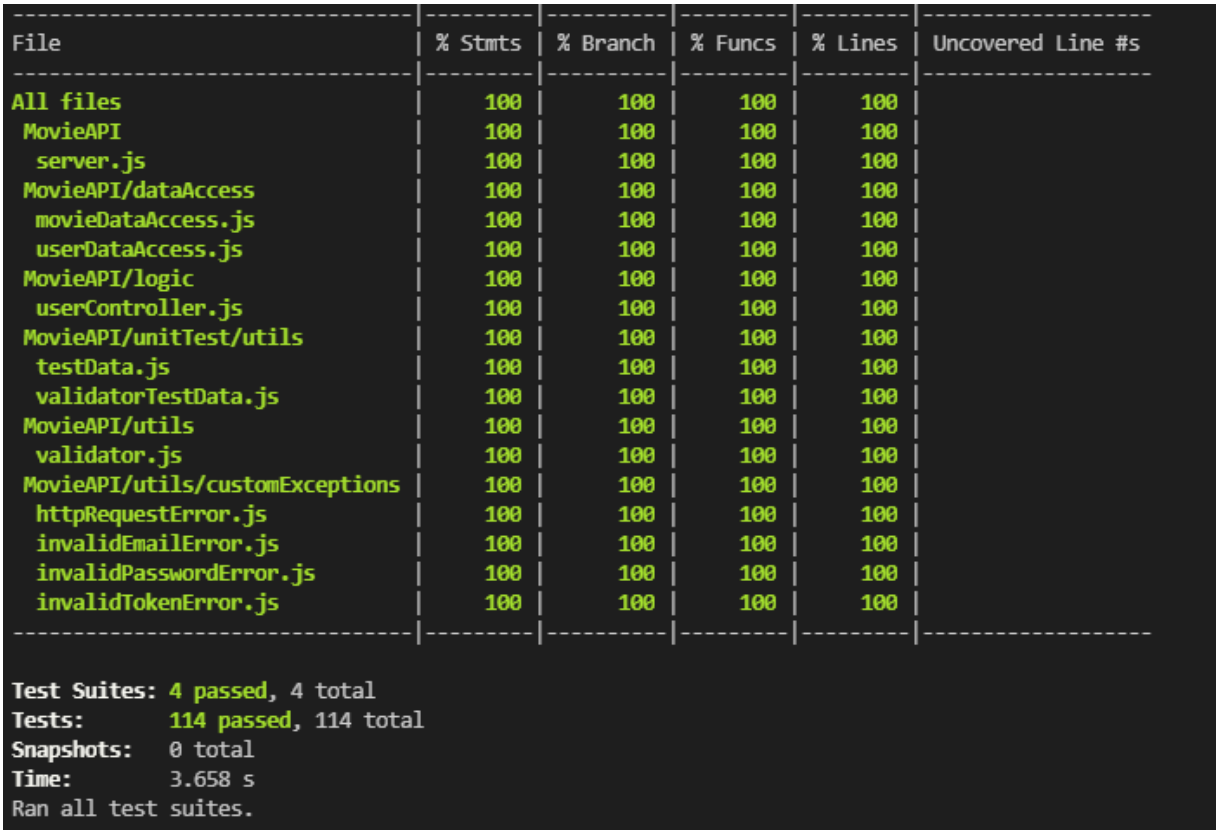
16/7/2021

Índice:

Características generales del proyecto:

Técnicas de codificación

El challenge fue realizado utilizando la técnica TDD (Test-driven development). De esta manera podía asegurarme de que a medida que agregaba funcionalidades, estas iban a estar testeadas y prevenir bugs en caso de que una nueva funcionalidad interfiera con una ya implementada. Esto también me proporcionó una buena cantidad de test unitarios, 114 en total, con un coverage del 100%.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
MovieAPI	100	100	100	100	
server.js	100	100	100	100	
MovieAPI/dataAccess	100	100	100	100	
movieDataAccess.js	100	100	100	100	
userDataAccess.js	100	100	100	100	
MovieAPI/logic	100	100	100	100	
userController.js	100	100	100	100	
MovieAPI/unitTest/utils	100	100	100	100	
testData.js	100	100	100	100	
validatorTestData.js	100	100	100	100	
MovieAPI/utils	100	100	100	100	
validator.js	100	100	100	100	
MovieAPI/utils/customExceptions	100	100	100	100	
httpRequestError.js	100	100	100	100	
invalidEmailError.js	100	100	100	100	
invalidPasswordError.js	100	100	100	100	
invalidTokenError.js	100	100	100	100	

```
Test Suites: 4 passed, 4 total
Tests:       114 passed, 114 total
Snapshots:   0 total
Time:        3.658 s
Ran all test suites.
```

Testing

Para los tests unitarios se utilizó el framework de testing Jest, que si bien lo conocía por haberlo utilizado en Fundamentos de Ing. de Software, implicó una cantidad considerable de investigación ya que tuve que implementar mocks y spies, herramientas en las que no había ahondado en este framework.
Como complemento para poder testear los endpoints, también se utilizó SuperTest. El cual desconocía antes de comenzar con el challenge pero fue importante para realizar este tipo de testings unitarios.
También se realizó testing funcional utilizando PostMan, herramienta con la cual tampoco había tenido contacto anteriormente. Utilizar esta herramienta permitió además definir ejemplos de uso para la documentación de la API.

Estándares de codificación

En cuanto a los estándares de codificación, tomando en cuenta que la empresa trabaja para clientes mayoritariamente de Estados Unidos, se decidió codificar enteramente en inglés e intentar seguir con los principios de Clean Code.
También se utilizó el linter de Google con ESLint para JavaScript con algunos cambios, por ejemplo que el máximo largo de líneas sea de 100 caracteres. Esto ayudó a tener una mayor consistencia y que el código sea más prolijo.

Repositorio

Teniendo en cuenta que el proyecto era individual, se decidió utilizar GitHub pero unicamente a modo de almacenamiento y control de versionado del mismo, es decir, no utilicé ningún workflow, ya que si bien he utilizado Git Flow en otros proyectos (ejemplo), consideré que no sería necesario en un proyecto corto individual.

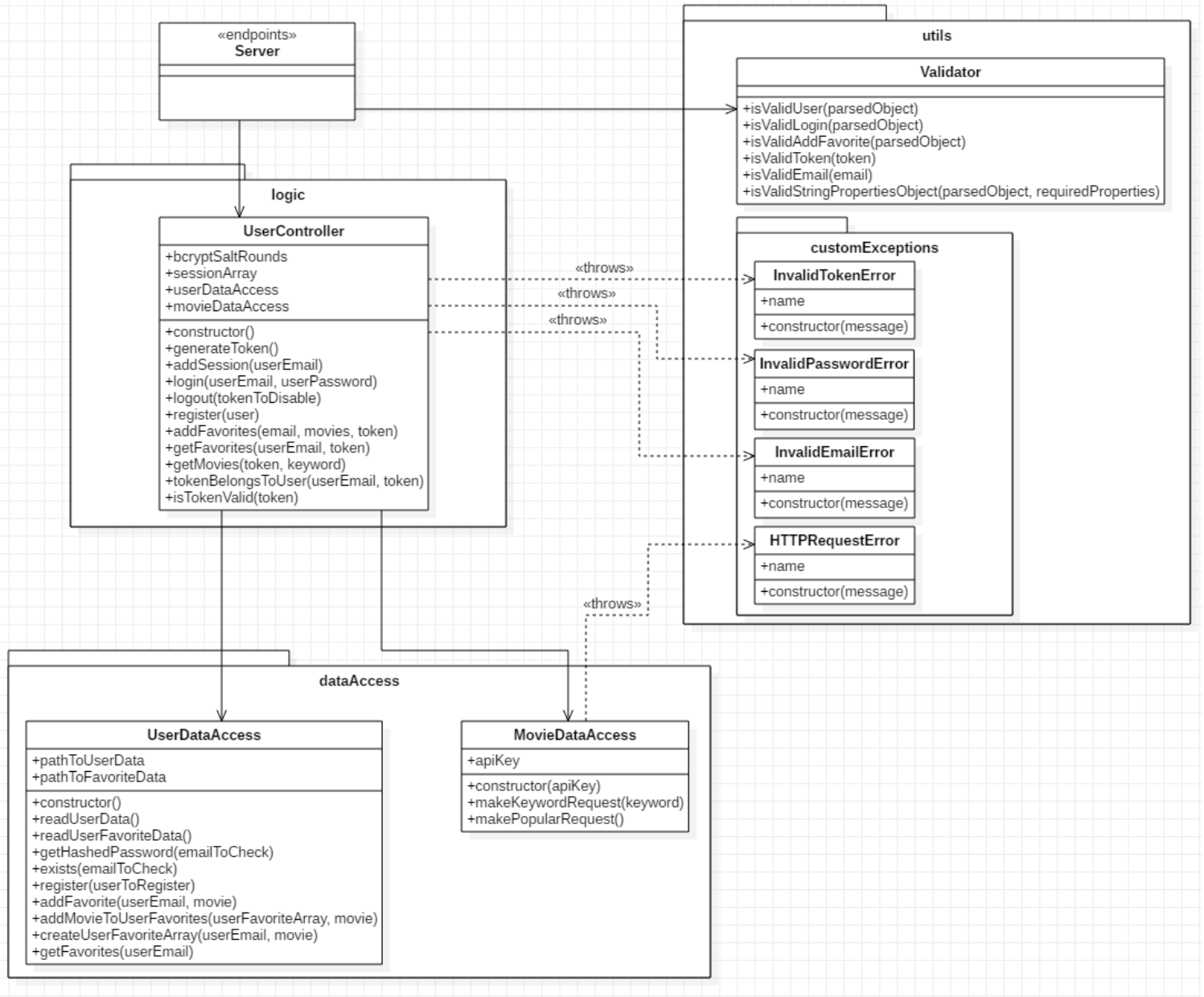
Nomenclatura de archivos, clases, variables y funciones

Tanto para nombrar los archivos como las variables y funciones se decidió usar camelCase, mientras que para nombrar las clases se decidió usar PascalCase.

Diseño y análisis de requerimientos:

Para el diseño de la solución se decidió usar TDD como fue mencionado al principio, con una solución orientada a objetos mayoritariamente. Se separó en tres principales capas, la primera siendo la de los *endpoints*, que se encuentran en server.js, la segunda la de la *logic* y por último el dataAccess, que se encarga de acceder a los archivos de texto y a la API de TheMovieDB. También se utilizó un paquete auxiliar (*utils*), que contiene excepciones custom y validadores de objetos JSON, que fueron surgiendo como necesidad durante el desarrollo.

A continuación se adjunta un diagrama de clases para ayudar a visualizar la estructura de la solución.



(El server fue agregado en el diagrama, si bien no es una clase, sirve para entender mejor la estructura, por eso se usó el estereotipo de endpoints dando a entender que no se hace referencia a una clase sino a un archivo conteniendo los mismos).

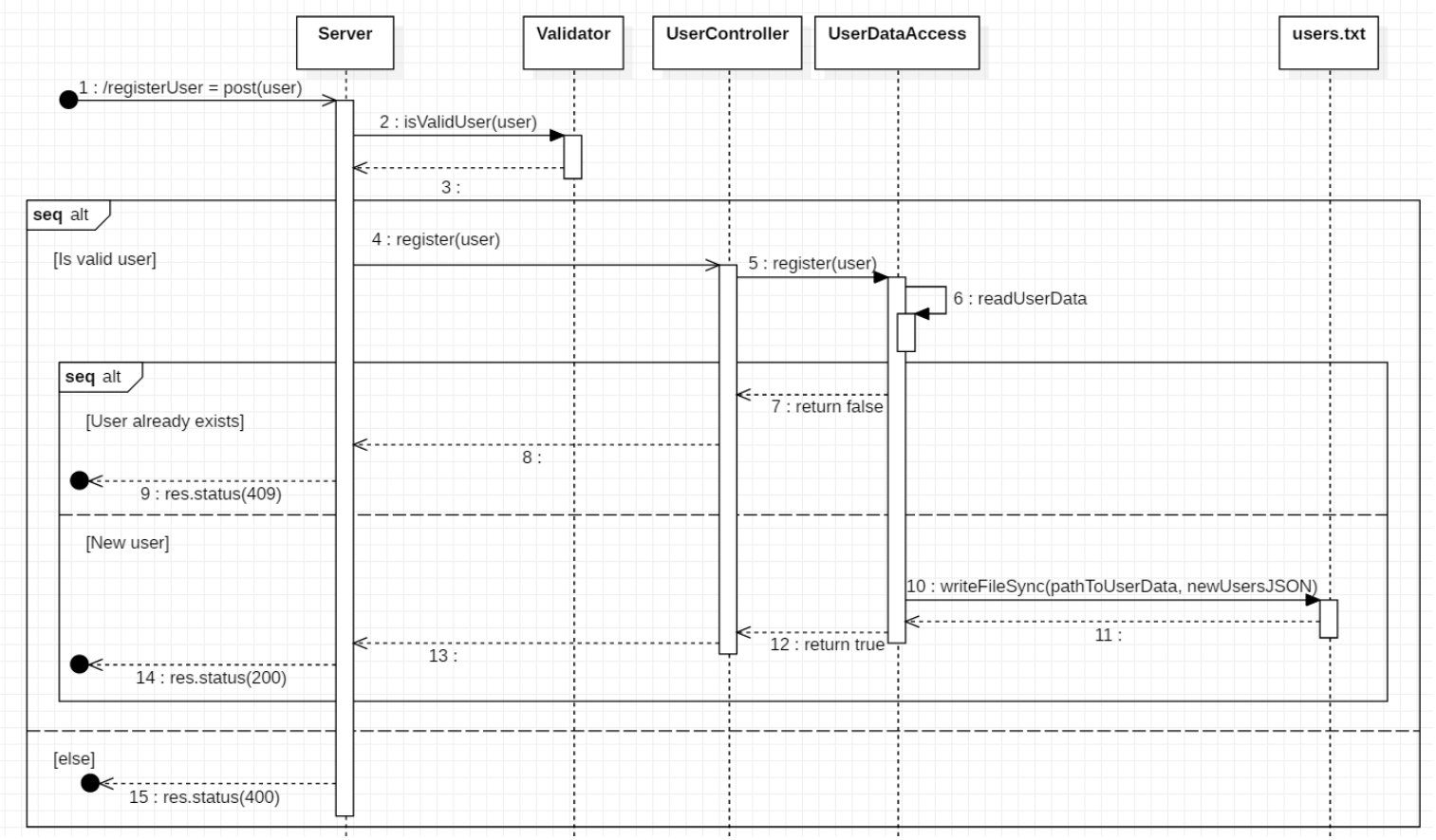
Endpoints:

Se utilizó el framework Express para permitir el acceso a los endpoints, enviar respuestas y manejar los datos recibidos, utilizando el puerto 3000.
A partir de ahora los códigos a los que hago referencia para las respuesta son los códigos de estado de HTTP.
Para todos los endpoints se realiza un análisis previo de los querys o JSON recibidos, dependiendo de cada uno, y en caso de no cumplir con los formatos esperados, se envía un mensaje de error correspondiente (puede referir a un JSON invalido como a un token con formato invalido por ejemplo) con el código 400. En todos los endpoint descritos a continuación no se toma en cuenta este caso para los flujos alternativos (Ni tampoco los errores inesperados, a los cuales se decidió responder con el código 500.

Registrar usuario:

Para el registro de usuarios se creó el endpoint /registerUser, al cual se le envía un post con un JSON conteniendo el email, firstName, lastName y la password del usuario.
Para guardar y acceder a los datos de los archivos txt se decidió hacer un archivo users.txt donde se guarda esta información en formato JSON. Se utilizó el módulo fs incluido en el core de Node.js.
También se implementó un hashing de la password, usando bcrypt y almacenando la contraseña hasheada en users.txt. Este tipo de hashing se basa en ser lento, por lo cual se debe esperar su llamada asincrónica.
En caso de ya existir un usuario registrado con el mismo cambio identificador (email), se devuelve el código de error 409 con un mensaje de error como respuesta de la request.

A continuación se adjunta un diagrama de secuencia mostrando como es un registro desde que llega la request hasta que se responde:



Autenticar usuario:

Para el requerimiento de autenticar usuario, debemos utilizar los mismos módulos que para registrarlo. Es decir fs para conseguir los datos de los archivos de texto y bcrypt para chequear la contraseña ingresada contra la hasheada.
También se implementa una noción de sesión, ya que se debe devolver un token alfanumérico de 20 caracteres que el usuario deberá usar para poder acceder a los otros endpoints. Como se decidió implementar el manejo de múltiples usuarios, cada token le pertenece a un único usuario y ese usuario solo puede acceder a su lista de favoritos por ejemplo. Para el manejo de tokens, se utilizó un array conteniendo pares de {email,token}. Este se guardará en memoria mientras el servidor esté activo, una vez se apague, se invalidará todos los tokens y todos los usuarios deberán volver a autenticarse.
Si un usuario ya autenticado pide un token, se deshabilitará el token antiguo y se le dará uno nuevo.

Para autenticar un usuario se debe enviar un post request al endpoint /login con el email y la password del usuario en formato JSON en el body.
En caso de no existir el email del usuario que se intenta autenticar, se devuelve un mensaje de error acorde con el estado 409. Si el email está registrado pero la password es incorrecta, se envía otro mensaje de error pero con el mismo estado (según lo indicado en la consulta al cliente, aunque esto probablemente comprometa la seguridad porque podrían hacer un ataque enviando request para conseguir los emails).

A lo largo del proyecto se intentó devolver un único tipo de datos en las funciones, es por eso que en este caso se decidió crear excepciones custom para estos flujos alternativos, ya que en un curso normal, la función de login debe devolver un token como string.

Capacidad de hacer logout:

Otro requerimiento opcional era dar la posibilidad de hacer logout, para eso se creó el endpoint /logout, para utilizarlo hay que mandar un post request con el token como un query.
Una vez recibida una request se chequea si existe el token y luego lo elimina del array donde están guardados.

Obtener películas:

Para este requerimiento fue necesario poder realizar requests a la API de TheMovieDB, para esto se tuvo que solicitar una API Key e investigar la documentación. Una vez entendida la documentación se comenzó a probar los endpoints que nos provee la API con Postman, para luego quedarme con 2 endpoints:

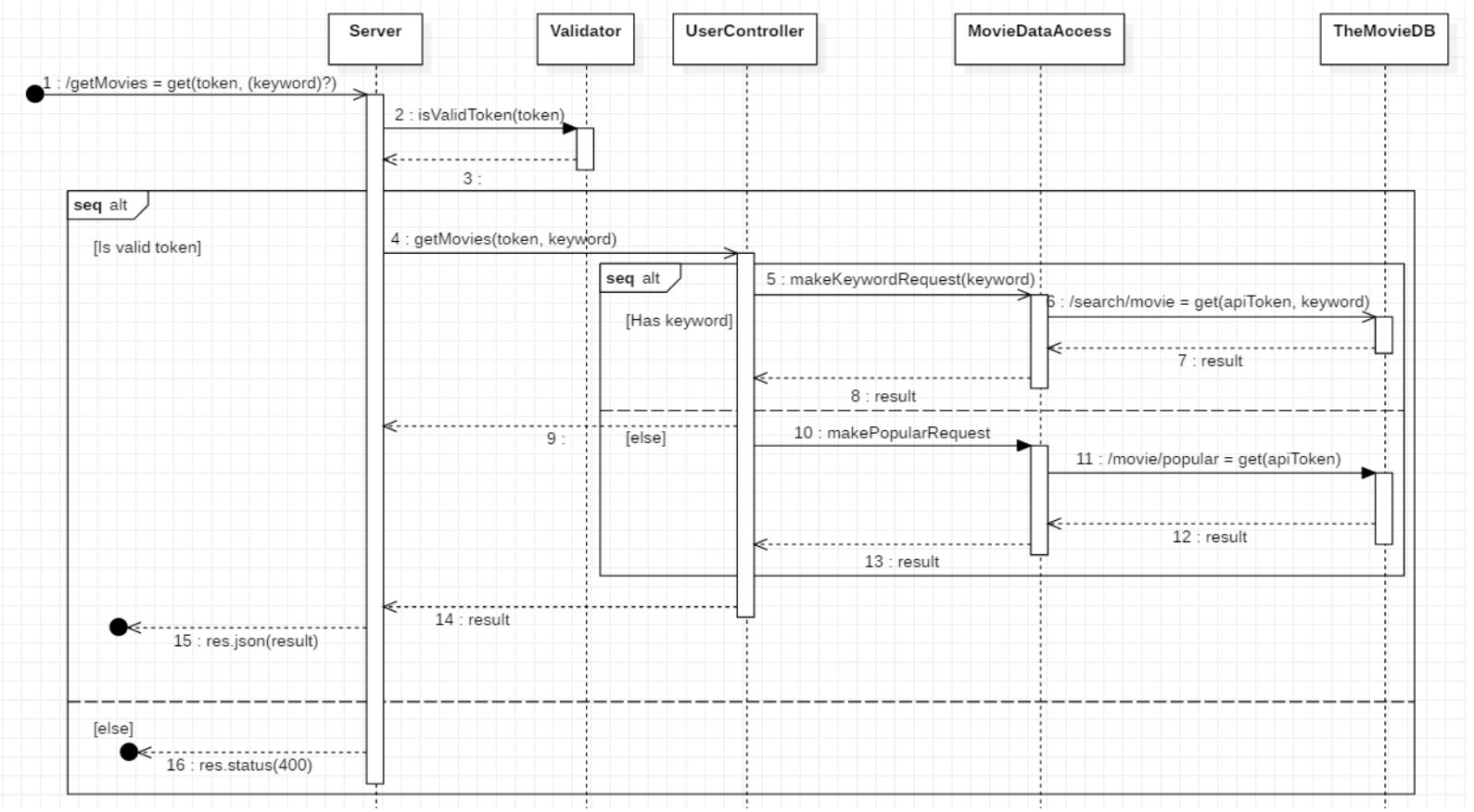
- /search/movie, para buscar películas por keyword
- /movie/popular, para obtener las películas más populares, en caso de no recibir un keyword

Las request a TheMovieDB se realizan en la clase MovieDataAccess.
Para ambos casos se pide una página de películas, que son 20 películas en total, y son enviadas en formato JSON, como no se especifica ningún filtro sobre las propiedades que le interesan al cliente se envía toda la información de las películas recibidas, más un campo suggestionScore (autogenerado entre 0 y 99) por el cual se ordena el array de películas.

En un principio para hacer las consultas HTTP se pensó en utilizar request, pero al darse cuenta que estaba obsoleto, se decidió usar unirest ya que es muy simple de usar y teniendo que realizar únicamente dos request de get a la API de TheMovieDB, no generaba ningún problema.

El endpoint encargado de recibir estas request es /getMovies, que recibe como query en el request el token y una keyword (opcional) y devuelve un array de 20 películas en formato JSON.

A continuación se muestra un diagrama de secuencia para visualizar el curso normal de pedir películas (no se muestran los cursos alternativos como un error de comunicación con TheMovieAPI o un token no autenticado):



Para la obtención de películas también se tienen en cuenta los flujos alternativos. Si por ejemplo se envía un token no autenticado se envía como respuesta el código 401, si hay un problema al hacer la request a TheMovieAPI se envía el 502, ambos acompañados de un mensaje acorde.

Agregar películas a favoritos:

Para agregar películas a favoritos, de manera similar a el requerimiento de registrar usuario se decidió usar fs y almacenar los datos en el archivo favorites.txt. También se almacena en formato JSON, con objetos conteniendo el email del usuario y un array con sus favoritos, al cual se agrega para cada película, la fecha en que fue agregada.
Al momento de agregar películas se controla que no hayan sido ya agregadas y en ese caso se evita insertar duplicados (para este chequeo se usa el id de las películas que recibe el usuario al hacer una request a /getMovies).

El endpoint encargado de recibir esta request es /addFavorites y recibe por medio de un post el objeto que quiere añadir, es decir el JSON con el email y un array de películas, junto con el token recibido al autenticar ese email, como query.

Los posibles flujos alternativos son que no se encuentre registrado el email del usuario al que queramos agregar favoritos o que el token ingresado no este linkeado a el email ingresado, para ambos se responde con un código 401 y mensajes acorde para cada caso.

Obtener películas favoritas:

Este requerimiento se puede ver como una mezcla entre obtener películas y autenticar usuario, en el sentido que vamos a devolver películas pero consultando a favorites.txt con fs en lugar de hacer una request HTTP con unirest. También se debe validar que el token pertenezca a el email al que se hace la consulta, ya que como se implementó el manejo de múltiples usuarios, un usuario solo puede acceder a su lista de favoritos.

Para hacer una consulta a este endpoint se debe realizar un get a /getFavorites especificando el token y email como query. Se devolverá la lista de películas que el usuario tenga en favorites.txt, agregando el campo suggestionForTodayScore (número aleatorio de 0 a 99) y ordenando por el mismo.

El único flujo alternativo que existe aparte de los mencionados al principio de esta sección, es el de un token que no sea el autenticador del email del usuario que se quieren obtener sus favoritos, a esto se responde con un mensaje de error y estado 401.

Manejo de archivos

Para el manejo de archivos, como ya fue mencionado anteriormente se decidió utilizar fs y guardar todos los datos en formato JSON, lo que nos permite tener la posibilidad de migrar a una base de datos NoSQL como lo es MongoDB sin mayores problemas.
Cuando se leen los archivos se parsean los objetos, en caso de que exista un error en esta operación, se tomó la decisión de borrar todo el archivo y comenzar a escribirlo de nuevo, pues es imposible interpretar su contenido si no se puede parsear.