ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

SISTEMA DE INFORMACIÓN PARA PELÍCULAS CON ANGULARJS

INFORMATION SYSTEM FOR MOVIES WITH ANGULARJS

Realizado por
Jose Antonio Díaz González
Tutorizado por
Antonio Jesús Nebro Urbaneja
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, septiembre 2016

Fecha defensa:
El Secretario del Tribunal

E.T.S. DE INGENIERÍA INFORMÁTICA, UNIVERSIDAD DE MÁLAGA

Resumen

En este Trabajo Fin de Grado se llevará a cabo el desarrollo de una aplicación Web que muestre información sobre películas y series de televisión, así como de las personas que intervienen en las mismas.

El sistema se conforma de una serie de controladores y servicios que, atendiendo al patrón de arquitectura MVC (*Modelo Vista Controlador*), se encargan de proporcionar la información necesaria para las vistas. De tal forma que estos controladores son el nexo de unión entre las vistas de la aplicación y la API REST de Trakt.tv. Las vistas permiten a los usuarios acceder a las distintas secciones de información necesarias para cubrir los requisitos propuestos.

Trakt.tv tiene a disposición de los desarrolladores una API con información recopilada sobre películas y series. De modo que los usuarios puedan consultar cualquier tipo de datos referente a la temática de la aplicación a desarrollar, con la garantía de que estos datos irán en aumento conforme va surgiendo nuevo contenido en la actualidad.

Palabras clave

AngularJS, Consultar, Películas, Series, Actores, Información, aplicación cinéfila, API, REST, Trakt.tv

E.T.S. DE INGENIERÍA INFORMÁTICA, UNIVERSIDAD DE MÁLAGA

Abstract

In this Final Degree Project, a Web application will be developed. It shows information about movies and television shows to the user, as well as participant in that kind of content.

The system has a set of services and controllers, which follows the software architectural pattern MVC (*Model View Controller*), that has the duty to populate the views with information. So, controllers are the union point between application views and Trakt.tv API. The views allow the users to access different sections of information so as to guarantee the application requirements.

The API is given by Trakt.tv to developers with a huge amount of information about movies and shows. In such a way that users can seek any data about application's topics. This content is likely to become bigger as well as new content appears in order to provide the user with the possibility to access current information.

Keywords

AngularJS, Search, Films, Shows, Characters, Information, cinephile application, API, REST, Trakt.tv

Índice

1 Introducción **1**
- 1.1 Motivación 1
- 1.2 Objetivos 2
- 1.3 Tecnologías 2
- 1.4 Estructura de la memoria 4

2 Planificación **7**

3 Análisis de requisitos **9**
- 3.1 Requisitos funcionales 9
- 3.2 Requisitos no funcionales 10
- 3.3 Requisitos de la API 11

4 Diseño **15**
- 4.1 Organización de la funcionalidad 15
- 4.2 Diseño de interfaces 16
- 4.3 Casos de uso y comportamiento del sistema 16

5 Desarrollo **23**
- 5.1 Herramientas 23
- 5.2 AngularJS 24
- 5.3 Gestión de controladores 26
- 5.4 Gestión de vistas 30
- 5.5 Comunicaciones 34
- 5.6 Otros elementos 39

6 Pruebas **43**
6.1 Karma y Jasmine 43
6.2 Pruebas unitarias 45

Conclusiones **51**

Bibliografía **55**

A Anexos técnicos **57**
A.1 Vista Movies.html 57
A.2 Vista Movie.html 58
A.3 Vista Shows.html 60
A.4 Vista Show.html 61
A.5 Vista Season.html 63
A.6 Vista Person.html 64

Capítulo 1

Introducción

En este capítulo se expondrá la temática principal del Trabajo Fin de Grado (en adelante, proyecto) a realizar, así como los objetivos que se buscan alcanzar, las tecnologías que se plantean utilizar para lograr dicho fin y la estructura de la memoria en sí.

De este modo se le ofrece al lector una visión general sobre el contenido que conformará el cuerpo de la memoria y establecemos las bases de la materia del proyecto en cuestión.

1.1. Motivación

El auge de las nuevas tecnologías es algo evidente hoy en día y entre ellas las aplicaciones Web han ganado bastante popularidad e importancia en nuestras vidas. Tanto es así que vivimos rodeados de aplicaciones que usamos en distintos aspectos del día a día y que no existían hace no muchos años (redes sociales, banca electrónica, comercio electrónico, etc).

Una característica clave es que las aplicaciones Web permiten el acceso a distintos servicios a través de un navegador Web e Internet. Este crecimiento tecnológico ha permitido que los sistemas de información tradicionales pasen a formar parte de la Web, a través de aplicaciones que permiten el tratamiento de la información, así como el acceso a la misma para su uso.

Un sistema de información es un conjunto de elementos orientados al tratamiento y administración de datos e información, organizados y listos para su uso posterior, generados para cubrir una necesidad o un objetivo.

1.2. Objetivos

El objetivo principal de este proyecto es la creación de un Sistema de Información para películas y opcionalmente series de televisión. Para ello se pretende:

- Aprender sobre las distintas tecnologías y lenguajes que se pretenden usar durante la realización del proyecto, ya sea para el desarrollo, control de versiones o diseño de pruebas, entre otros.
- Analizar, diseñar e implementar la aplicación Web que representará el sistema de información para películas (y en caso de que se cuente con tiempo suficiente, para series) utilizando información de una API Rest que ofrece Trakt.tv con datos reales sobre este tipo de contenido.
- Profundizar y practicar los conceptos adquiridos sobre metodologías ágiles, en nuestro caso SCRUM, que se adaptarán para su aplicación durante el desarrollo del trabajo.
- Prestar atención a la calidad del producto software resultante mediante el diseño e implementación de pruebas unitarias para el código de la aplicación.

1.3. Tecnologías

La principal tecnología que se emplea en el desarrollo del proyecto, como su título indica (*"Sistema de Información para películas con AngularJS"*), es AngularJS.

Figura 1.1: AngularJS

AngularJS o Angular es un framework de JavaScript ideado para el diseño front-end de aplicaciones Web de una sola página. Se trata de una tecnología de libre uso que promueve y usa patrones de diseño de software, en concreto el patrón de arquitectura software MVC (*Modelo Vista Controlador*).

La idea fundamental de Angular es mejorar el desarrollo Web de aplicaciones de una sola página y a su vez facilitar el desarrollo de pruebas sobre estos sistemas, reforzando la importancia que deben tener las pruebas en el proceso de desarrollo software.

Las aplicaciones de una sola página o SPA (*Single Page Applications*) son aquellas en las que el contenido de la página se actualiza de forma dinámica, sin ser necesario una nueva carga completa de la misma. De esta forma el usuario interacciona con el sistema y la aplicación se adapta automáticamente conforme a las necesidades presentes. Para ello se emplean una serie de servicios y directivas definidos por Angular, que permiten disociar la manipulación del DOM de la lógica de la aplicación. En los próximos capítulos se procederá a comentar algunos de estos servicios y directivas.

Además de contar con Angular como pilar fundamental de la aplicación, se emplearán otros lenguajes y tecnologías Web de última generación como pueden ser HTML, CSS, Bootstrap, REST, JavaScript, JSON:

HTML (*HyperText Markup Language*) Lenguaje de etiquetado para la elaboración de páginas Web.

CSS (*Cascading Style Sheets*), lenguaje para definir el diseño y la presentación de la apariencia de páginas HTML.

Bootstrap Framework CSS.

REST (*Representational State Transfer*) hace referencia a un conjunto de principios de arquitectura para describir cualquier interfaz entre sistemas que utilice HTTP directamente.

JavaScript Lenguaje de programación interpretado.

JSON (*JavaScript Object Notation*) formato de datos ligero para el intercambio de datos, alternativo a XML.

1.4. Estructura de la memoria

La memoria se constituye de una serie de capítulos en los que se aborda cada parte del proceso de elaboración del proyecto, donde nos podremos encontrar con las siguientes fases:

- **Planificación**. Donde se decidirá qué pasos seguir y cómo hacer el proyecto para lograr los objetivos propuestos y satisfacer los requisitos planteados.
- **Análisis de requisitos**. Donde se mostrará el punto de partida del proyecto, así como la delimitación de sus límites.
- **Diseño**. Donde se analizará cómo debe realizarse el proyecto, y que comportamiento debe seguir la aplicación para cumplir con los requisitos propuestos.
- **Desarrollo**. Donde se explicará el proceso de desarrollo seguido para las distintas partes que conforman la aplicación.
- **Pruebas**. Donde se detallarán las pruebas realizadas y el sistema de realización.
- **Conclusiones**. Donde se expondrán las ideas finales que se obtienen tras la elaboración del proyecto y se hablará de los caminos a seguir para la continuación del proyecto.

- **Referencias bibliográficas**. Documentación de apoyo utilizada para el proyecto.
- **Anexos**. Información complementaria.

Además de las partes mencionadas anteriormente, de las que se tratará en mayor profundidad en los próximos capítulos, cabe destacar que ha todo ello precede una fase de aprendizaje.

En la fase de aprendizaje se llevó a cabo la lectura de diversos libros o capítulos específicos de material alternativo, para adquirir el conocimiento necesario de cada una de las tecnologías o servicios que resultaban necesarios para la elaboración del proyecto. Al mismo tiempo, conforme se profundiza sobre algún aspecto específico de los lenguajes, se llevaban a cabo el desarrollo de pruebas complementarias con pequeños programas, de forma que se pudiese constatar su correcto funcionamiento y afianzar los conocimientos adquiridos.

Capítulo 2

Planificación

El objetivo principal en la planificación es lograr acabar el proyecto para la primera semana del mes de septiembre. Para ello se establece una serie de hitos a cumplir. Se pretende seguir una metodología de desarrollo ágil inspirada en SCRUM, ya que al tratarse de un Trabajo Fin de Grado, no se cuenta con un equipo de desarrollo, ni con todos los perfiles que conforman el equipo establecido por SCRUM.

En este capítulo hablaremos de las partes de la planificación y las fechas orientativas a cumplir, los hitos que se pretenden alcanzar, para llevar un control temporal de cada fase del proyecto y marcar unas fechas, con la meta de intentar garantizar la finalización del mismo en la fecha deseada.

La ejecución del proyecto se comienza el día 25 de junio de 2016, desde donde se plantea la siguiente ejecución:

- Etapa de aprendizaje de las tecnologías, antes del 17 de julio.
- Etapa de análisis y diseño, antes del 24 de julio.
- Etapa de desarrollo, antes del 21 de agosto.
- Etapa de pruebas, antes del 25 de agosto.
- Desarrollo de la memoria, antes del 7 de septiembre.

La consecución de estas etapas no requiere un desarrollo independiente y cronológico, ya que existirán distintos puntos del proyecto en los que se trabaje de manera paralela en algunas de estas fases, intercalando trabajo de ambas entre sí.

Durante la etapa de desarrollo se contará con un pool de requisito, desde el que se elige que requisito implementar durante el sprint de duración variable. De tal forma que finalizado el sprint el requisito se evalua si se ha conseguido completar o por el contrario podemos continuar desarrollando en un nuevo sprint, en cuyo caso volvería al pool de requisitos. Siendo posible dedicar el próximo sprint a otro requisito o volver a continuar con los requisitos no completos

Siguiendo esta planificación se espera lograr la realización del proyecto para el día 1 de Septiembre, contando con un margen de maniobra de 6 días hasta la fecha tope establecida del 7 de septiembre.

Capítulo 3

Análisis de requisitos

En este capítulo procedemos a detallar los distintos requisitos que debe satisfacer la aplicación que se pretende desarrollar, para ello los clasificaremos en tres grupos, requisitos funcionales, requisitos no funcionales y requisitos de la API. De este modo se establecen los objetivos que se buscan alcanzar en el desarrollo de la aplicación y se establece de manera formal los límites del proyecto.

Los requisitos de información no tienen lugar en este proyecto debido a que no se guarda ningún tipo de información en nuestro sistema, toda la información procede de la API de Trakt.tv, página encargada de recopilar y almacenar todo tipo de información referente a películas y series de televisión.

3.1. Requisitos funcionales

Los requisitos funcionales son los que describen la funcionalidad o lógica que debe tener nuestro sistema. De este modo se establecen las posibles acciones que puede llevar a cabo un usuario en nuestra aplicación, definiendo a su vez el comportamiento de la misma.

Los requisitos funcionales que se establecen para el proyecto son los que se pueden ver a continuación:

- **RF0** - Consultar peliculas por popularidad.

- **RF1** - Consultar información de una película.
- **RF2** - Consultar series por popularidad.
- **RF3** - Consultar información sobre una serie.
- **RF4** - Consultar información sobre una persona que participa en una serie o película.
- **RF5** - Consultar información sobre una temporada de una serie.
- **RF6** - Realizar una búsqueda por nombre de películas, series, o personas.
- **RF7** - Filtrar películas.
- **RF8** - Filtrar series.

3.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que describen características o condiciones que debe de cumplir el software, sin llegar a especificar ningún tipo de funcionalidad adicional, ya que esto se recoge en los requisitos funcionales, ni describen información a guardar, ya que eso es objetivo de los requisitos de información.

Los requisitos no funcionales que se establecen para el proyecto son:

- **RNF0** - Facilidad de uso.
- **RNF1** - Interfaces intuitivas.
- **RNF2** - Acceso multiplataforma.
- **RNF3** - Tiempo de respuesta corto.
- **RNF4** - Tolerancia a fallos.
- **RNF5** - Sistema en tiempo real.
- **RNF6** - Manejo de errores.

3.3. Requisitos de la API

Como se ha comentado anteriormente, para poblar de información nuestro sistema usaremos una API externa perteneciente a Trakt.tv. De esta forma la aplicación Web a desarrollar contará con una gran cantidad de información recopilada a lo largo de los años y que día tras día este contenido va en aumento con la adición de nuevos datos.

Figura 3.1: Trakt.tv

La información que podemos encontrar se estructura en objetos bajo el formato JSON, a los que podemos acceder a través de la URL de la propia API *https://api.trakt.tv*, donde todas las operaciones que se realicen deben realizarse sobre SSL (*Secure Sockets Layer, en español Çapa de puertos seguros"*).

Los objetos estándar de información que nos podemos encontrar son *Movie*, *Show*, *Season*, *Episode*, *Person*. Un ejemplo del objeto *Movie* puede ser como el que aparece en el Listado 3.1.

Listing 3.1: Ejemplo de Objeto Movie

```
{
  "title": "Batman v Superman: Dawn of Justice",
  "year": 2016,
  "ids": {
    "trakt": 129583,
    "slug": "batman-v-superman-dawn-of-justice-2016",
    "imdb": "tt2975590",
    "tmdb": 209112
  }
}
```

Por defecto, las consultas que se realicen sobre la API devolverán la mínima información del objeto, como se muestra en el Listado 3.1. Pero también es posible indicar a la API que requerimos más datos acerca de un objeto en cuestión.

Trakt.tv define dos tipos de categorías para la información adicional, que son *full* e *images*, lo que significa que realizando una petición con *full* nos devolvería toda la información referente al objeto, mientras que con *images* se nos proporcionaría todas las imagenes asociadas al mismo, ofreciendo a su vez distintos tipos de foto, de las que cuenta con distintos tamaños para adaptarse a las necesidades del desasrrollador.

Para indicar una de estas categorías o ambas, solo debemos añadir a la URL de la consulta la siguiente expresión "*?extended=categoría*", de modo que si lo que se desea es pedir toda la información e imagenes de la película que aparece en el Listado 3.1, la URL necesaria para realizar la consulta de información extendida quedaría del siguiente modo "*https://api.trakt.tv/movies/1?extended=full,images*", donde el número 1 hace referencia al identificador de la película en cuestión, como aparece en el Listado 3.1.

En algunas situaciones las consultas pueden resultar no ser sobre un objeto en cuestión, sino que por el contrario se requiere una lista de objetos, como por ejemplo las series más populares, por tanto lo que se recibirá no será un objeto JSON, en su lugar será un array (lista de objetos) de objetos. Por defecto, este tipo de consultas nos enviará una lista con 10 objetos, ya que Trakt.tv así lo define. Sin embargo, si se desea obtener más objetos y de hecho dividir la lista por páginas, que por defecto es 1, se le puede indicar a la API del mismo modo que se realizaba anteriormente en las consultas de información adicional.

Para el caso de requerir objetos adicionales debemos añadir como parámetros a la URL la siguiente expresión "*?page=npage&limit=nlimit*", donde *npage* es el número de páginas en la que se desea dividir la lista y *nlimit* el número de elementos por página. Así por ejemplo, si se desease recibir dos páginas con 20 elementos de las series más populares, la URL resultante sería del siguiente modo "*https://api.trakt.tv/shows/popular?page=2&limit=20*".

Cabe destacar que tanto los parámetros de *limit* y *page*, como el parámetro *extended*, se pueden combinar en consultas en las que la API especifiqué su validez. Para más información sobre la API de Trakt.tv puede consultar la documentación [1] oficial en el siguiente sitio Web "http://docs.trakt.apiary.io/#".

Además debemos tener en cuenta que todas las consultas que se realicen sobre la API deben contar con una serie de cabeceras requeridas por Trakt, donde las variables necesarias son:

Content-type Indica el tipo de contenido que se espera recibir, por tanto en este caso su valor ha de ser .ªplication/json", ya que todos los objetos de la API están bajo el formato JSON.

Trakt-api-version Como indica la API debe ser el valor 2.

Trakt-api-key Donde se asocia la clave de la API que proporciona Trakt a cada desarrollador que la solicita. De este modo se realiza una identificación de la aplicación que está realizando las operaciones sobre la API.

Por último, para resumir y expresar de manera más formal los requisitos derivados de la API, mostramos un pequeño listado de los mismos.

- **RA0** - Consultas sobre SSL.
- **RA1** - Uso de parámetros, si se requiere.
- **RA3** - Uso de cabeceras requeridas.

[1]

[1] *Nota para el lector*: En este capítulo se ha empleado una nomenclatura especial para referirnos a los distintos requisitos, por ejemplo *RF0*, esto simplemente quiere decir Requisito Funcional 0, de modo que podamos identificarlos fácilmente en caso de ser necesario referirnos a un requisito en cuestión. Lo mismo ocurre con *RNF0* y *RA0*, Requisito No Funcional 0 y Requisito Api 0, respectivamente.

Capítulo 4

Diseño

En este capítulo trataremos los aspectos más destacables de la fase de diseño para la ejecución del proyecto, desde la estructura que se desea seguir para organizar la funcionalidad, hasta los patrones de diseño que se seguirán en la creación de interfaces de la aplicación, seguido de los casos de uso junto con el comportamiento esperado del sistema.

Esta fase resulta una parte muy importantes del proyecto ya que establece las bases del trabajo a realizar a lo largo del desarrollo de la aplicación Web, ya que expondremos los distintos puntos que conciernen al diseño, mostrando las decisiones tomadas sobre cómo organizar la aplicación, de acuerdo al aspecto visual y funcional marcados como objetivos.

4.1. Organización de la funcionalidad

Al tratarse de una aplicación de una sola página, ya que hablamos de un sistema desarrollado con AngularJS, contaremos con una página de elemento central (*index.html*), en la que estará asociado el módulo de nuestra aplicación que contendrá todo el apartado de funcionalidad (*myApp.js*).

Aprovechando los servicios que nos ofrece este framework de desarrollo Web, organizaremos la funcionalidad referente a cada una de las vistas en un controlador independiente para cada uno de las interfaces. De este modo contamos con

una serie de vistas con su controlador asociado, lo que nos permite el desarrollo y prueba de cada una de estas vistas de forma aislada a las restantes partes del sistema.

De esta manera también podemos organizar las distintas configuraciones que debemos establecer sobre el servicio $resource para realizar las peticiones oportunas de información, requerida por cada vista y cada consulta en particular.

4.2. Diseño de interfaces

En cuanto a las interfaces, se ha optado por un diseño simple e intuitivo, para que permita a cualquier usuario usar el sistema de manera sencilla. Para ello contamos con un diseño que se divide en parte estática (común a todas las vistas) y parte dinámica que cambia en función de la interacción del usuario con la aplicación.

En la parte estática nos encontramos con la barra de navegación, que debe permitir el movimiento del usuario a las distintas secciones que ofrece la Web, como son página principal, películas, series, contacto y breve resumen acerca de la aplicación.

Mientras que en la parte dinámica nos encontramos con un cuerpo central que cambia en función de las vistas requeridas, de modo que cada una de las vistas está almacenada en un archivo HTML independiente, donde se encuentra la parte de código que se requiere inyectar en la página principal, así mantenemos un control independiente de cada una de las vistas, facilitando su desarrollo y testeo.

4.3. Casos de uso y comportamiento del sistema

Los casos de uso presentes son compartidos por cualquier usuario que utilice la aplicación, ya que no se realiza una distinción de roles de acceso, por el hecho de que no se encuentra ningún tipo de funcionalidad presente en el sistema que

requiera de ello.

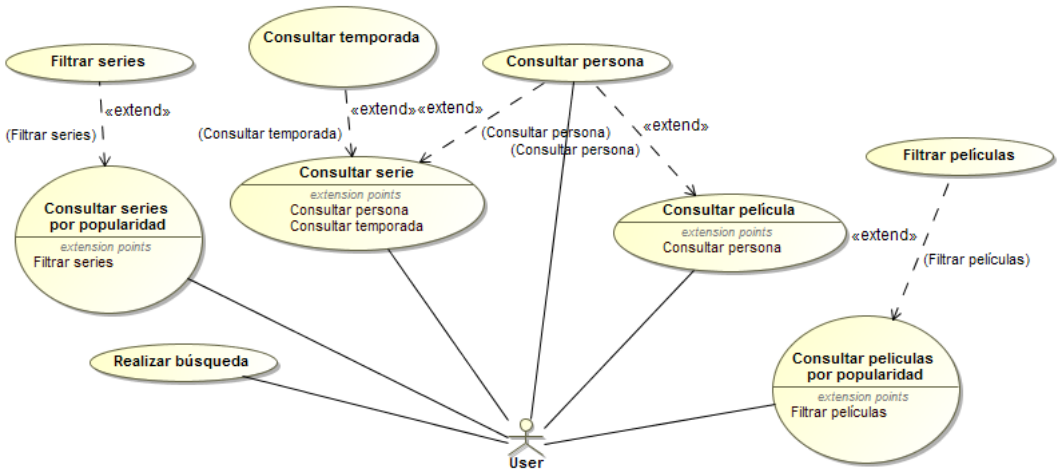


Figura 4.1: Casos de uso

Consultar Películas Cuando el usuario acceda al apartado de películas el sistema deberá mostrar un listado con las películas que están de moda, siendo posible la petición de películas populares o películas que están siendo vistas.

Filtrar Películas Si el usuario lo desea podrá realizar un filtrado de las películas mostradas en el sistema en función de su género, además en la categoría watched se puede pedir un filtrado por períodos de tiempo, como pueden ser semana, mes, año.

Consultar Película Cuando el usuario quiera consultar información de una película, ya sea porque realiza una búsqueda, porque selecciona una película de la sección películas o por hacer clic sobre una película en la que apareció un actor, debe mostrar toda la información posible sobre la misma, presentando la información de manera organizada.

Consultar Series Cuando un usuario accede al apartado de series el sistema deberá mostrar un listado con las series que están de moda, siendo posible la petición de series más populares o series que están siendo vistas.

Filtrar Series Si el usuario lo desea podrá realizar un filtrado de las series mostradas en el sistema en función de su género, además en la categoría watched se puede pedir un filtrado por períodos de tiempo, como pueden ser semana, mes, etc.

Consultar Serie Cuando un usuario quiera consultar información de una serie, ya sea porque realiza una búsqueda, porque selecciona una serie de la sección series o por hacer clic sobre una serie en la que apareció un actor, debe mostrar toda la información posible sobre la misma, presentando la información de manera organizada.

Consultar Temporada Cuando el usuario hace clic sobre una temporada de una serie, el sistema debe mostrar toda la información posible sobre la temporada en cuestión y presentarla de manera adecuada.

Consultar Persona Cuando un usuario quiera consultar información sobre una persona en cuestión, ya sea por realización de búsqueda, por seleccionar un actor que apareció en una película o seleccionar un actor que apareció en una serie, el sistema debe mostrar toda la información posible acerca de la persona, de manera organizada.

Realizar búsqueda Cuando el usuario quiere consultar directamente una película, serie o persona puede realizar una búsqueda por nombre y categoría desde el buscador de la barra de navegación.

Con el objetivo de ilustrar un poco el comportamiento que pueden surgir de los casos de uso vamos a comentar unos posibles escenarios de consultar una película, de manera que el lector pueda apreciar las interacciones con el sistema y los flujos de información.

Escenario normal 1.

1. El usuario accede al sistema
2. Se accede a la sección películas
3. El sistema realiza una petición a la API de películas de moda
4. La API envía la información y se muestra
5. El usuario selecciona una película
6. El sistema realiza una petición a la API de la película
7. La API envía la información y se muestra

Escenario normal 2.

1. El usuario accede al sistema
2. Se realiza una búsqueda
3. El sistema realiza una petición a la API sobre la búsqueda
4. La API envía información y se muestra
5. El usuario selecciona una película
6. El sistema realiza una petición a la API de la película
7. La API envía la información y se muestra

Escenario normal 3.

1. Se realizan los pasos del escenario normal 1
2. El usuario selecciona un actor del listado de personajes
3. El sistema envía una petición a la API de la persona
4. La API envía la información y se muestra
5. El usuario hace clic sobre una película en la que participó la persona
6. El sistema envía una petición a la API de la película
7. La API envía la información y se muestra

También pueden aparecer escenarios en los que se produzca un error, por ejemplo si la API de Trakt.tv no se encuentra activa por algún motivo, se producirá un error, ya que la información requerida no estaría disponible.

Escenario alternativo 1.

1. El usuario accede al sistema
2. Se accede a la sección películas
3. El sistema realiza una petición a la API de películas de moda
4. La petición no se resuelve satisfactoriamente
5. El sistema genera una excepción controlada
6. Se muestra al usuario un mensaje de error

De modo ilustrativo también exponemos un diagrama de máquinas de estado sobre la vista *shows.html*, donde podemos apreciar las distintas interacciones y cambios que se producen en esta vista en particular, donde el usuario cuenta con la opción de listar las series que se encuentran actualmente de moda (Trending), las series más populares a lo largo del tiempo o las series que están siendo vistas últimament (Watched).

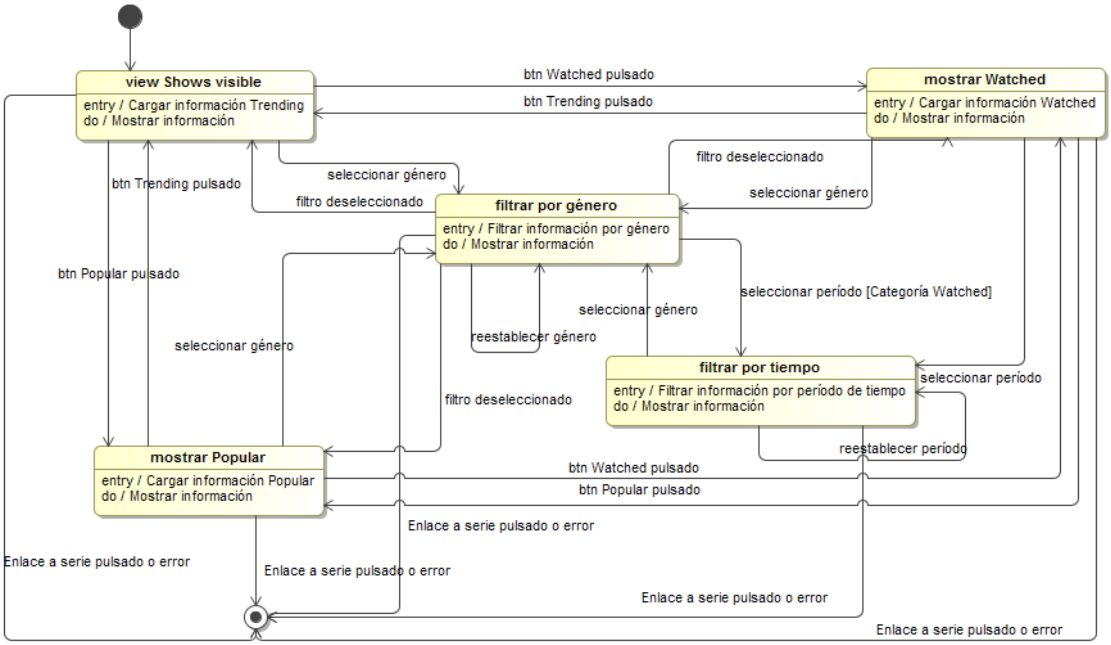


Figura 4.2: Máquina de estados de la vista *shows.html*

Capítulo 5

Desarrollo

En este capítulo hablaremos sobre los distintos pasos que se han seguido en el desarrollo de la aplicación. Además se explicará con más detalle distintos servicios que se han usado de AngularJS, así como servicios creados para facilitar las tareas a realizar por algún tipo de funcionalidad deseada.

Del mismo modo se expondrán ciertos listados de código para ilustrar los procedimientos y usos de AngularJS.

5.1. Herramientas

Durante la fase de desarrollo se han utilizado una serie de programas para facilitar su realización, entre los que podemos encontrar:

Sublime Text 2 Editor de texto recomendado para desarrolladores, ya que aparte de cumplir con las funcionalidades básicas de cualquier editor de textos, cuenta con un sistema de plugins y reconocimiento de sintaxis, que lo convierten en una herramienta muy potente.

Postman Aplicación de Chrome específica para APIs REST. Con postman es posible realizar operaciones sobre una API y comprobar el funcionamiento de la misma, así como ver directamente las respuestas recibidas a ciertas peticiones, los formatos de los datos, estado de las peticiones, etc.

Esta aplicación nos ha brindado la posibilidad de testear y analizar las operaciones implementadas en la aplicación y comprobar su correcto funcionamiento antes de ser implantadas en la misma

LiveReload Aplicación de escritorio (aunque también existe un plugin para Sublime Text) encargada de registrar los cambios realizados sobre un archivo y auto refrescar las páginas asociadas al archivo editado en el navegador Web.

Esta herramienta ha resultado especialmente útil en la elaboración de vistas de la aplicación.

Git Herramienta para el control de versiones.

El control de versiones ha resultado muy útil para la etapa de desarrollo, permitiendo llevar un mejor control de los distintos hitos alcanzados.

Al igual que con otras partes del aprendizaje que nos resultaron más interesantes, nos hubiese gustado profundizar más en este aspecto, pero la planificación preestablecida para la realización de cada fase del proyecto no permitía emplear más tiempo para profundizar en mayor detalle.

Deployd Herramienta para la creación de APIs REST de forma fácil y práctica.

Con esta herramienta se pudo crear una API propia con la que realizar pruebas, con el fin de comprender el funcionamiento de los servicios *$http* y *$resource* durante la fase de aprendizaje.

5.2. AngularJS

Ya hemos comentado qué es Angular, pero aún no hemos hablado de los conceptos básicos de este framework. Esta sección está dedicada a realizar un pequeño paseo por Angular, por lo que si el lector ya conoce cómo trabajar con esta tecnología, puede saltarse este apartado y continuar con la sección de "Gestión de controladores".

Angular ofrece una serie de etiquetas o directivas que permiten extender el uso de HTML, de tal forma que podemos incluir mayor funcionalidad o usar elementos que antes no eran posibles. Las directivas en Angular, como ya hemos

comentado, son etiquetas, etiquetas especiales que indican al compilador que su funcionamiento proviene de las librerías de Angular.

Además otro aspecto importante es la no necesidad de manipulación del DOM (*Document Object Model*) para enviar información desde el modelo a la vista a través de los controladores. Simplemente se emplea el objeto *$scope* que se encarga de hacer accesible la información de las variables o funciones que declaremos en su ámbito.

Para ilustrar algunas de las partes fundamentales de toda aplicación en Angular podemos ver el Listado 5.1, donde se muestra un código de ejemplo de una aplicación sencilla.

Listing 5.1: Código de aplicación simple

```
<html ng-app="myApp">
    <head>
        <title>Ejemplo</title>
        <script src="js/angular.min.js"></script>
        <script>
            var app = angular.module("myApp",[]);

            app.controller("MyCtrl", function ($scope) {
                var someData = {
                    firstName: 'JENNA',
                    surname: 'GRANT'
                };
                $scope.data = someData;
            });
        </script>
    </head>
    <body ng-controller="MyCtrl">
        <p>
        <strong>First Name</strong>: {{data.firstName}}<br/>
        <strong>Surname:</strong> {{data.surname}}
        </p>
    </body>
</html>
```

Uno de los primeros pasos en Angular es crear un módulo en el que desarrollar, para ello se emplea el método *angular.module*, asignándole un nombre e inyectando dependencias de módulos independientes al principal con el que cuenta Angular. Como es un ejemplo sencillo no es necesario módulo adicional y por tanto queda vacío este argumento "[]". Por ejemplo, en caso de querer usar

ngResource, como explicaremos en secciones posteriores, solo habría que añadir ["ngResource"].

Una vez contamos con el módulo, podemos crear controladores asignados a ese módulo con el método *.controller*, y a partir de ahí podemos comenzar a desarrollar funcionalidad para nuestro sistema. La manera de asociar módulos y controladores a las páginas HTML es mediante las etiquetas *ng-app* y *ng-controller*, de tal forma que el ámbito de acción del mismo se extenderá desde el elemento en el que la hemos incluido hasta sus sucesivos hijos.

Para acceder a la información enviada a través del objeto *$scope* hacia la vista solo tenemos que usar la expresión {{ }}, estas dobles llaves indican al compilador la presencia de variables o funciones de Angular.

Además de crear controladores en nuestro módulo, se pueden crear constantes con el método *.constant(* identificador , valor *)*, o crear servicios personalizados con *.factory(* identificador , función *)* o *.service(* identificador , función *)*.

5.3. Gestión de controladores

En esta sección hablaremos sobre los distintos controladores que se han utilizado para ofrecer funcionalidad en las distintas páginas que componen la aplicación final, como del modelo de los datos.

Como comentamos en apartados anteriores, la idea principal de Angular está centrada en el patrón de arquitectura software MVC (*Modelo Vista Controlador*). Dentro de estas tres partes diferenciadas, lo que se busca es separar de manera visible la representación final de los datos hacia el usuario, de la lógica de la aplicación, facilitando de esta manera el desarrollo de la presentación y la funcionalidad.

Inicialmente hablaremos de los componentes *Modelo* y *Controlador*, ya que están más relacionados con la temática de la sección actual y posteriormente se tratará el elemento de la *Vista*, para así completar el recorrido por el patrón de

diseño MVC.

Atendiendo a la definición de MVC, el *Modelo* es la representación de la información usada en el sistema, por lo tanto gestiona todos los accesos a los datos en cuestión y envía a la *Vista* la información que en cada momento se necesita para que sea mostrada hacia el usuario final.

Mientras que el *Controlador* es el encargado de responder a los eventos que se originan en la aplicación por parte de la interacción del usuario o cualquier otro tipo de evento relacionado. Estas peticiones se envían al *Modelo* cuando se requiere alguna parte de la información con la que este cuenta (por ejemplo, listar las películas más populares). Al ser el nexo de unión entre el *Modelo* y la *Vista*, se encarga de enviar la información a la *Vista* asociada y si fuese necesario también puede enviar cierta información de control que se encargue de adaptar la *Vista* para determinados tipos de representación de la información.

Durante el desarrollo del proyecto se han creado un total de 11 controladores, cada uno de ellos asociado a su vista. Un listado de los controladores con su vista asociada sería como el que sigue:

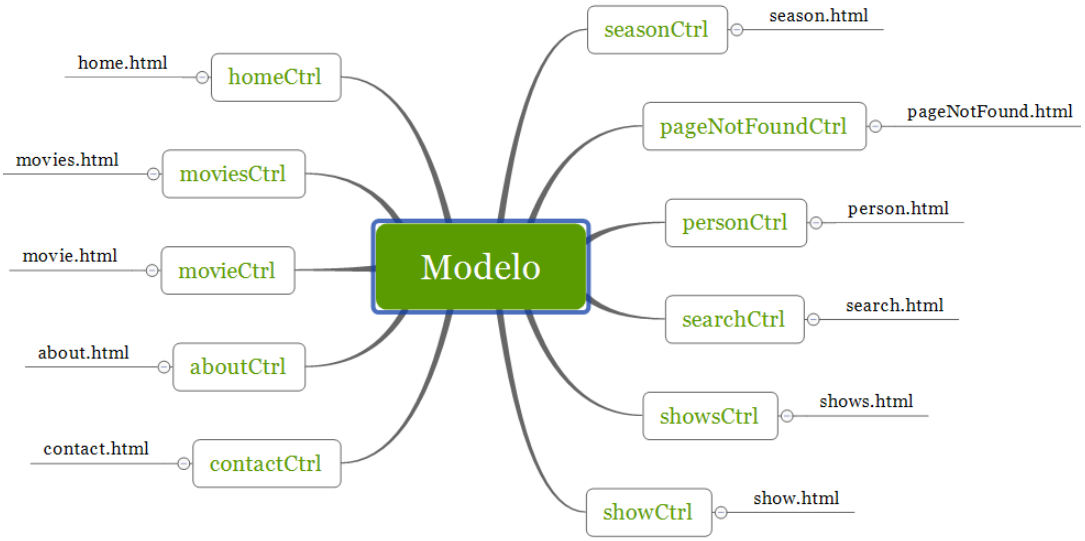


Figura 5.1: Asociación de controladores y vistas

Para la asociación de controladores a sus vistas y el cambio dinámico de uno a otro, conforme el usuario interacciona con la aplicación, aparece el módulo de Angular ngRoute.

NgRoute es un módulo de Angular que provee de una serie de servicios y directivas para la creación y configuración de rutas en aplicaciones desarrolladas con este framework. Uno de estos servicios que lo conforman es *$routeProvider*, es un servicio que nos permite establecer las relaciones entre controladores y vistas, así como los cambios de uno a otro en función de la URL actual.

Un ejemplo de la configuración que se realiza con ngRoute es el que podemos ver en el Listado 5.2. De igual modo se procede a declarar todos los cambios posible de la URL, asociando así cada controlador a su vista.

Listing 5.2: Código parcial de la configuración de ngRoute

```
app.config(function ($routeProvider) {
    // configure the routes
    $routeProvider

    .when("/", {
        // route for the home page
        templateUrl: "pages/home.html",
        controller: "homeController"
    })

    .when("/movies", {
        // route for the movies page
        templateUrl: "pages/movies.html",
        controller: "moviesController"
    })
});
```

En concreto, como ya se mencionó en anteriores capítulos, las aplicaciones desarrolladas con Angular son aplicaciones de una sola página o SPA (*Single Page Applications*), de forma que cuando cambiamos de una página a otra, la aplicación no recarga por completo la página, sino que el contenido cambia dinámicamente.

Al igual que ocurre con el controlador y la URL, la *Vista*, cambia su contenido en función de la URL actual mediante el uso de ngRoute. Para esto interviene

una directiva presente en el módulo ngRoute, la directiva ngView, de tal forma que en la página principal de nuestra aplicación (página que contendrá el contenido estático de la aplicación, como por ejemplo barra de navegación) se indica el lugar donde se inyectará el código HTML requerido por cada vista. Un ejemplo de ello es el que se puede apreciar en el Listado 5.3.

Listing 5.3: Código parcial de *index.html*

```
<div id="main">
    <!-- La directiva ngView indica donde se inyectará el código -->
    <div ng-view></div>
</div>
```

El código del Listado 5.3, se encuentra presente en nuestra página principal, *index.html*, donde podemos encontrar el código de la barra de navegación y el código del Listado 5.3, que indica el lugar exacto donde el contenido se cambia dinámicamente.

Una vez asociados los controladores a las vistas, cada uno de ellos actuará cuando su vista sea la vista actual. De tal forma que de manera independiente desarrollamos y controlamos la funcionalidad de cada página de la aplicación. En el Listado 5.4 podemos ver un ejemplo de uno de los controladores que forman parte del sistema desarrollado.

Listing 5.4: Código parcial de moviesController

```
app.controller("moviesController", function ($scope, $http, $resource, baseUrl,
    ↪$location, ShareFactory, SearchFactory) {

    var trendingResourceImages= $resource(baseUrl + "movies/trending?extended=images&
        ↪page=1&limit=20");
    var resourceFilmFullImages = $resource(baseUrl + "movies/:trakt?extended=full,images
        ↪",{trakt:"@trakt"});

    $scope.getTrending = function () {
        var aux = trendingResourceImages.query();
        aux.$promise.then(function (data){
              $scope.peliculas = data;
              $scope.view= "normal";
              $scope.message= "Trending";
         })
         .catch(function (error){
              $location.path("error");
```

```
        })
        .finally(function (){
            $scope.spinner = false;
        });
    }

    $scope.mostrarPelicula = function (film) {
      var item = resourceFilmFullImages.get(film.movie.ids);
      item.$promise.then(function (data){
          ShareFactory.setShareItem(data);
          $location.path("movie");
      })
      .catch(function (error){
          $location.path("error");
      });
    }

    $scope.lookUp = function (search) {
      SearchFactory.lookUp(search);
      $location.path("search");
    }
});
```

5.4. Gestión de vistas

Esta sección se centra en describir las distintas vistas a las que tiene acceso el usuario y cómo se organizan entre sí, para completar así el recorrido del MVC, donde el elemento objeto de explicación es la *Vista*.

Atendiendo nuevamente a la definición del patrón MVC, la *Vista* presenta de forma adecuada el *Modelo* (información del sistema) en un formato adecuado para el usuario de manera que este pueda interactuar con los datos y con la aplicación de la forma que marca la funcionalidad del sistema. En todo momento las acciones que se realicen serán enviadas hacia el *Controlador* que será el encargado de gestionar los cambios necesarios para que la *Vista* se adapte a las necesidades del usuario en cada momento, atendiendo a la lógica existente en el sistema.

Como vimos en la sección anterior, el módulo ngRoute se encarga de asociar controladores a vistas y establecer cuál será el actual en función de la URL del momento. Para que esto ocurra deben de registrarse cambios en la URL y es

en ese momento cuando aparece otro servicio de angular denominado *$location*.

Antes de describir este servicio vamos a explicar el sistema de URL de Angular, ya que se utiliza una URL en modo hashbang, como podemos apreciar en la Figura 5.2.

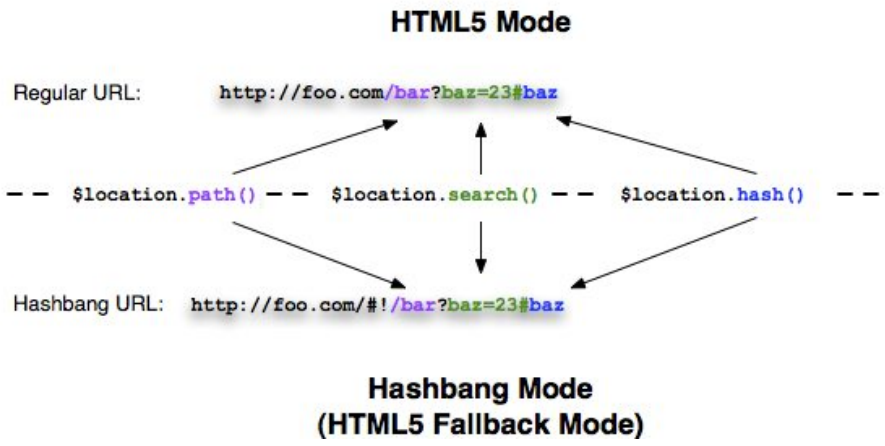


Figura 5.2: AngularJS URL

En este tipo de URL las partes de la misma se mantienen, añadiendo un separador. Como muestra la Figura 5.2, el primer hashtag ("#") muestra la separación entre la URL base de la aplicación, en nuestro caso sería la página *index.html*, y la parte que se modifica dinámicamente, que en nuestro caso solo será la parte de path. El uso de otras secciones de la URL se encuentra disponible en este tipo de enlace, al igual que en los normales, pero para nosotros no ha sido necesario usarlas.

$location es un servicio de angular que se encarga de parsear la URL del navegador y hacerla accesible desde la aplicación. Cualquier cambio que se produzca en la URL se verá reflejado en el servicio y viceversa. Por lo que existen dos formas de cambiar la URLs usadas en nuestro sistema.

En primer lugar mediante la barra de navegación, como podemos apreciar en el Listado 5.5, mientras que la otra forma es usar el servicio *$location*, para ello desde los métodos del controlador en los que se requiere un cambio de vista

se emplea el método *path*, de tal forma que si queremos cambiar a la vista *movie.html*, se escribiría $location.path("movie").

Listing 5.5: Código parcial de la barra de navegación

```
<ul class="nav navbar-nav navbar-right">
   <li><a href="#/"><i class="fa fa-home" aria-hidden="true"></i> Home</a></li>
   <li><a href="#movies"><i class="fa fa-film" aria-hidden="true"></i> Films</a></li>
   <li><a href="#shows"><i class="fa fa-ticket" aria-hidden="true"></i> Shows</a></li>
</ul>
```

El método *path* sin argumentos, es un método del tipo GET, o dicho de otro modo es un método que nos devuelve el valor de la parte path de la URL, como vimos en la Figura 5.2, mientra que *path* con argumento, es un método del tipo SET, que quiere decir, que se cambia el valor del apartado path de la URL por la cadena que se pasa por argumento.

En cuanto a las vistas, nos encontramos con un total de 11 páginas HTML, donde *index.html* es nuestra página principal. De hecho es la página que define aspectos como la inyección dinámica de contenido HTML según la vista que requiera el usuario. Además se encarga de designar el ámbito del controlador y el ámbito del módulo creado para ofrecer funcionalidad en el sistema, que en nuestro caso se llama app.

Entre el conjunto de vistas de la aplicación Web nos encontramos con una serie de páginas HTML enfocadas a satisfacer los requisitos establecidos durante el capítulo 3, además de algunas páginas adicionales para mostrar información adicional. El conjunto de páginas HTML está formado por:

Index Se trata de la página principal de la aplicación, en ella se define el contenido estático del sistema y además se incluyen las directivas para definir el ámbito de módulos y controladores, así como el lugar donde se producirá la inyección de código HTML en función de las vistas requeridas.

Movies En esta vista el usuario verá por defecto las películas que están de moda o dicho de otro modo películas trending.

Aparecen en la parte superior tres botones que nos permiten cambiar el contenido del listado por las películas más populares a lo largo de los años o simplemente ver que películas se están viendo últimamente. Además a estos listados podemos realizar un filtrado por géneros y en la categoría watched también por período de tiempo.

Si el usuario desea ver información sobre una película en cuestión, solo tiene que hacer clic sobre la imagen de la película que desea conocer o sobre su título.

Movie Esta página ofrece información sobre una película, desde su valoración, título, sinopsis, . . . , hasta información sobre las personas que han participado en la misma y comentarios sobre el largometraje.

Si el usuario lo desea se incluye el trailer de la película que puede ser reproducido en la propia página de la aplicación. Además si quiere conocer más detalles acerca de algún actor en particular, puede realizar clic sobre la imagen o nombre de la persona deseada.

Shows Al igual que ocurre con la página *movies.html*, en esta vista podemos consultar los listados de series en función de la categoría popular, trending o watched, filtrar la información. Así como consultar más información acerca de una serie en cuestión.

Show La página *show.html* es la encargada de mostrar toda la información relevante de una serie, ofreciendo datos similares a los que oferta la página *movie.html*, incluyendo en esta las distintas temporadas con las que cuenta una serie, por lo tanto la consulta sobre una temporada en cuestión se encontrará también accesible a través de su imagen o título, como ocurre con los actores participantes.

Season En la vista *season.html* el usuario puede ver los datos referentes a la temporada seleccionada de una serie e información sobre los capítulos de la misma, datos que incluyen una descripción del capítulo, la valoración de los usuarios, entre otros.

Person La página *person.html* se encarga de presentar los datos referentes a una persona, donde nos podremos encontrar su información básica, una

pequeña biografía y una serie de películas y/o series en las que haya participado, siendo posible el acceso a más información sobre las películas o series que aparezcan entre ese contenido.

Search Sirve como vista auxiliar para mostrar los resultados obtenidos al realizar algún tipo de consulta a través del buscador con el que cuenta la aplicación.

Las consultas realizadas se pueden hacer sobre todo el contenido de la API, o por el contrario si se conoce y se desea especificar el tipo de contenido que desea buscar el usuario, se puede establecer su categoría para solo requerir elementos coincidentes con la categoría seleccionada. Estos elementos pueden ser película, serie o persona.

Contact En esta vista nos encontramos con información sobre el desarrollador del sistema y su información de contacto por si el usuario quiere contactar con el mismo por algún tema en cuestión.

About En esta página nos encontramos con información sobre el sistema desarrollado mediante una breve descripción, así como la exposición de las motivaciones que llevan a desarrollar esta aplicación.

PageNotFound Se trata de una página auxiliar, que se mostrará en caso de intentar acceder a una URL no existente en el sistema desarrollado, mostrando un mensaje de error por la excepción producida.

5.5. Comunicaciones

Una de los aspectos fundamentales de la aplicación es la comunicación que se establece con la API de Trakt.tv, ya que es el lugar donde se obtiene toda la información que se muestra en el sistema. Esta sección se centra en la explicación de cómo funcionan las comunicaciones con la API e ilustrar qué servicios de Angular son los responsables de ello.

Como ya se mencionó en capítulos anteriores, la información que se presenta en nuestra aplicación Web proviene de la API de Trakt.tv. Dicha API emplea REST, un estilo de arquitectura software para sistemas hipermedia distribuidos.

Figura 5.3: Conjunto de operaciones de REST

REST establece una serie de acciones u operaciones bien definidas, entre las más habituales GET, PUT, POST, DELETE. Estas acciones se suelen asociar con distintas acciones básicas que se pueden dar en un sistema software, donde GET tendría la función de obtener información sobre algo, PUT y POST suelen referirse a la creación o modificación de un objeto en particular, mientras que por último, DELETE se asocia a la eliminación de un ítem.

Además REST es un protocolo cliente/servidor sin estado, lo que quiere decir que ni cliente ni servidor guardan un estado de las comunicaciones tras una petición o interacción. Para el intercambio de información entre ambos se emplea el protocolo HTTP, donde los recursos que se van a tratar son únicamente identificables a través de su URI. De esta forma es posible la navegación de un recurso a otro siguiendo enlaces.

En nuestro caso, al tratarse de un sistema que se dedica a la presentación de información hacia el usuario, el método principal que usaremos será GET, para la obtención de los datos necesarios desde la API. Dentro del uso de los métodos proporcionados por REST, existe la posibilidad de crear acciones customizadas para la adaptación de dichas operaciones a las necesidades de cada API y/o sistema.

Desde Angular se ofrecen dos servicios para las comunicaciones HTTP, el

primero de ellos *$http* y el segundo *$resource*. Cabe destacar que *$http* se encuentra en el módulo principal de Angular, mientras que *$resource* cuenta con un módulo independiente para este servicio, llamado ngResource.

En particular *$resource* es un servicio de más alto nivel que *$http*, que encapsula a este mismo. *$resource* se diseño con el objetivo de cubrir mejor las necesidades para interacciones con servicios REST. Al estar enfocado a ello y servir de abstracción a las peticiones HTTP, hemos optado por usar *$resource* en lugar de *$http*, aunque se realizaron diversas pruebas con el servicio descartado.

Como podemos observar en el Listado 5.6, el módulo ngResource cuenta con un conjunto de acciones por defecto, con la posibilidad de crear nuevas operaciones personalizadas empleando los métodos de REST.

Listing 5.6: Acciones por defecto de *$resource*

```
{ 'get':    {method:'GET'},
  'save':   {method:'POST'},
  'query':  {method:'GET', isArray:true},
  'remove': {method:'DELETE'},
  'delete': {method:'DELETE'} };
```

Entre estas acciones, los métodos *get* y *query*, basados en el método GET son los que nos proporcionan el acceso a los datos necesarios de la API. El método *get* envía una petición sobre un objeto en concreto, por lo que en la respuesta se espera un objeto JSON, mientras que el método *query* solicita una serie de objetos determinados, por lo que la respuesta resultante contendrá un array (o lista) de objetos JSON.

Antes de emplear dichas operaciones, se debe realizar una configuración previa de *$resource* y *$http*. Cómo *$resource* encapsula a un nivel de abstracción mayor el servicio *$http*, para cumplir con el requisito RA3, de uso de cabeceras en las operaciones sobre la API, debemos usar el servicio *$httpProvider*, que no es más que un servicio que nos permite configurar ciertos aspectos con los que contarán los mensajes HTTP que se utilicen.

En el Listado 5.7 podemos apreciar cómo se realiza la asignación de cabeceras para que todos los mensajes HTTP cuenten con estos parámetros por defec-

to. Donde apiKey, contentType y apiVersion son constantes que hemos definido con los valores necesarios, indicados en el apartado de requisitos de la API en el capítulo 3, para cumplir con los requerimientos de Trakt.tv.

Listing 5.7: Código de configuración de cabeceras para HTTP

```
app.config(function ($httpProvider, apiKey, contentType, apiVersion){
    $httpProvider.defaults.headers.common["content-type"]= contentType;
    $httpProvider.defaults.headers.common["trakt-api-version"]= apiVersion;
    $httpProvider.defaults.headers.common["trakt-api-key"]= apiKey;
});
```

Una vez establecidas las cabeceras, el siguiente paso es definir el recurso *$resource* para cada uno de los tipos de peticiones que se pretenden realizar, para ello cada controlador cuenta con una serie de variables que contienen un recurso en particular para un tipo de petición específica. Para ilustrar esto podemos ver un ejemplo en el Listado 5.8, sobre las peticiones de un solo objeto y en el Listado 5.9 veremos cómo se utiliza la operación *query* para esperar listas de objetos. Ambos códigos pertenecientes al controlador moviesController.

Listing 5.8: Código para petición de una película

```
    var resourceFilmFullImages = $resource(baseUrl + movies/:trakt?extended=full,images"
        ↪,{trakt:"@trakt"});

    $scope.mostrarPelicula = function (film) {
        var item = resourceFilmFullImages.get(film.movie.ids);

        item.$promise
          .then(function (data){
              ShareFactory.setShareItem(data);
              $location.path("movie");
          })

          .catch(function (error){
              $location.path("error");
          });
    }
```

Listing 5.9: Código para petición de un conjunto de películas

```
    var trendingResourceImages = $resource(baseUrl + "movies/trending?extended=images&
        ↪page=1&limit=20");
```

```
$scope.getTrending = function () {
    var aux = trendingResourceImages.query();

    aux.$promise
     .then(function (data){
          $scope.peliculas = data;
          $scope.view= "normal";
          $scope.message= "Trending";
     })

     .catch(function (error){
          $location.path("error");
     })

     .finally(function (){
          $scope.spinner = false;
     });
}
```

Como se aprecia en el Listado 5.8 y Listado 5.9, para definir el recurso para peticiones se emplea el servicio *$resource*, con la posibilidad de incluir uno o dos argumentos. El primero de ellos define la URL del recurso a consultar o la base del mismo, por lo tanto puede incluir parámetros o no. Si la URL incluye parámetros que identifican objetos dentro de un conjunto, entonces como segundo argumento se incluye los atributos del objeto que contendrán los valores de dichos parámetros.

Existe la posibilidad de añadir un tercer argumento, donde indicar la creación de operaciones personalizadas en base a los métodos de REST, pero en nuestro caso no ha sido necesario, ya que las operaciones por defecto *get* y *query* cubren todas las necesidades de la aplicación.

Todos los métodos de *$resource* devuelven inicialmente una promise de Angular. Las promises son funciones asíncronas, debido al carácter asíncrono de las peticiones HTTP (esto quiere decir que no se sabe con certeza cuando una petición estará resuelta). Estas funciones devuelven el valor de las respuestas o las excepciones que han ocurrido durante la comunicación, para ello cuentan con tres métodos a usar. *Then* nos devuelve el valor de la respuesta en caso de que se haya producido una comunicación sin errores, la función *catch* se ejecuta cuando se han producido errores y *finally* se ejecuta siempre que una promise está resuelta, sea cual sea su estado.

Lo normal en las peticiones GET, es no requerir de ninguna de estas funciones, a no ser que se desee hacer esperar la aplicación hasta la recepción de los datos. Véase por ejemplo el caso del Listado 5.8, donde se espera la resolución de la petición para poder avanzar hacia otra vista y que esta se muestre con toda la información, sin que se dé el caso de que la vista se cargue sin información.

Sin embargo nosotros hemos decidido usar los método *then* y *catch*, para garantizar la correcta recepción de los datos requeridos y en caso contrario, mostrar un mensaje de error en la página. De manera que cuando se ejecuta la función *catch*, por error en la resolución de la promises, lanzamos una excepción controlada intentando acceder a una URL inexistente, de esta forma aparece la página *pageNotFound.html* con un mensaje especial para los casos en los que hayamos sido los responsables de lanzar la excepción.

Además en la opción de búsqueda de películas "Trendingz series "Trending", resultados por defecto en las sección movies y shows al acceder a sus respectivas vistas, empleamos el método *finally*, como podemos apreciar en el Listado 5.9, para finalizar la animación de un spinner de carga , activo hasta que las peticiones han sido resueltas.

El uso de las funciones asociadas a las promises suele aparecer cuando se trata con operaciones como DELETE, PUT o POST, para garantizar en todo momento la sincronía entre los datos que ve el cliente y los datos almacenados en la API, ya que un error en estos métodos puede significar que un ítem no se ha eliminado correctamente, que no se ha llevado a cabo una modificación o que el objeto que se pretendía crear no existe aún en la base de datos.

5.6. Otros elementos

A lo largo de esta sección vamos a hablar de varios servicios y directivas de Angular que también han intervenido durante el desarrollo de la aplicación Web, para representar también aquellos elementos que no tienen un papel tan destacable como pueden ser los controladores o las comunicaciones, pero que

también suponen de bastante importancia, facilitando el desarrollo del sistema.

En primer lugar vamos a tratar la directiva *ng-repeat*, este es uno de los más usados, ya que su finalidad es tomar un array o lista desde el objeto *$scope* y mostrarlo en la vista. Para ello lo que realiza es una copia de todo el código HTML que se encuentra entre el elemento donde se introduce la etiqueta, hasta su último hijo. En el Listado 5.10 podemos ver un ejemplo de presentación de las películas más populares, para ello *ng-repeat* copia el mismo código HTML para todos los elementos de la lista, y muestra para cada uno de ellos la información de su ítem asociado.

Listing 5.10: Código parcial de *movies.html*

```
<div class="col-md-3" ng-repeat = "item in peliculas" ng-show = "view == 'popular'">
    <a href="" ng-click="mostrarPeliculaPopular(item)">
        <img ng-src="{{item.images.poster.thumb}}" width="100%" height="300px" id="
            ↪photoBorder"></img>
    </a>
    <div class="text-center">
        <h4>
            <a class="text-muted" href="" ng-click="mostrarPeliculaPopular(item)">{{item
                ↪.title}}</a>
        </h4>
         <p><small>{{item.year}}</small></p>
    </div>
</div>
```

Otra directiva interesante que podemos apreciar en el Listado 5.10 es *ng-show*, la funcionalidad de ng-show evalúa la expresión que se le pasa como argumento, siendo posible la utilización de variables de Angular, como en ese caso, view que es una variable declarada en el ámbito de *$scope*. Esta variable puede tomar los valores popular o normal, cuando esta toma el valor popular la expresión se evaluará al valor true, todo el contenido que hereda de esa etiqueta se mostrará por pantalla, mientras que en caso contrario esa porción de código HTML quedaría oculta en la vista.

Una directiva similar a *ng-show*, que también hemos usado es *ng-if*, donde se evalúa la expresión asociada a los valores true o false, como si de una sentencia if (sentencia con la que cuentan numerosos lenguajes de programación para evaluar condiciones lógicas) se tratase y en función del resultado se muestra el

contenido del código HTML que se encuentra bajo la influencia de la directiva *ng-if*.

En el Listado 5.10, podemos observar también la aparición de otra directiva llamada *ng-click*, esta directiva se encarga de controlar la acción de clicado sobre objetos por parte del usuario, de tal forma que cuando detecta ese evento, se lanza la ejecución del método o función que le hayamos asignado. En este caso se pide al controlador que la aplicación muestre la vista de *movie.html* con la información de la película que se pasa por argumento.

La última directiva que podemos ver en el Listado 5.10 es *ng-src*, su función es la misma que el atributo src presente en HTML, con la salvedad de que ng-src permite incluir variables de Angular procedentes del ámbito de *$scope*. Del mismo modo existe una directiva llamada *ng-href*, que cumpliría con el mismo propósito en sustitución de href, atributo perteneciente a HTML.

Por último comentar la directiva *ng-style* que permite establecer directamente a los elementos HTML reglas de estilo CSS. Como podemos ver en el Listado 5.11, al igual que funciona la etiqueta style, en los elementos HTML, la directiva *ng-style* permite la asignación de reglas de estilo a los elementos en los que se incluye.

Listing 5.11: Código parcial de *movies.html*

```
<div class="col-md-4 col-md-offset-1" ng-style="{'padding-bottom':'40px'}">
    <img class="img-responsive" ng-src="{{item.images.screenshot.thumb}}" id="
        ↪photoBorder">
</div>
```

También nos encontramos con la necesidad de crear dos servicios que nos permitieran llevar a cabo ciertas acciones necesarias para el desarrollo. En primer lugar nos encontramos con ShareFactory (presente en el Listado 5.12), que es un servicio que creamos para enviar datos de un controlador a otro, ya que en determinadas vistas realizamos una consulta y esperamos a su resolución para acceder a la siguiente interfaz con todos los datos cargados. Para ello entra en juego ShareFactory que nos permite almacenar hasta dos objetos a los que podemos acceder desde otro controlador en el que se haya inyectado como de-

pendencia este servicio.

El segundo de los servicios creados recibe el nombre de SearchFactory, cuyo papel es parecido al de ShareFactory con la salvedad de que este recopila la información necesaria para realizar una consulta, desde el formulario de la barra de navegación y la prepara de manera adecuada para que desde la vista search.html se realice la consulta hacia la API y se muestren los resultados hacia el usuario.

Para ilustrar la creación de un servicio en Angular podemos ver el Listado 5.12, donde se encuentra el código perteneciente al servicio ShareFactory.

Listing 5.12: Código del servicio *ShareFactory*

```
//############ Servicio para intercambiar información de una vista a otra ############
  app.factory("ShareFactory", function(){
    var myShareFactory = {};
    myShareFactory.shareItem = null;
        myShareFactory.auxShareItem = null;

    myShareFactory.setShareItem = function (value) {
      this.shareItem = value;
    };

    myShareFactory.setAuxShareItem = function (aux) {
        this.auxShareItem = aux;
    };

    return myShareFactory;
  });
```

Capítulo 6

Pruebas

En este capítulo nos centraremos en el apartado de pruebas, donde describiremos las herramientas empleadas para el diseño y desarrollo de las mismas, así como comentar las distintas pruebas que se han realizado sobre el código de la aplicación.

Como indica la documentación de AngularJS [2], JavaScript es un lenguaje de tipado dinámico, donde los objetos cambian constantemente sobre una misma variable junto con sus valores y tipos, lo que proporciona una gran expresividad, con la carencia de la falta de ayuda por parte del compilador. Por esta razón desde Angular piensan que el código escrito en JavaScript debe estar minuciosamente probado, para favorecer esto, proporciona una serie de herramientas que facilitan el desarrollo de pruebas sobre aplicaciones desarrolladas con Angular, para que no existan excusas por parte del programador a la hora de testear su sistema.

6.1. Karma y Jasmine

En esta sección hablaremos sobre las herramientas empleadas para el desarrollo y ejecución de las pruebas, entre las que se encuentran Karma, Jasmine y ngMock.

Karma es una aplicación desarrollada en NodeJS, que proporciona una herramienta de línea de comandos JavaScript. Su objetivo principal es generar

un servidor Web donde ejecutar el código de la aplicación y realizar las pruebas.

Karma cuenta con una serie de opciones de configuración, que nos permite por ejemplo ejecutar las pruebas sobre distintos buscadores, para asegurarnos de que nuestro sistemas funcionan correctamente en cualquier navegador Web. Nosotros hemos optado por realizar las pruebas sobre Chrome y Firefox, dos de los navegadores más extendidos en cuanto a uso por parte de los usuarios.

Para la ejecución de Karma se emplea la línea de comandos del sistema operativo, donde se muestran los distintos resultados obtenidos. En la siguiente sección se mostrarán las salidas obtenidas tras la ejecución de las pruebas realizadas sobre el sistema desarrollado.

Jasmine es un framework para pruebas sobre JavaScript. Resulta ser el más popular para realizar pruebas sobre aplicaciones desarrolladas con Angular, ya que proporciona una serie de funciones que ayudan a estructurar las pruebas y a realizar asertos para crear casos de prueba. Además Jasmine se puede combinar con el módulo de Angular ngMock.

ngMock es un módulo perteneciente a Angular que permite la inyección y simulación, en pruebas unitarias, de los distintos elementos existentes en Angular. Gracias a este módulo se facilita en gran medida la fase de pruebas de este tipo de aplicaciones, permitiendo testear las distintas partes del sistema.

Figura 6.1: Karma y Jasmine para pruebas unitarias sobre AngularJS

6.2. Pruebas unitarias

Las pruebas unitarias tienen como objetivo aislar distintas partes del código de una aplicación, para someterlas de manera independiente a una serie de pruebas. De esta manera se puede comprobar que los distintos módulos de una aplicación realizan la tarea que se espera que lleven a cabo. Por ello en esta sección explicaremos y mostraremos los distintos casos que se han configurado para testear nuestro sistema.

Durante el análisis de las pruebas, antes de su desarrollo y ejecución, se decidió que al contar con distintas partes del código donde se emplean los mismos servicios de Angular, o que en su lugar cuentan con un desarrollo similar en busca de un mismo objetivo, se realizarán pruebas en base al servicio común, de tal forma que se aprecie el correcto funcionamiento del elemento que interviene, simplificando y reduciendo el número de pruebas.

Por ello se determinó que las pruebas deben abarcar:

- Comprobar la correcta carga del módulo de la aplicación
- Comprobar la correcta carga de un controlador
    - Correcta creación de variables
- Comprobar el correcto funcionamiento del servicio *$http*, con petición de un objeto
    - Correcta petición
    - Objeto existente
    - Objeto recibido igual al esperado
- Comprobar el correcto funcionamiento del servicio *$http*, con petición de varios objetos
    - Correcta petición
    - Objetos existentes

    - Objetos recibidos igual a esperados
    - Orden de objetos recibidos igual a orden esperado

- Comprobar el correcto funcionamiento de un servicio personalizado
    - Correcta inyección
    - Correcta creación de variables
    - Correcta asignación de valores
    - Correcta petición de valores

Para ello se han estructurado las pruebas en dos archivos distintos, donde agrupamos las pruebas del controlador en *controllerTest.js* por un lado, mientras que por otro tenemos las pruebas sobre el servicio, en el archivo *serviceTest.js*.

Para agrupar un conjunto de pruebas y estructurar así los asertos que se aplican sobre un elemento común, se usa el método *describe(*{nombre}*, function () {* {cuerpo de pruebas} *}*, donde {nombre} sería la cadena que identifica a este conjunto, mientras que {cuerpo de pruebas} sería todas las sentencias que recoge este conjunto, con las distintos casos de prueba que se hayan incluido.

Para comprobar la correcta carga de un módulo solo debemos emplear el método *angular.mock.module(*{nombre módulo}*)* y si además queremos inyectar dependencias con algún servicio de angular o servicio que hayamos definido previamente, se puede emplear el método *angular.mock.inject(function (*{servicio}*) {* {cuerpo} *})*, un ejemplo de ambos se encuentra en el Listado 6.1. Como se puede apreciar ambos están incluidos en el método *beforeEach*, que indica que la carga e inyección se realizará antes de cada prueba unitaria.

Listing 6.1: Código parcial de *serviceTest.js*

```
//Definimos variables
var myService;

//Cargamos el módulo
beforeEach(angular.mock.module("myApp"));

//Inyectamos el servicio
```

```
beforeEach(angular.mock.inject(function (ShareFactory) {
    myService = ShareFactory;
}));
```

Al igual que ocurre en el Listado 6.1, además de inyectar servicios para dependencias, también es posible realizar la carga de los controladores con sus respectivas dependencias. Como podemos ver en el Listado 6.2, se realiza la carga de un controlador, así como la inyección de los servicios que utilizará el mismo.

Listing 6.2: Código parcial de *controllerTest.js*

```
//Cargamos el controlador e inyectamos dependencias
beforeEach(angular.mock.inject(function ($controller, $rootScope, $http) {
mockScope = $rootScope.$new();
    controller = $controller("defaultCtrl",{
       $scope: mockScope,
       $http: $http
   });
   backend.flush();
}));
```

Una vez se realiza la carga de modulos, controladores e inyección de servicios, se procede con la realización de las pruebas, para ello se definen asertos que conforman los distintos casos de uso. Para definir cada caso de prueba se utiliza el método *it(*nombre prueba*, function () {* {cuerpo prueba} *})* que ejecuta una función para realizar la prueba. Un ejemplo de casos de prueba se puede apreciar en el Listado 6.3.

Listing 6.3: Código parcial de *controllerTest.js*

```
it("Creación de variables", function () {
    expect(mockScope.pelicula).toBeDefined();
    expect(mockScope.peliculas).toBeDefined();
});

it("Realizar peticiones", function () {
    backend.verifyNoOutstandingExpectation();
});

it("Procesar la respuesta de peliculas", function () {
    expect(mockScope.peliculas).toBeDefined();
    expect(mockScope.peliculas.length).toEqual(4);
});

it("Comprobar orden elementos peliculas", function () {
  expect(mockScope.peliculas[0].movie.ids.trakt).toEqual(193079);
```

```
  expect(mockScope.peliculas[1].movie.ids.trakt).toEqual(176503);
  expect(mockScope.peliculas[2].movie.ids.trakt).toEqual(167397);
  expect(mockScope.peliculas[3].movie.ids.trakt).toEqual(129583);
});
```

Cabe destacar el empleo de la función *verifyNoOutstandingExpectation()*, proveniente del servicio *$httpBackend*, que se encarga de lanzar todas las peticiones HTTP que tengamos definidas para los casos de prueba, controlar su recepción y finalización correcta, de modo contrario nos mostrará una excepción en el caso de prueba correspondiente.

El servicio *$httpBackend* nos permite simular las distintas acciones HTTP que realiza nuestra aplicación con lo que sería, por ejemplo en nuestro caso, la API de Trakt.tv, de tal forma que podemos definir las URL a usar, junto con su respuesta asociada y el método HTTP que se espera recibir en las peticiones a cada URL.

Existen una gran cantidad de funciones pertenecientes a Jasmine para la comprobación de los resultados del test. Como vemos en el Listado 6.3, el método expect identifica el resultado de una prueba y a este se le pueden aplicar funciones de comprobación, entre las que están:

- **expect(x).toEqual(val)** x y val deben ser iguales (pero no quiere decir que sean el mismo objeto).
- **expect(x).toBe(obj)** x y obj deben ser el mismo objeto.
- **expect(x).toMatch(regexp)** x coincide con la expresión regular regexp
- **expect(x).toBeDefined()** x ha sido definido
- **expect(x).toBeUndefined()** x no ha sido definido
- **expect(x).toBeNull()** x es null
- **expect(x).toBeTruthy()** x es true o es evaluado a true
- **expect(x).toBeFalsy()** x es false o es evaluado a false

- **expect(x).toContain(y)** la cadena x contiene a la cadena y
- **expect(x).toBeGreaterThan(y)** x es mayor que y

Además de esto, a cada una de estas funciones se puede aplicar not para obtener el inverso de cada uno de ellas, un ejemplo sería *expect(x).not.toBeNull()*, donde se espera que x no sea null.

Una vez se definen todos los casos de prueba podemos ejecutar karma, para conocer los resultados en cada uno de los navegadores y saber si estos se completan satisfactoriamente con lo esperado. Para ello se ejecuta el comando *karma start karma.config.js* desde la línea de comandos, como se ve en la Figura 6.2.

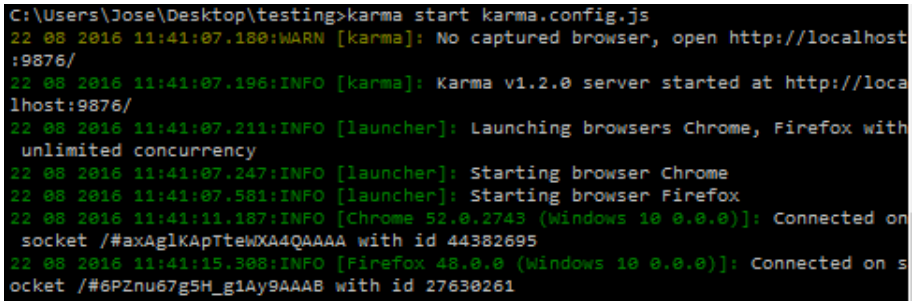


Figura 6.2: Inicializando pruebas con Karma y Jasmine

Ambos navegadores se inician, junto con karma para ejecutar las pruebas, comprobar los resultados e indicar por consola si se superan o no los distintos casos definidos. Como podemos ver en la Figura 6.3, todos los casos se superaron satisfactoriamente en cada uno de los navegadores.

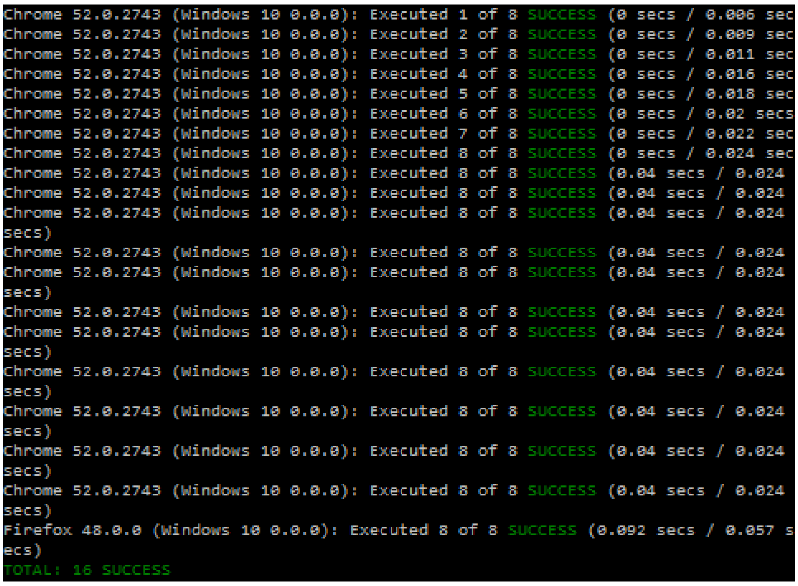


Figura 6.3: Resultados de las pruebas

Conclusiones

El desarrollo del Trabajo Fin de Grado ha resultado una experiencia gratificante, el hecho de partir desde cero, con una tecnología que nunca había usado y con el reto de aprender sobre esta, y llevar a cabo la creación de una aplicación Web con los conocimientos adquiridos.

Hemos logrado crear un Sistema de Información para películas y series con AngularJS que cuenta con información proveniente de la API de Trakt.tv, donde podemos consultar una gran cantidad de datos sobre los elementos en los que estemos interesados dentro de esta temática.

Siempre existió la opción de hacer algo similar a lo ya visto en la carrera, permitiendo la comodidad de usar elementos que ya se conocen y que de alguna manera amenizan el desarrollo del proyecto y permiten dedicar más tiempo al desarrollo, para la implementación de más aspectos del sistema. Pero decidimos dar un paso más y aprovechar todas las horas que iban a ser necesarias para la consecución del trabajo y aprovechar la oportunidad de aprender algo distinto, algo nuevo y además realizar algo con ello.

En un principio el comienzo fue complicado, la incertidumbre de si se lograría alcanzar los objetivos propuestos, si se conseguiría llevar a cabo con los conocimientos que se adquirieron durante la fase de aprendizaje. Muchos errores, fallos de funcionamiento, inexperiencia, cosas que no salían como debían salir, pero poco a poco comenzó todo a marchar, la práctica, la lectura de documentación y de foros especializados, la mejora que se consiguió a la hora de identificar errores en consola y buscar soluciones para estos.

Profundizar en los aspectos que nos parecieron más claves para el sistema y

descubrir herramientas que nos ayudaban en gran medida en las distintas fases del proyecto resultó también bastante interesante.

Los objetivos planteados han sido logrados, partíamos de la base de que no podríamos hacer una aplicación totalmente funcional, con las ideas que teníamos en mente, debido al tiempo y requisitos con los que cuentan los Trabajos Fin de Grado. Por ello nos marcamos una serie de metas a cumplir, como se habla en los capítulos de cap.2 Planificación, cap.3 Análisis de Requisitos y cap.4 Diseño. A partir de ahí se comenzó el desarrollo, la hora de poner en práctica lo aprendido, paso importante, ya que desde el punto teórico las cosas siempre son más fáciles de lo que resultan y fue a base de práctica y práctica, cuando los conocimientos adquiridos se afianzaron mejor.

La práctica de conceptos e ideas aprendidos durante la universidad, como pueden ser las distintas fases de un proyecto software, la necesidad de mantener un formalismo a la hora de organizar y gestionar un proyecto. Fases que han resultado muy útiles, ya que la actividad dedicada a cada una de ellas ha sido vital para llevar a cabo una correcta realización de acuerdo con las fechas previstas y las metas establecidas.

Fases como análisis y diseño, pruebas, resultan fundamentales para cerciorarnos de que lo que se está haciendo, se hace en base a algo establecido, con la ventaja de contar con metas concretas y de poder evaluar lo que se ha realizado.

El contar con un análisis previo y con una serie de especificaciones de lo que se esperaba del sistema a implementar, es un aspecto clave, pues de esta forma se consigue enfocar mejor el planteamiento de la solución final, realizar una mejor elección de que elementos usar y cómo hacerlo.

El desarrollo Front-End de un sistema Web, ha resultado muy interesante, aunque aún queda mucho que andar, ya que nos hemos percatado que los conocimientos en CSS no vendrían mal ampliarlos, así como otro tipo de lenguajes y tecnologías.

Pero no queremos quedarnos ahí, el siguiente paso sería pensar en realizar un desarrollo Back-End para nuestro sistema sobre películas y series. Con el objetivo de hacer realidad todas las ideas que teníamos propuestas para la aplicación y que no pudimos llevar a cabo por las limitaciones del proyecto.

Las líneas futuras podrían discurrir por los caminos de llevar a cabo un desarrollo que incluya el control de usuarios, con accesos mediante distintos roles. Conocer y aprender lenguajes y tecnologías que nos ayuden a llevar el desarrollo Back-End a cabo, conocer sus distintas partes.

Conseguir que los usuarios puedan llevar un listado de las cosas que ven o que desean ver, controlar los capítulos vistos de una serie y avisar cuando se vayan a estrenar capitulos, series o películas nuevas.

Permitir la adición de contenido por parte de los usuarios, aunque habría que estudiar su viabilidad, en función de si aportaría realmente resultados positivos. Permitir que los usuarios puedan personalizar sus cuentas o buscar algún tipo de manera para que puedan personalizar o colaborar de alguna forma, intentando imitar un poco a las redes sociales, cuyo éxito en parte viene de ese poder que tienen los usuarios para poder hacer cosas por sí mismos dentro del propio sistema.

Añadir de alguna forma un sistema de recomendación de contenido en función de las visitas o búsquedas por parte del usuario.

Bibliografía

[1] http://docs.trakt.apiary.io/#.

[2] https://docs.angularjs.org/api?PHPSESSID=cae8e98e7ca559b4605d75c813b358ee.

[3] Andrew Grant. *Beginning AngularJS*. Apress, 2014.

[4] Adam Freeman. *Pro AngularJS*. Apress, 2014.

[5] Marijn Haverbeke. *Eloquent Javascript*. No Starch Press, 2014.

[6] David Sawyer McFarland. *CSS The Missing Manual*. O'Reilly Media, 2015.

[7] http://librosweb.es/libro/pro_git/.

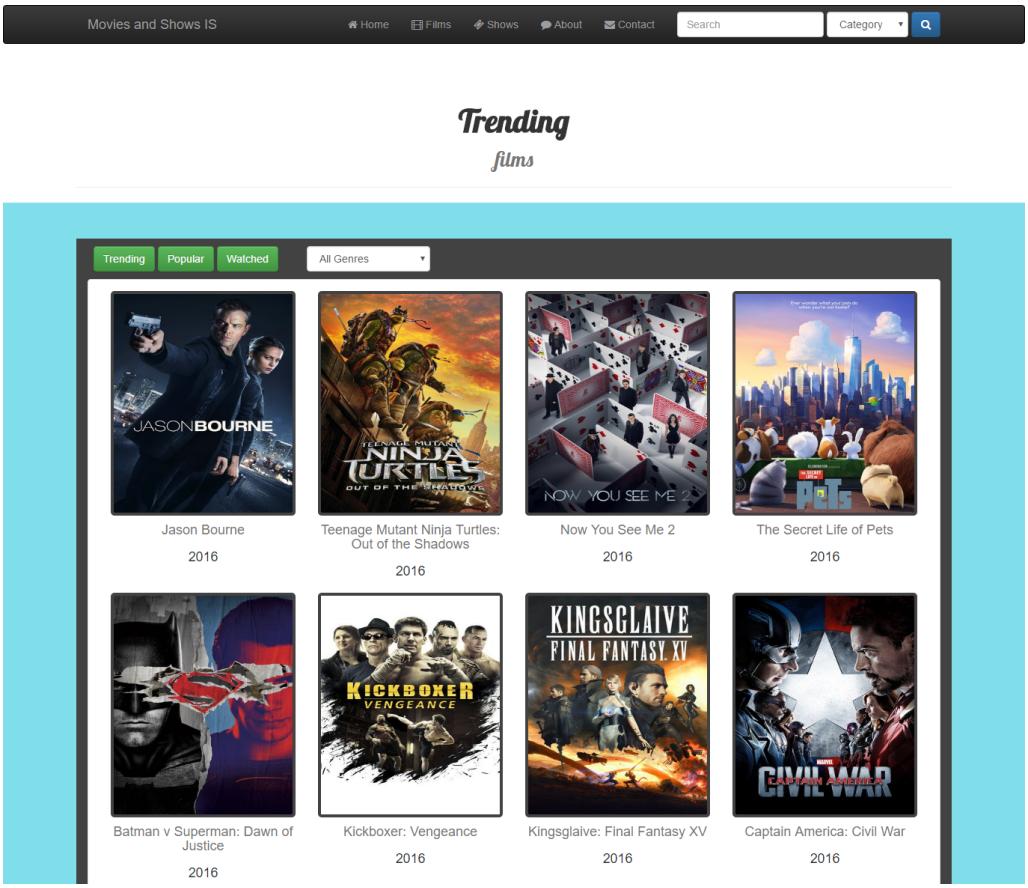
[8] http://getbootstrap.com/.

[9] http://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html.

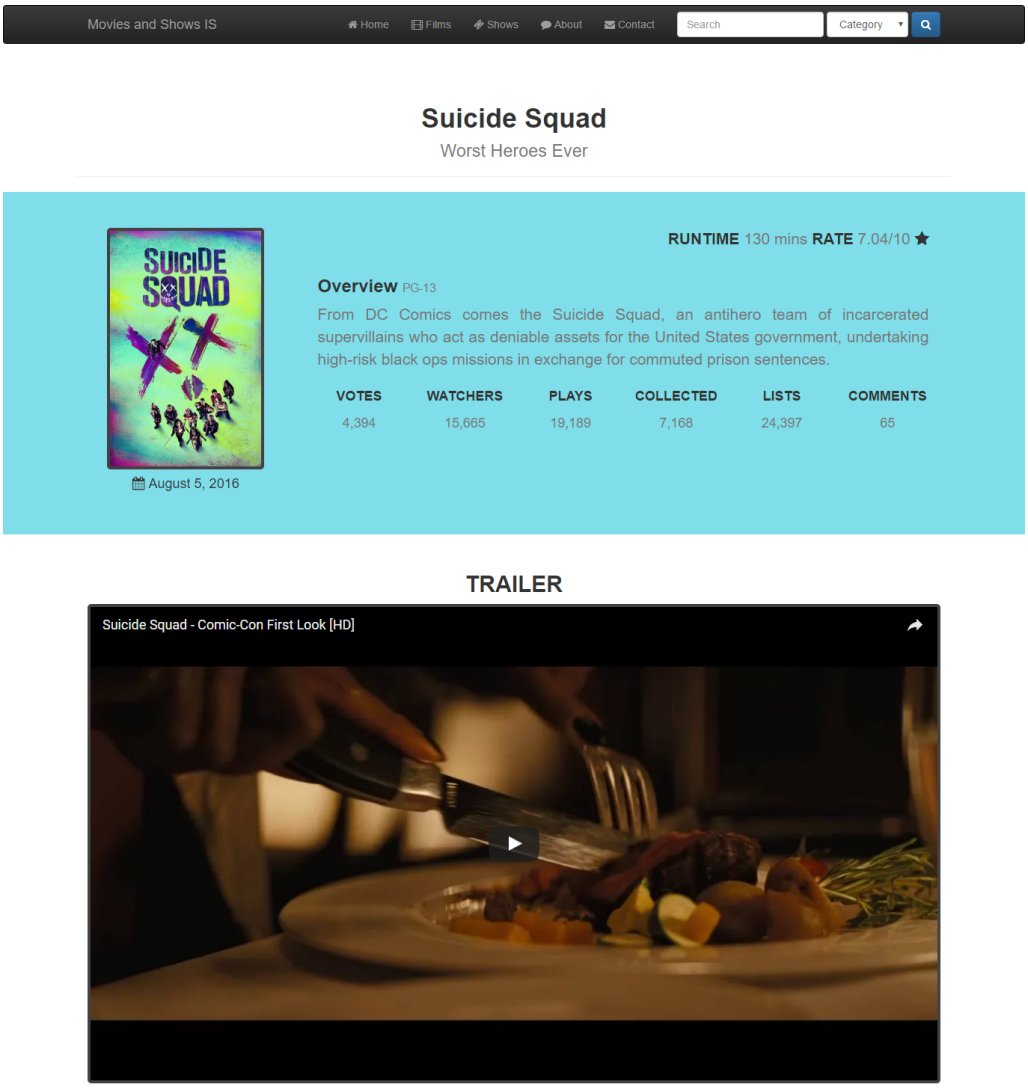
Apéndice A

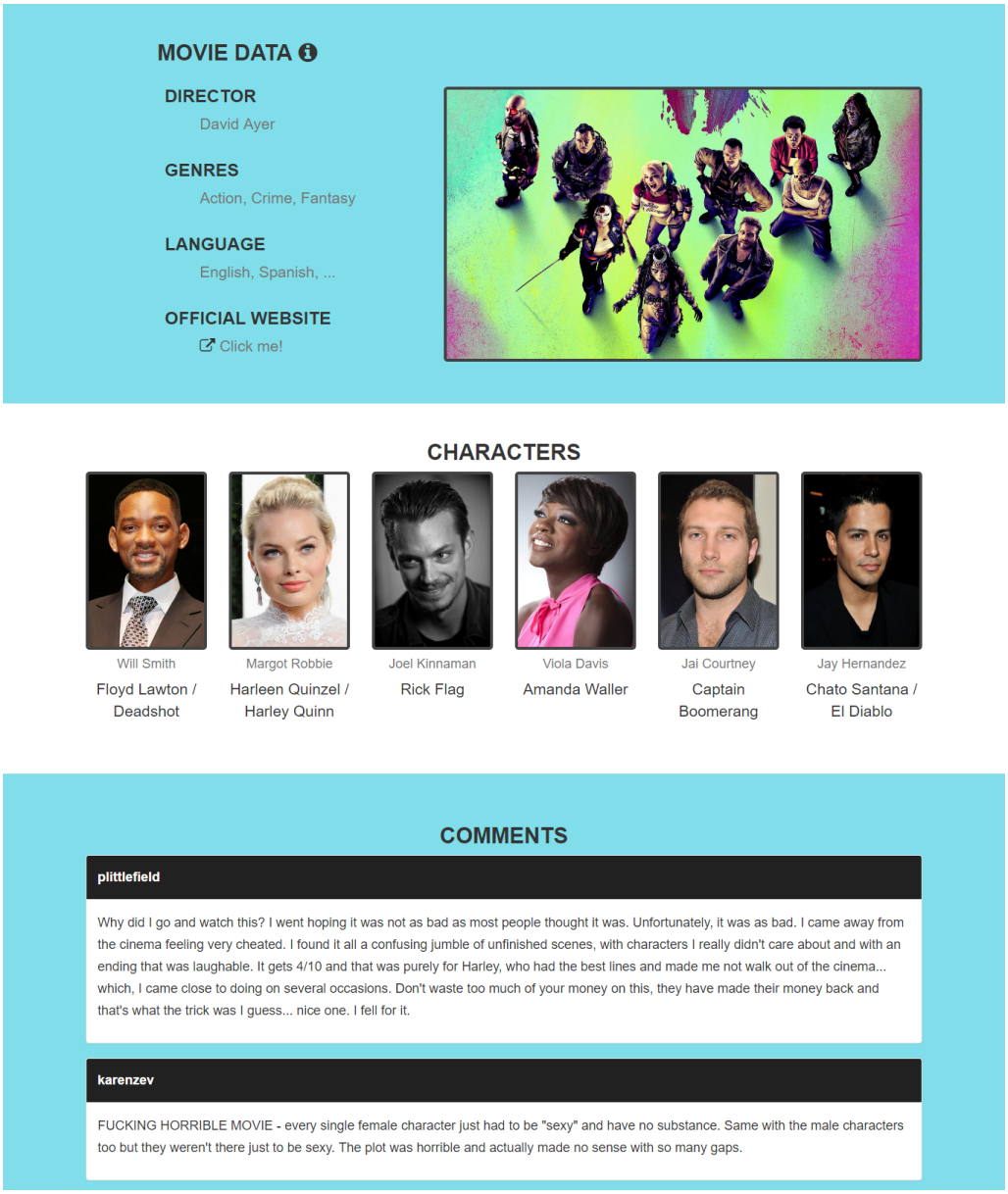
Anexos técnicos

A.1. Vista Movies.html

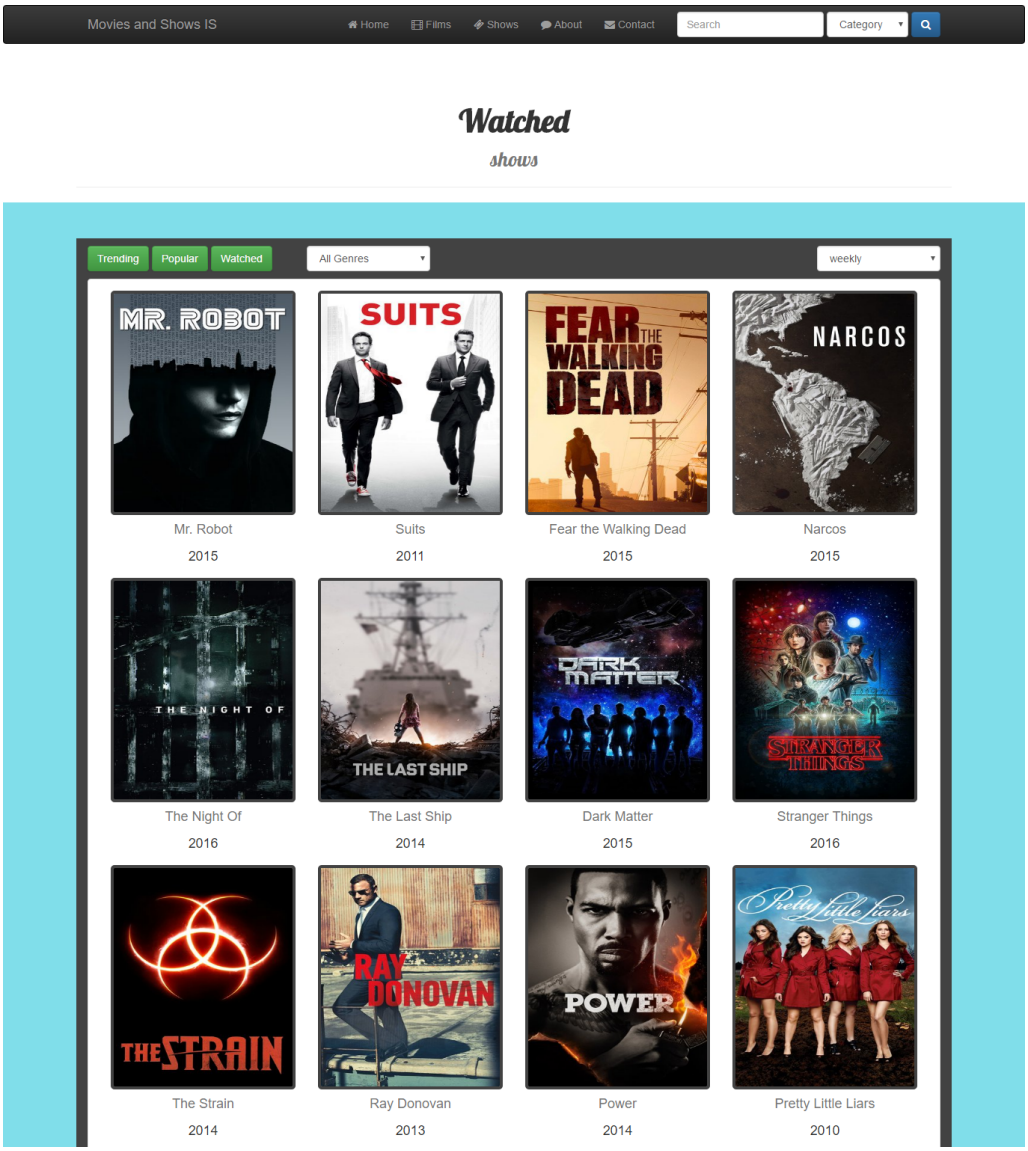


A.2. Vista Movie.html

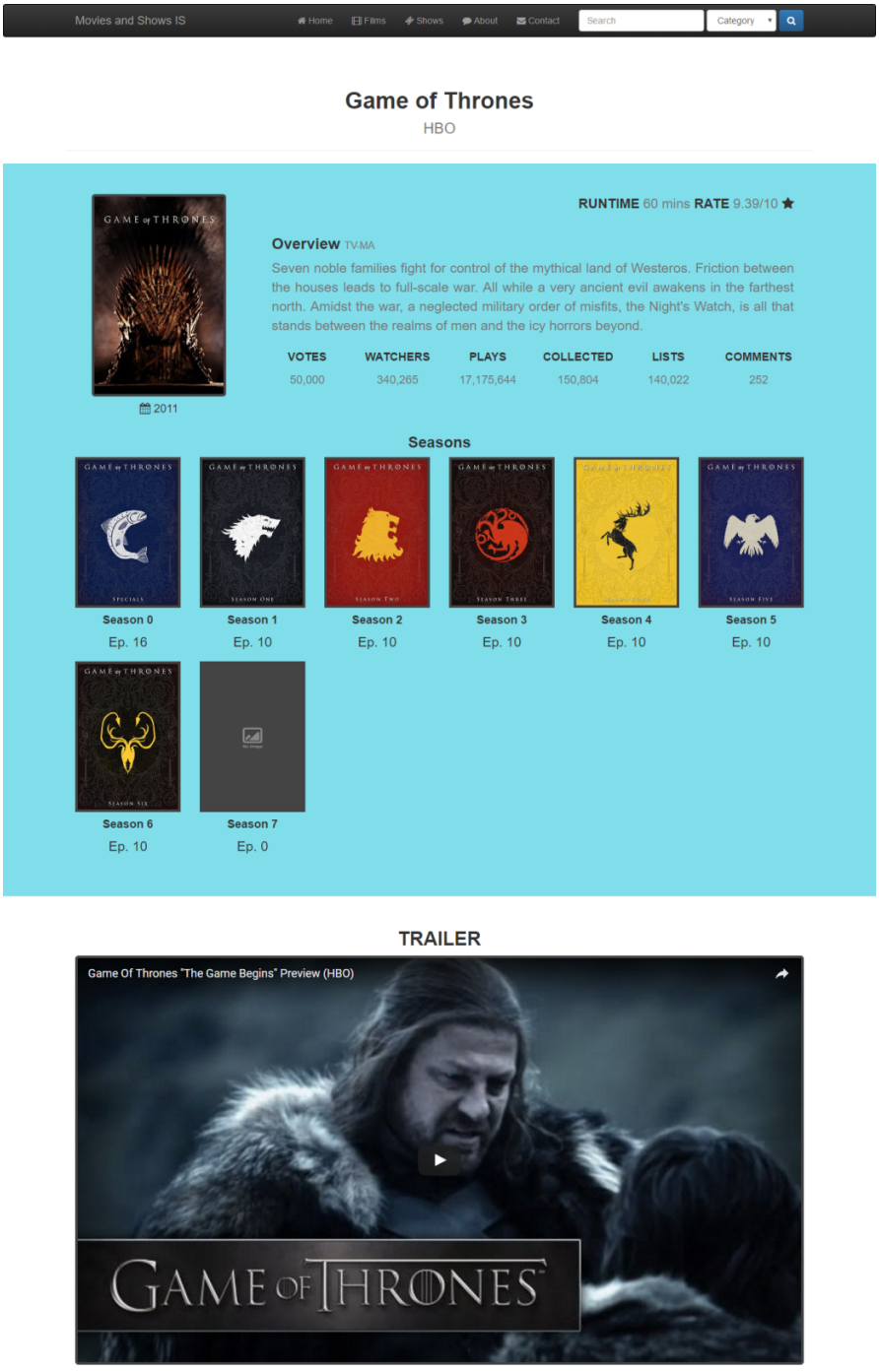



MOVIE DATA
DIRECTOR
David Ayer
GENRES
Action, Crime, Fantasy
LANGUAGE
English, Spanish, ...
OFFICIAL WEBSITE
Click me!
CHARACTERS
Will Smith
Floyd Lawton / Deadshot
Margot Robbie
Harleen Quinzel / Harley Quinn
Joel Kinnaman
Rick Flag
Viola Davis
Amanda Waller
Jai Courtney
Captain Boomerang
Jay Hernandez
Chato Santana / El Diablo
COMMENTS
plittlefield
Why did I go and watch this? I went hoping it was not as bad as most people thought it was. Unfortunately, it was as bad. I came away from the cinema feeling very cheated. I found it all a confusing jumble of unfinished scenes, with characters I really didn't care about and with an ending that was laughable. It gets 4/10 and that was purely for Harley, who had the best lines and made me not walk out of the cinema... which, I came close to doing on several occasions. Don't waste too much of your money on this, they have made their money back and that's what the trick was I guess... nice one. I fell for it.
karenzev
FUCKING HORRIBLE MOVIE - every single female character just had to be "sexy" and have no substance. Same with the male characters too but they weren't there just to be sexy. The plot was horrible and actually made no sense with so many gaps.

A.3. Vista Shows.html



A.4. Vista Show.html



SHOW DATA

FIRST AIRED

April 18, 2011

AIR DATE

Sunday 21:00
America/New_York

GENRES

Drama, Fantasy, Science-Fiction

OFFICIAL WEBSITE

Click me!

CHARACTERS

Kit Harington
Jon Snow

Emilia Clarke
Daenerys Targaryen

Peter Dinklage
Tyrion Lannister

Lena Headey
Cersei Lannister

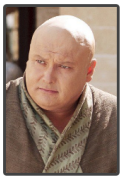
Nikolaj Coster-Waldau
Jaime Lannister

Sophie Turner
Sansa Stark

Maisie Williams
Arya Stark

Alfie Allen
Theon Greyjoy

Kristofer Hivju
Tormund Giantsbane



Conleth Hill
Lord Varys

Natalie Dormer
Margaery Tyrell

Liam Cunningham
Davos Seaworth

COMMENTS

frank18liang

I don't have any streams on shiftv #ShiftvW8

The Dragon Queen 1970

One of my favorite shows!

A.5. Vista Season.html

A.6. Vista Person.html