Universidad Autónoma de Madrid

Escuela Politécnica Superior

Trabajo de Fin de Máster

NEWS BACKEND: SERVIDOR DE AGREGACIÓN DE NOTICIAS

Máster en Ingeniería Informática

Autor: Bartosz Andrzej Zawada
Tutor: Roberto Latorre Camino

Septiembre 2014

Resumen

Resumen

Este Trabajo de Fin de Máster consiste en el diseño y desarrollo de un sistema que desempeñe la función de servidor para una aplicación móvil de noticias. El sistema debe obtener datos de fuentes RSS de terceros, procesarlos obteniendo información útil y, por último, ofrecer una interfaz para que una aplicación móvil pueda obtenerlos y entenderlos.

El sistema en sí consta de varios módulos, cada uno encargado de realizar una tarea diferente:

News Downloader
Módulo más importante y complejo.

- Periódicamente, descarga noticias de diversas fuentes RSS en paralelo.
- Normaliza los datos para evitar errores.
- Obtiene imágenes de alta resolución representativas de cada noticia.
- Calcula la resolución de dichas imágenes de forma eficiente.
- Inserta en una base de datos los datos procesados.
- Ofrece flexibilidad total a la hora de realizar la configuración.

News Database
Módulo intermediario, que guarda los datos procesados por el Downloader.

News Server
Módulo cuyo objetivo es ofrecer los datos mediante una API sencilla.

Aunque, tanto el backend como la aplicación móvil que hará de frontend con el usuario final pertenecen al mismo proyecto, es importante destacar que en esta memoria de Trabajo de Fin de Máster no se describe en detalle el frontend de la aplicación ya que ha sido desarrollado por otro equipo.

Palabras Clave

Agregador de Noticias, Backend, Servidor, Aplicación Móvil, RSS, Web Service, JSON, API, Node JS

Summary

Summary

This Masters' Thesis consists on the design and development of a backend system for a news mobile application. The system must obtain data from third party RSS feeds, process it, filter it to keep the useful information and, lastly, offer an interface for a mobile app to download and understand it.

The system is composed by three modules, each one dealing with a different task:

News Downloader
Most complex and important module.

- Periodically downloads news from multiple RSS feeds in parallel.
- Normalizes data to avoid errors.
- Obtains high resolution images to represent each article.
- Efficiently calculates the resolution of those images.
- Inserts the processed data into the database.
- Can be configured with absolute flexibility.

News Database
Intermediary module that stores the data processed by the Downloader.

News Server
Module whose purpose is to offer the data through a simple API.

Although, both the backend and frontend belong to the s ame project, it's important to highlight that this Master's Thesis does not include the development of the mobile application because it was done by another team.

Keywords

News Aggregator, Backend, Server, Mobile Application, RSS, Web Service, JSON, API, Node JS

Agradecimientos

En primer lugar, agradecer a mi tutor Roberto Latorre Camino por ayudarme y darme la oportunidad de realizar y presentar este trabajo.

Por otra parte, agradecer a mis padres que me han soportado y me han educado para llegar a ser quién soy. A mi gata, por no arañarme demasiado. A mi novia por estar ahí siempre apoyándome y, sobretodo, a mi hermano que me ha ayudado y me ha servido de ejemplo para no dejar nunca de aprender y mejorar.

Por último, un agradecimiento a toda la comunidad de software Open Source por crear las herramientas que he utilizado y sigo utilizando a diario para realizar mis desarrollos y proyectos.

Índice general

Índice de figuras IX

Glosario de acrónimos XI

1. Introducción **1**

1.1. Motivación del trabajo . . . 1

1.2. Objetivos y enfoque . . . 2

1.3. Metodología y plan de trabajo . . . 2

1.4. Contenido del documento . . . 3

2. Análisis **5**

2.1. Análisis de requisitos . . . 5

2.1.1. Requisitos funcionales . . . 5

2.1.2. Requisitos no funcionales . . . 7

2.2. Análisis de problemas . . . 8

3. Tecnologías Utilizadas **11**

3.1. Lenguaje de programación . . . 11

3.1.1. CoffeeScript es JavaScript . . . 11

3.1.2. ¿Por qué JavaScript? . . . 12

3.2. Formato de intercambio de datos . . . 14

3.3. Base de datos . . . 14

3.4. Librerías externas . . . 15

3.4.1. Cheerio . . . 16

3.4.2. Express . . . 16

3.4.3. Feedparser . . . 16

3.4.4. Iconv . . . 17

3.4.5. Imagesize-ex . . . 17

3.4.6. UnderscoreJS 17
3.4.7. Winston 17

4. Diseño **19**
4.1. Primera iteración 19
4.1.1. Problema de procesado de fuentes RSS 20
4.1.2. Problema de estabilidad 20
4.1.3. Necesidad de registros 21
4.1.4. Arquitectura Propuesta 22
4.2. Segunda iteración 22

5. Desarrollo **25**
5.1. Primera iteración 25
5.2. Segunda iteración 26
5.2.1. Downloader 26
5.2.2. Database 27
5.2.3. Server 29

6. Conclusiones **31**

7. Mejoras futuras **33**
7.1. Peticiones con timestamps 33
7.2. Utilización de caché 33
7.3. Posible Optimización de uso de memoria 34
7.4. Comentarios en noticias 34

Bibliografía y Referencias **35**

Índice de figuras

1.1. Desarrollo iterativo 3

2.1. Estructura de Newspaper Index 6

2.2. Estructura de Newspaper Show 7

4.1. Arquitectura base del sistema 19

4.2. Arquitectura del sistema (Primera iteración) 22

4.3. Arquitectura del sistema (Segunda iteración) 23

4.4. Diagrama UML de clases 24

5.1. Comando fork() en Node.js 25

5.2. Consulta normal realizada con *pg* 27

5.3. Consulta parametrizada realizada con *pg* 27

5.4. Definición de la tabla de periódicos 28

5.5. Definición de la tabla de secciones 28

5.6. Definición de la tabla de noticias 28

Glosario

- **ACID**: Atomicity, Consistency, Isolation, Durability. Conjunto de propiedades que aseguran que una transacción en la base de datos es segura.
- **Backend**: Componente del sistema que procesa los datos y es invisible a los usuarios.
- **Chrome V8**: Motor de JavaScript de código abierto creado por Google y utilizado en el navegador web Chrome.
- **Encoding**: Codificación de caracteres es el método que permite convertir un carácter de un lenguaje natural en un símbolo de otro sistema de representación, como un número o una secuencia de pulsos eléctricos en un sistema electrónico.
- **Escalabilidad horizontal**: Un sistema escala horizontalmente si al agregar más nodos al mismo, el rendimiento de éste mejora.
- **Framework**: Estructura conceptual y tecnológica de soporte definido, con módulos de software concretos, que puede servir de base para el desarrollo de software. Típicamente incluye una arquitectura predefinida, ciertas convenciones para que los desarrollos con dicho Framework mantengan una homogeniedad y unas herramientas para facilitar el desarrollo.
- **Javascript**: Lenguaje de programación interpretado. Utilizado principalmente como lenguaje de scripting en páginas web.
- **JOIN**: Cláusula SQL que sirve para combinar los registros de múltiples tablas de la base de datos.
- **JSON**: JavaScript Object Notation. Formato de datos ligero.
- **Middleware**: Función que intercepta peticiones, puede ejecutar cualquier código y decide si la petición continua o si bloquearla. Generalmente, se forma una cadena de Middlewares que se ejecutan en serie. Una petición sólo es respondida cuando todos los Middlewares dan su visto bueno. Un ejemplo de Middleware es una función que comprueba si el usuario que realiza la petición está autenticado y le deja continuar o le bloquea mostrando un error.
- **Nginx**: Servidor Web/proxy inverso ligero y de muy alto rendimiento.

- **Node.js**: Entorno de programación en la capa de servidor basado en Javascript ejecutado en el motor Chrome V8 (fuera del navegador).
- **NoSQL**: Amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS) en aspectos importantes, el más destacado que no usan SQL como el principal lenguaje de consultas. Los datos almacenados no requieren estructuras fijas como tablas, normalmente no soportan operaciones JOIN, ni garantizan completamente ACID, y habitualmente escalan bien horizontalmente.
- **ORM**: Object-Relational Mapping. Técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y una base de datos relacional. Esto crea una base de datos de objetos virtual que se puede utilizar desde dentro del lenguaje de programación
- **Overhead**: Consumo de recursos extra, necesario para la realización de una tarea. En formatos de intercambio y representación de datos se trata de la cantidad de caracteres extra necesarios para cumplir con el sintaxis del formato y que no da ninguna información útil.
- **Parsing**: Análisis de un texto para obtener sus componentes lógicas. Por ejemplo, convertir el contenido de un fichero XML o JSON a una representación que permita a la aplicación operar con ellos.
- **Routing**: Mecanismo que sirve para redirigir las peticiones a sus controladores y acciones correspondientes. Se basa en el parsing de URLs.
- **Script**: Programa usualmente simple, casi siempre interpretado. Su uso habitual es realizar tareas sencillas que interactúen con el sistema operativo y/o el usuario. Es frecuente que las shells sean los intérpretes muchas veces.
- **Shell**: Programa que provee una interfaz de usuario para acceder a los servicios del sistema operativo.
- **Syntactic Sugar**: Añadidos a la sintaxis de un lenguaje de programación que no afectan a la funcionalidad, pero que facilitan expresar algunas construcciones de una forma más clara o concisa, o en un estilo alternativo.
- **RSS**: Really Simple Syndication. Formato XML para compartir contenido en la web. Se utiliza para difundir información actualizada frecuentemente a usuarios suscritos a la fuente de contenidos.
- **URL**: Localizador de recursos uniforme.
- **XML**: Extensible Markup Language. Es un lenguaje de marcas utilizado para almacenar datos en forma legible.

1
Introducción

1.1. Motivación del trabajo

Este Trabajo fin de Máster constituye uno de los módulos de un proyecto software desarrollado en el ámbito empresarial. Dicho proyecto se inició a principios de 2013 por un equipo de tres desarrolladores que en su tiempo libre trabajaban en él.

El propósito de la aplicación era permitir a los usuarios leer noticias de múltiples medios informativos, con una interfaz intuitiva, rápida y atractiva; siendo el objetivo una experiencia de uso mejor que las demás aplicaciones que sirvieran para dicha finalidad.

El proyecto requería del desarrollo de una componente de servidor cuyo objetivo fuese obtener las noticias de diversas fuentes RSS y generar ficheros JSON con los datos procesados para que la aplicación móvil los pudiera descargar y presentar. Esto permitía a la aplicación móvil reducir la cantidad de procesamiento necesario y, por tanto, su consumo de batería.

Al ser un proyecto que se realizaba en horas libres el progreso era lento. A principios de 2014, la aplicación móvil estaba prácticamente lista. En cambio, el sistema del servidor aún se encontraba lejos de estar terminado, era frágil y a veces fallaba de forma crítica.

Por ello, en marzo de 2014 me propusieron hacerme cargo de redefinir y desarrollar el módulo de backend de la aplicación. Esto resultaba una gran experiencia para mí ya que además de darme la oportunidad para aprender y utilizar nuevas tecnologías sobre las que había leído pero que aún no había utilizado, me permitía desarrollar un trabajo de responsabilidad en un proyecto real.

1.2. Objetivos y enfoque

El objetivo es diseñar y desarrollar una componente de servidor que pueda dar servicio a una aplicación móvil de noticias. Sus capacidades deben ser las siguientes.

- Conectarse y descargar las noticias de múltiples periódicos. Donde cada periódico tiene múltiples secciones (cada una siendo una fuente RSS) y cada sección tiene sus propias noticias.
- Procesar los datos descargados para obtener la información que la aplicación móvil utiliza.
- Para cada noticia, encontrar una imagen que la represente y obtener sus dimensiones, para que la aplicación móvil pueda estructurar correctamente su contenido.
- Ofrecer los datos procesados para que la aplicación móvil pueda obtenerlos y presentarlos a los usuarios.

El enfoque por el que se ha optado para el desarrollo de este sistema es uno centrado en flexibilidad, estabilidad, mantenibilidad y velocidad de desarrollo, poniendo en segundo lugar el rendimiento puro. Esto se puede comprobar observando las herramientas y tecnologías utilizadas para la realización del proyecto (ver capítulo 3).

Se valora ante todo el principio DRY (Don't Repeat Yourself). Este principio dicta que toda pieza de conocimiento debe tener una única, precisa y autoritaria representación dentro de un sistema. Cuando este principio se cumple, una modificación de cualquier elemento no requiere cambios en otros elementos no relacionados lógicamente, mientras que, elementos que sí están relacionados cambian de forma predecible y uniforme, manteniéndose sincronizados. En otras palabras, la información nunca debería estar duplicada debido a que eso incrementa la dificultad en los cambios y puede introducir inconsistencias.

1.3. Metodología y plan de trabajo

La realización del trabajo ha consistido en una primera parte de análisis de requisitos (los cuales son explicados en el capítulo 2) donde se han tenido en cuenta los distintos aspectos y necesidades a cumplir para después efectuar un diseño consecuente a ello.

El desarrollo se ha producido de una forma iterativa (figura 1.1). Empezando por una primera versión que pueda dar servicio a la aplicación móvil de forma básica. Luego se ha ido expandiendo para añadir funcionalidades y eliminar los problemas que tuviera la anterior solución. Se han ido añadiendo nuevos requisitos cuando surgían nuevas necesidades y problemas en el proyecto.

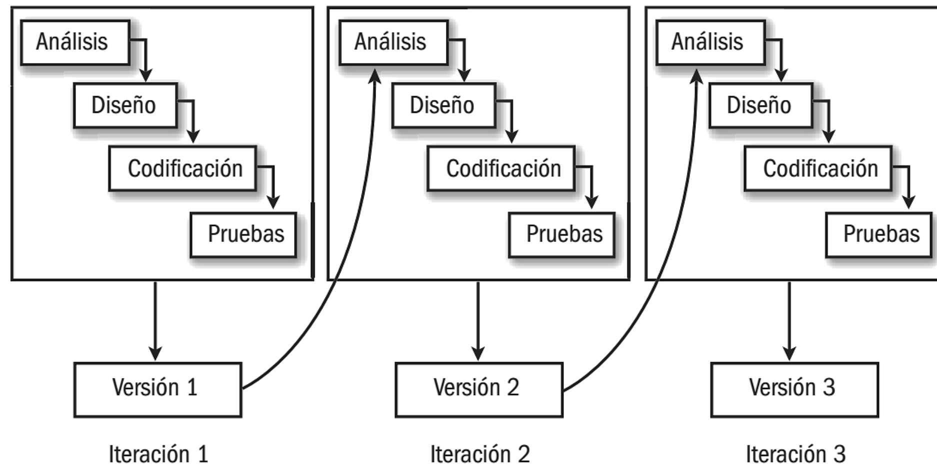


Figura 1.1: Desarrollo iterativo

1.4. Contenido del documento

Esta memoria de Trabajo fin de Máster ofrece una introducción y motivación del proyecto realizado (sección 1.1), establece unos objetivos claros y explica cuál ha sido el enfoque para cumplir dichos objetivos (sección 1.2). Por último, describe la metodología utilizada (sección 1.3).

En los apartados intermedios se exponen las tecnologías utilizadas (capítulo 3), el diseño (capítulo 4) y desarrollo (capítulo 5) del sistema con los distintos cambios, mejoras y correcciones realizados durante el desarrollo del proyecto.

Al final del documento se explican las conclusiones obtenidas del desarrollo del Trabajo fin de Máster (capítulo 6).

Finalmente, hay un último capítulo que expone mejoras futuras al sistema (capítulo 7).

2

Análisis

2.1. Análisis de requisitos

A continuación se explican tanto los requisitos funcionales como los no funcionales referentes al sistema.

2.1.1. Requisitos funcionales

El sistema debe:

- Trabajar con un número ilimitado de periódicos
- Admitir un número ilimitado de secciones por cada periódico.
- Descargar fuentes RSS[1] (cada sección del periódico es una fuente RSS).
- Procesar los datos de las noticias recibidas de las distintas fuentes RSS almacenando la siguiente información:
  - Identificador único
  - URL de la noticia
  - Título
  - Texto
  - Autor
  - Fecha de última actualización
  - URL de imagen

  - Ancho de la imagen
  - Alto de la imagen

- Obtener una imagen de portada para cada periódico. Esta imagen ha de obtenerse de la primera sección de dicho periódico y debe corresponderse con la primera noticia que tenga una imagen.

- Ofrecer los datos procesados en formato JSON (se explica la razón en la sección 3.2) estructurados de formas específicas que la aplicación móvil pueda entender. Las dos estructuras diferentes utilizadas por el sistema serán denominadas **Newspaper Index** y **Newspaper Show** en esta memoria para simplificar la explicación. Las estructuras exactas vienen impuestas por la aplicación móvil.

  Newspaper Index Su propósito es dar información global a la aplicación sobre todos los periódicos que están disponibles. La información debe ser suficiente para que la aplicación pueda exponer los periódicos en una lista y mostrar sus nombres, logos, la noticia en portada y una URL para poder acceder a sus noticias. Estructura descrita en figura 2.1.

```
1  {
2      "updatedAt": Timestamp, # nanoseconds
3      "countries": [
4          { # Pais
5              "country": String, # ISO 3166-2 code,
6              "newspapers": [
7                  { # Periodico
8                      "title": String,
9                      "id": String,
10                     "titleColor": String, # 6 char hex code
11                     "imageUrl": String, # Logo del periodico
12                     "jsonUrl": String, # URL de Newspaper Show
13                     "coverNews": { # Noticia en portada
14                         "id": String,
15                         "title": String,
16                         "imageUrl": String,
17                         "imageWeight": Integer,
18                         "imageHeight": Integer
19                     }
20                 }
21                 ...
22             ]
23         }
24         ...
25     ]
26 }
```

Figura 2.1: Estructura de Newspaper Index

  Newspaper Show Su objetivo es dar toda la información correspondiente a un periódico. Todas sus secciones y noticias. Estructura descrita en figura 2.2.

```
1  {
2      "id": String,
3      "title": String,
4      "jsonUrl": String,
5      "imageUrl": String,
6      "titleColor": String, # 6 char hex code
7      "originalUrlLight": Boolean,
8      "sections": [
9          { # Seccion
10             "title": String,
11             "url": String, # RSS feed
12             "updatedAt": Timestamp,
13             "news": [
14                 { # Noticia
15                     "id": String,
16                     "originalUrl": String,
17                     "title": String,
18                     "text": String,
19                     "date": Timestamp,
20                     "imageUrl": String,
21                     "imageWidth": Integer,
22                     "imageHeight": Integer
23           }
24           ...
25             ]
26         }
27         ...
28     ]
29 }
```

Figura 2.2: Estructura de Newspaper Show

- **(Primera iteración)** Guardar los datos procesados en el disco en ficheros con formato JSON[2]. Permitiendo que un servidor web como Nginx[3] o Apache[4] los sirva.
- **(Segunda iteración)** Utilizar una base de datos para almacenar la información y ofrecerla mediante una API.

2.1.2. Requisitos no funcionales

El requisito principal es que el sistema sea fácilmente mantenible. Para ello hay que reducir la cantidad de líneas de código al mínimo y reutilizar el código para tareas semejantes.

El sistema ha de ser extremadamente estable y no dejar de ejecutarse en ningún caso (a menos que fuese manualmente apagado).

En caso de ocurrir un error en una noticia (porque esté mal formada o le falten datos principales) el sistema debe descartar dicha noticia sin afectar al resto del periódico. Si esto no fuera posible el error no debe en ningún caso afectar a otros

periódicos y, si fuera posible, tampoco a otras secciones del mismo periódico.

El sistema debe volver a un funcionamiento normal por sí mismo en caso de error o interrupción del servicio.

(Segunda iteración) El rendimiento resulta importante en el servidor web. Se debe reducir su carga de trabajo lo máximo posible, guardando en la base de datos la información ya procesada.

2.2. Análisis de problemas

En esta sección se van a explicar los distintos problemas que se identificaron durante la fase de análisis del sistema. Como parte de este análisis, se evaluó la versión del módulo de backend desarrollada previamente.

Durante la evaluación se encontró que abundaba la repetición de código; el mantenimiento resultaba difícil y costoso. La mitad de periódicos tenían problemas de *encoding* (las fuentes pueden utilizan UTF8, ISO 8859-1 u otras variantes de codificación de caracteres) o, directamente, los datos no llegaban a obtenerse. Además, eran alrededor de 20 *scripts* distintos, uno por cada periódico. Cada *script* hacía falta ejecutarlo y pararlo por separado, lo que complicaba controlar su ejecución. Por otra parte, no se registraba ninguna información y, por tanto, no era fácil obtener las causas de errores, sólo se sabía que el *script* en cuestión se detenía. Por último, de vez en cuando, se corrompía el fichero de índice de periódicos JSON (Newspaper Index) debido a que los procesos intentaban escribir en ese fichero al mismo tiempo.

Por tanto, se podría decir que los principales problemas son:

- La fuente RSS de cada periódico es distinta. Cada una tiene sus propias peculiaridades en cuanto a las etiquetas XML que utilizan y sus formatos. Por tanto cada uno de los periódicos requiere un código ligeramente distinto para realizar el procesamiento.

- Si se estructura mal el sistema, es posible acabar con una copia del mismo código por cada periódico. El problema real aparece cuando hay que realizar alguna modificación y el código afectado se encuentra en todas esas copias a la vez. El tiempo necesario para el mantenimiento crece linealmente con el número de periódicos en este caso.

- Debido a las peculiaridades de cada fuente, la forma en las que se gestiona las noticias excepcionales (sin imagen, url inexistente, formato extraño) en cada fuente es diferente. En caso de que el sistema no esté preparado para soportar todo tipo de entrada, por muy mal formada que ésta sea, es posible que el proceso lance una excepción y muera. En el peor de los casos, una sola fuente puede tirar abajo todo el sistema impidiendo que se mantenga un procesamiento normal en el resto de fuentes.

- Se trabaja con una cantidad muy grande de datos obtenidos de terceros que pueden en cualquier momento cambiar su formato. Es necesario estar al tanto cuando eso ocurra o se corre el riesgo de que los usuarios no puedan leer su periódico favorito durante demasiado tiempo y decidan pasarse a otra aplicación.

- En caso de optar por un diseño multiproceso, es posible entrar en complicaciones a la hora de controlar la ejecución.

- **(Primera iteración)** A la hora de escribir el fichero JSON global, si el sistema utiliza una solución multiproceso se pueden producir problemas a la hora de escribir el fichero por varios procesos a la vez resultando en funcionamiento anómalo o incluso pérdida de datos.

Tras la evaluación de esta versión del código se decidió diseñar y desarrollar un nuevo módulo backend en lugar de intentar aplicar parches al anterior hasta que funcionara correctamente.

Por otra parte, en la segunda iteración del diseño, cuando se quiere utilizar una base de datos, aparece el problema de como repartir las responsabilidades entre los tres módulos que surgen. Una decisión incorrecta podría afectar negativamente al rendimiento o peor, limitar futuros intentos de añadir funcionalidad al sistema.

3

Tecnologías Utilizadas

3.1. Lenguaje de programación

Para este desarrollo, he decidido utilizar CoffeeScript[8]. En este capítulo se explicará qué es y por qué razones se ha elegido.

3.1.1. CoffeeScript es JavaScript

CoffeeScript es un pequeño lenguaje que se compila a JavaScript.

Esto significa, que todo el código que se escriba en CoffeeScript se puede convertir a JavaScript con un sólo comando. CoffeeScript ofrece una síntaxis mucho más elegante; esconde muchas peculiaridades de JavaScript (por ejemplo los operadores '==' y '===' que funcionan de una forma inesperada para alguien acostumbrado a lenguajes de programación como C o Java), ofrece una sintaxis para definir clases más común (frente a la extraña sintaxis de la herencia de prototipos de JavaScript), etc. También hace que el código sea más conciso, elimina la necesidad de muchos paréntesis y transforma la sintaxis de las funciones (que en JavaScript se utilizan abundantemente) a una mucho más limpia.

En esencia, CoffeeScript añade *syntactic sugar* a JavaScript, y más tarde se compila a JavaScript normal y corriente. No resulta en pérdida de rendimiento. La única desventaja que tiene es que divide a la comunidad de desarrolladores entre los que lo usan y los que no. Aún así, al compilarse a JavaScript, cualquier librería que esté escrita en JavaScript o CoffeeScript, es usable por todos.

En conclusión, se puede considerar que el lenguaje utilizado es JavaScript para explicar sus ventajas y desventajas frente a otros lenguajes de programación.

3.1.2. ¿Por qué JavaScript?

JavaScript es un lenguaje dinámico muy extendido que normalmente es interpretado. Sin embargo, el motor de JavaScript implementado por Google (utilizado para este desarrollo) es capaz de compilar código justo antes de ejecutarlo y aplicar optimizaciones en tiempo de ejecución, aumentando su velocidad notablemente. Aún así su rendimiento es menor que C[9] o Java[10], aunque es mucho más rápido que otros lenguajes dinámicos como Python[11] o Ruby[12]. Su librería estándar es muy pobre, aunque eso se puede suplir con la enorme cantidad de librerías de terceros de las que dispone.

Por otra parte, ofrece arrays y hashes heterogéneos, muy útiles cuando se trabaja con grandes cantidades de datos que no siguen un formato rígido. En otros lenguajes de programación los tipos de datos heterogéneos conllevan un abundante uso del cast, que hace el código más complicado y (por ejemplo en Java) dificulta el uso de genéricos. JavaScript es extremadamente rápido trabajando con expresiones regulares. Cuenta con la ventaja de que JSON procede de JavaScript y por tanto son tecnologías que funcionan extremadamente bien juntas. Trata a las funciones como objetos de primera clase y, por último, JavaScript se presta muy bien a programación basada en eventos o, en otras palabras, a la programación asíncrona.

Antes de hacer comparaciones, hay que mencionar Node.js, la plataforma de desarrollo que usa JavaScript en servidores. Node.js se basa en utilizar el motor de JavaScript Chrome V8 (desarrollado por Google como software Open Source) para ejecutar JavaScript en el servidor. A esto se le han añadido librerías de funciones para las necesidades más básicas (interactuar con sistema operativo, sistema de ficheros, http, etc). Node.js ha sido implementado para aprovechar la programación asíncrona de Javascript; su punto fuerte es que no se bloquea cuando hace entrada-salida.

Ahora, veamos que ventajas y desventajas se encontraron, durante el análisis, en el uso de JavaScript frente a una alternativa de desarrollo más tradicional como JavaEE.

Ventajas

- JavaScript es muy rápido a la hora de escribir ya que con las librerías adecuadas se dispone de herramientas de muy alto nivel.
- En muchos desarrollos web, el rendimiento del lenguaje no es el principal cuello de botella, ya que la mayor parte del tiempo de respuesta se consume en tiempos de espera a la base de datos. Teniendo esto en cuenta, en una aplicación web de las características de la que se está desarrollando, en la que la componente transaccional es muy simple, Node.js es más eficiente porque no se bloquea.

  Suponiendo 10 peticiones que tienen que hacer un cálculo y obtener un dato de la base de datos, Node.js cogería la primera petición, realizaría el cálculo y mandaría una petición a la base de datos. Cogería otra petición y haría lo mismo. En ningún momento esperaría a la base de datos. La base de datos respondería cuando hubiera obtenido el dato y entonces Node.js ejecutaría un callback que respondería al cliente correspondiente. En esencia, Node.js ha servido los paquetes de forma asíncrona.

  Mientras que un servidor convencional, cogería la primera petición, realizaría el cálculo, mandaría una petición a la base de datos, esperaría a la base de datos, respondería al cliente y luego cogería otra petición, etc. Un servidor convencional respondería las peticiones, básicamente, en serie.
- JavaScript combina muy bien con JSON. Formato muy extendido debido a lo ligero que es.
- Node.js funciona sin necesitar casi configuración alguna. Mientras que JavaEE conlleva una carga de configuración muy grande. Esto permite en muy poco tiempo y código tener una aplicación funcionando.
- A la hora de crear aplicaciones web completas ayuda utilizar un único lenguaje tanto para backend como para frontend. Incluso es posible compartir código entre ellos (por ejemplo, verificación de modelos).

Desventajas

- JavaScript es más lento en cuanto a velocidad de cálculo puro.
- Node.js todavía está bastante inmaduro, no es extraño encontrar bugs en la librería estándar.

Por último, tengo cierta curiosidad respecto a Node.js porque está en alza. Hay cada vez más compañías que utilizan esta plataforma de desarrollo y puede resultar útil tener experiencia en ella.

3.2. Formato de intercambio de datos

Los formatos de intercambio de datos más habituales son: XML, JSON y YAML.

XML es una tecnología bastante simple y sólida. Conlleva un *overhead* bastante grande debido a todas las etiquetas. Su principal ventaja es que los tipos de datos quedan especificados porque se utilizan esquemas. Por otra parte, el *parsing* de XML es ligeramente más rápido que el de JSON. Esto se debe a que al admitir un esquema rígido no necesita buscar el tipo de dato de cada entrada. Sin embargo, esto no es una ventaja hasta que se manejan ficheros extremadamente grandes debido a que el *overhead* lo hace inicialmente más lento. Por último, los ficheros XML son más grandes y por tanto requieren un gasto de conexión mayor lo que es una desventaja bastante grande cuando tratas con móviles con tarifas de datos y conexiones inestables.

JSON es un formato de representación de datos extremadamente ligero. Originalmente proviene de JavaScript, por tanto combina muy bien con él. Su *overhead* es prácticamente nulo y su velocidad de *parsing* es buena. Esto se debe a que ofrece muy pocos tipos de datos: booleanos, strings, números, arrays y objetos (hashes).

YAML es un formato superconjunto de JSON. Es mucho más elegante y complejo, tiene funcionalidades más poderosas y resulta mucho más fácil de leer para humanos. Pero a cambio su velocidad de *parsing* es mucho más lenta que JSON. Por ello es una excelente opción para ficheros que sólo se van a leer una vez como, por ejemplo, configuraciones.

De entre los tres, JSON es el más obvio candidato, debido a que es muy fácil trabajar con él en JavaScript.

3.3. Base de datos

En cuanto a bases de datos. Node.js suele combinar muy bien con MongoDB. Que es una base de datos NoSQL[6] de documentos.

Mientras buscaba más información al respecto, me encontré referencias de otros desarrolladores que me hicieron dudar al respecto[7]. MongoDB es muy buena base de datos cuando tienes datos que no se relacionan casi en absoluto. En principio, los datos manejados por el backend cumplen esta condición. Sin embargo,

es bastante probable que en un futuro sea necesario añadir una funcionalidad que requiera relaciones entre datos. Por otra parte, MongoDB no dispone de transacciones y es posible que cuando ocurra un error al escribir en la base de datos se corrompan algunos datos de forma silenciosa (sin ningún indicio).

Teniendo en cuenta eso decidí usar una base de datos SQL relacional.

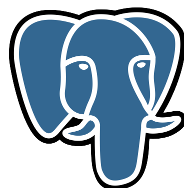
Las principales son:

Oracle La mejor base de datos, sin embargo, es cara y el soporte que ofrecen no es necesario para un proyecto de escala tan reducida.

MySQL La base de datos más popular en el desarrollo web. Probablemente por formar parte de packs como LAMP o WAMP. Por desgracia, ha sido comprada por Oracle y su desarrollo se ha estancado. Además, otro motivo para no utilizar MySQL es que en el pasado me dio problemas durante el desarrollo de otras aplicaciones web al realizar ciertas consultas avanzadas.

SQLite Una base de datos minimalista, utilizada principalmente en aplicaciones pequeñas sin multiusuario y que realizan escrituras poco frecuentas, y en tests. Últimamente, también se utiliza mucho en desarrollo de aplicaciones móviles. En esta aplicación se espera gran cantidad de escrituras, por tanto no es la mejor candidata.

PostgreSQL Base de datos avanzada, Open Source y totalmente gratuita. Cumple muy bien con estándares SQL y su rendimiento es bastante bueno. Como extra, incluye JSON como posible tipo de dato, permitiendo realizar operaciones con él.



En conclusión, decidí utilizar PostgreSQL como módulo Database.

3.4. Librerías externas

Se han utilizado múltiples librerías externas en el desarrollo de este sistema. Si existe una librería que realiza cierta tarea es preferible usarla antes que implementar el código a mano, porque estas librerías han sido utilizadas por miles de desarrolladores y sus errores están mayormente pulidos. En esta sección sólo se especifican las librerías más importantes.

3.4.1. Cheerio

Cheerio[13] es una librería de JavaScript cuyo objetivo es ofrecer las mismas funcionalidades que una de las librerías de JavaScript más conocidas: jQuery. La diferencia es que *Cheerio* incluye sus funcionalidades al lado del servidor. Al estar pensado para Node.js, se evita tener mucho código cuyo objetivo era normalizar las funcionalidades para las diferentes implementaciones de navegadores.

Durante el desarrollo ha sido utilizado, principalmente, para hacer pequeños cambios en los XMLs de las fuentes RSS justo antes de realizar el procesamiento, lo que sirve para normalizar ciertas peculiaridades de cada fuente y conseguir datos que las fuentes han colocado en etiquetas XML no estándar en RSS.

3.4.2. Express

Express[14] es un *framework* de desarrollo web para Node.js. Es bastante minimalista; incluye herramientas que permiten realizar funcionalidades básicas: ejecución del servidor, *routing* y permite escribir *middlewares* que funcionen a modo de filtros que se ejecuten justo antes de una petición y realicen alguna tarea. Por ejemplo, se encargan de la autorización permitiendo o bloqueando el acceso a recursos dependiendo de los privilegios de cada usuario, o realizan operaciones comunes a múltiples acciones distintas (si el usuario quiere ver o editar cierto recurso hay que buscarlo en la base de datos) evitando tener que repetir código.

La mayor parte de *frameworks* más complejos de desarrollo web en Node.js utilizan *Express* como base porque prepara los cimientos que toda aplicación necesita tener sin obligar a nada (no fuerza ninguna convención o librería sobre el desarrollador).

Por esto se ha utilizado para la implementación del News Server.

3.4.3. Feedparser

Feedparser[15] es otra librería de JavaScript. Esta librería es muy potente, su función es descargar y procesar fuentes RSS. Suele ser capaz de obtener la mayor parte de los datos como, por ejemplo, título de noticias, contenido y autor; Aunque eso depende de dónde los coloque cada fuente.

Esta librería me ha evitado tener que escribir código bastante complejo para trabajar con las fuentes RSS. Es una librería que funciona y lo hace bien. Además es Open Source, si hubiera algún problema se puede reportar en su repositorio[15].

3.4.4. Iconv

Iconv[16] es una librería cuyo objetivo es convertir texto entre diferentes *encodings*. Trabajar en múltiples *encodings* trae gran cantidad de problemas, por tanto, es necesario utilizar esta librería para realizar las transformaciones correspondientes normalizando todo a UTF-8.

3.4.5. Imagesize-ex

Imagesize-ex[17] es otra librería muy interesante que permite obtener los tamaños de imágenes GIF, PNG, JPG y BMP sin llegar a leerlas o descargarlas del todo. Es necesaria para poder implementar la obtención de tamaños de las imágenes representativas de las noticias de forma eficiente.

En caso de tener 25 periódicos, con 5 secciones por periódico y 40 noticias por sección. Esto supondría tener que establecer 5000 conexiones y descargar algo menos de 500 megabytes de imágenes (suponiendo que cada imagen ocupe sólo 100 kilobytes). Para evitar gastar más conexión en volver a descargarlas más tarde, se podrían guardar en el disco, pero eso sería un desperdicio muy grande de espacio y operaciones entrada-salida.

La solución que ofrece esta librería es mejor; sólo hay que establecer las conexiones, descargar los primeros kilobytes y desconectar.

3.4.6. UnderscoreJS

Esta es librería favorita; su propósito es agregar funcionalidades nuevas a JavaScript[18].

JavaScript tiene una librería estándar bastante limitada; deja mucho que desear comparada con la de otros lenguajes dinámicos como Ruby o Python.

Entre otras cosas, esta librería añade bastantes herramientas para hacer programación funcional. Personalmente, me gusta la programación funcional porque me parece una forma muy elegante de escribir código.

3.4.7. Winston

Winston[19] es una librería que ofrece una solución bastante completa para la necesidad de guardar registros de ejecución.

Es muy flexible; se puede especificar canales de salida muy diversos. Entre otros mostrar por consola, escribir a ficheros de texto, a bases de datos, mandar a sistemas de logging en la nube (p.e. Loggly[20]), enviar por correo.

También permite asignar un nivel a los mensajes (error, warn, info, etc) para que se puedan distinguir los mensajes importantes.

En conclusión, resulta una herramienta muy útil para guardar registros de todos los fallos y avisos generados por el backend.

4

Diseño

En este capítulo se explica el diseño escogido para el sistema y las razones por las que se ha elegido. El capítulo se dividirá en dos secciones, correspondientes a las dos iteraciones de desarrollo que se han realizado.

4.1. Primera iteración

Antes de nada, recordemos que diferencia ambas iteraciones: La primera iteración define un sistema que debe escribir los datos en el disco en ficheros JSON para que sean servidos por Nginx o Apache, mientras que, la segunda iteración debe ofrecer una API y utilizar una base de datos.

Un diagrama básico para describir el sistema desde fuera podría ser la figura 4.1

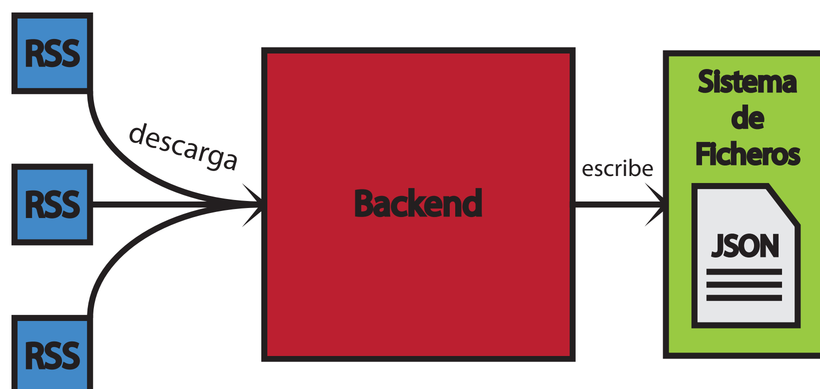


Figura 4.1: Arquitectura base del sistema

Profundicemos en el componente Backend. Su diseño debe solucionar los problemas expuestos en la sección 2.2.

4.1.1. Problema de procesado de fuentes RSS

Para evitar la duplicación de código entre periódicos se debe buscar una forma de realizar la descarga y el procesado de los datos que se pueda utilizar para todos los periódicos. Como se ha explicado anteriormente, cada fuente tiene sus propias peculiaridades. Así que una forma de conseguir este objetivo sería normalizar los datos antes de realizar el procesamiento. Por desgracia, no parece poderse hacer de una forma global, ya que arreglar el problema de una fuente podría causar otro problema en otra. Por tanto, la solución que hay que implementar debe permitir a cada fuente establecer una forma propia de normalizarse.

Una posible manera de estructurar este sistema sería tener un fichero de configuración que almacenara toda la información de los distintos periódicos y fuentes. A partir de los datos de este fichero se podrían instanciar objetos de la clase que represente el objeto encargado de realizar las descargas y el procesamiento (llamémosla Periódico). Por su parte, la clase Periódico debe llamar a unos *callbacks* cuando se vayan a producir ciertos eventos como empezar o terminar la descarga, el procesado, la búsqueda de imágenes, etc. El propósito de estos *callbacks* es normalizar datos.

Estos *callbacks*, que inicialmente no hacen nada (son funciones vacías), pueden redefinirse directamente en el fichero de configuración. Esto es posible gracias a que JavaScript trata a las funciones como objetos de primera clase (permite operar con ellas, guardarlas en variables, etc). Por tanto, es posible declarar una función anónima e insertarla en el objeto de clase Periódico correspondiente, para que cuando ocurra el evento, este *callback* sea llamado y ejecute el código que se haya inyectado.

Siguiendo este diseño, el único código que puede llegar a repetirse son los *callbacks* que, en un principio, se espera que no lleguen a repitirse entre distintos periódicos. En caso de que un callback se repita varias veces entre distintos periódicos, se puede declarar como método de la clase Periódico, y entonces utilizar un booleano en la configuración que sirva para indicar que se debe ejecutar dicho método. Con este diseño se soluciona el principal problema de la mantenibilidad: la repetición de código.

4.1.2. Problema de estabilidad

Por otra parte se encuentra el problema de la estabilidad. Debido a que Node.js es una tecnología ciertamente inmadura, durante el diseño se contempló la posibilidad de encontrarse con errores inesperados. Lo normal sería capturarlos y seguir con la ejecución normal. El problema llega cuando el error no es fácil de gestionar. En un caso así todo el proceso muere. Esto significa que un error en un Periódico causaría la interrupción del servicio de los demás periódicos.

Hay que minimizar los posibles daños en caso de error. Una solución interesante sería utilizar múltiples procesos. Esto además ofrece una ganancia de velocidad al aprovechar múltiples procesadores (Node.js es singlethreaded). Por desgracia, esto causa dos problemas: hay que poder controlar todos los procesos y eso puede resultar complejo, y es posible que se pueda corromper el fichero JSON de Newspaper Index. Esto se debe a que cada proceso se debe encargar de actualizar su porción del fichero.

El segundo problema es fácilmente solucionable (mientras se esté en el sistema operativo adecuado) mediante un *lock*[21] de escritura en el fichero. Impedir que más de un proceso pueda escribir a la vez evita que se corrompa el fichero y el tiempo durante el que un proceso bloquee a los demás es muy reducido y, por tanto, no supone ninguna pérdida de rendimiento.

La solución que se ha escogido es tener un proceso padre, cuyo objetivo sea crear un proceso hijo por cada periódico. Así los periódicos son independientes entre sí; si un proceso falla, puede morir sin llegar a afectar a los demás. Además, el proceso padre puede volver a crearlo para que empiece su ejecución desde cero. El problema del control desaparece, puesto que hay un único punto de entrada a la aplicación. Sólo hay que asegurarse de que al parar el proceso padre, todos los hijos se paren de una forma correcta para evitar procesos zombies. Como ya se ha dicho, el problema de la escritura paralela es fácilmente solucionable con un lock.

Otro enfoque para el problema de la corrupción del fichero es hacer que el proceso padre sea el único que escribe en ese fichero, y que los hijos se comuniquen con él mediante mensajes para indicar que cambios deben realizarse. Aunque resulta más complejo y no ofrece mayor ventaja frente al otro diseño.

4.1.3. Necesidad de registros

Es importante guardar registros de todo lo que ocurre durante la ejecución.

La forma de gestionar los errores es simple, cuando se capture una excepción se debe registrar todo lo que se pueda al respecto.

También es útil preparar avisos que nos informen sobre ocurrencias sospechosas durante la ejecución:

- El sistema debe avisar cuando una fuente RSS no se haya actualizado en mucho tiempo, porque significa que es el momento de investigar que le ha pasado y, posiblemente, eliminarla.

- Debe quedar registrado cuando una noticia no es procesada debidamente. Esto puede ayudar a detectar errores en la configuración de normalizado para la fuente de la que proviene la noticia.

- Sería interesante que el sistema enviara avisos de errores críticos por email o algún otro canal que permita llamar la atención de los administradores.

Resultaría útil que cada periódico guardara sus registros por separado, para que resulte más fácil buscar en ellos.

Por último, es importante guardar fecha y hora en los registros; puede ser importante saber que ha ocurrido justo antes de una interrupción de servicio.

4.1.4. Arquitectura Propuesta

Juntando todas las propuestas mencionadas en este capítulo obtenemos un diseño final que se puede representar como en la figura 4.2 y soluciona todos los problemas expuestos en la sección 2.2

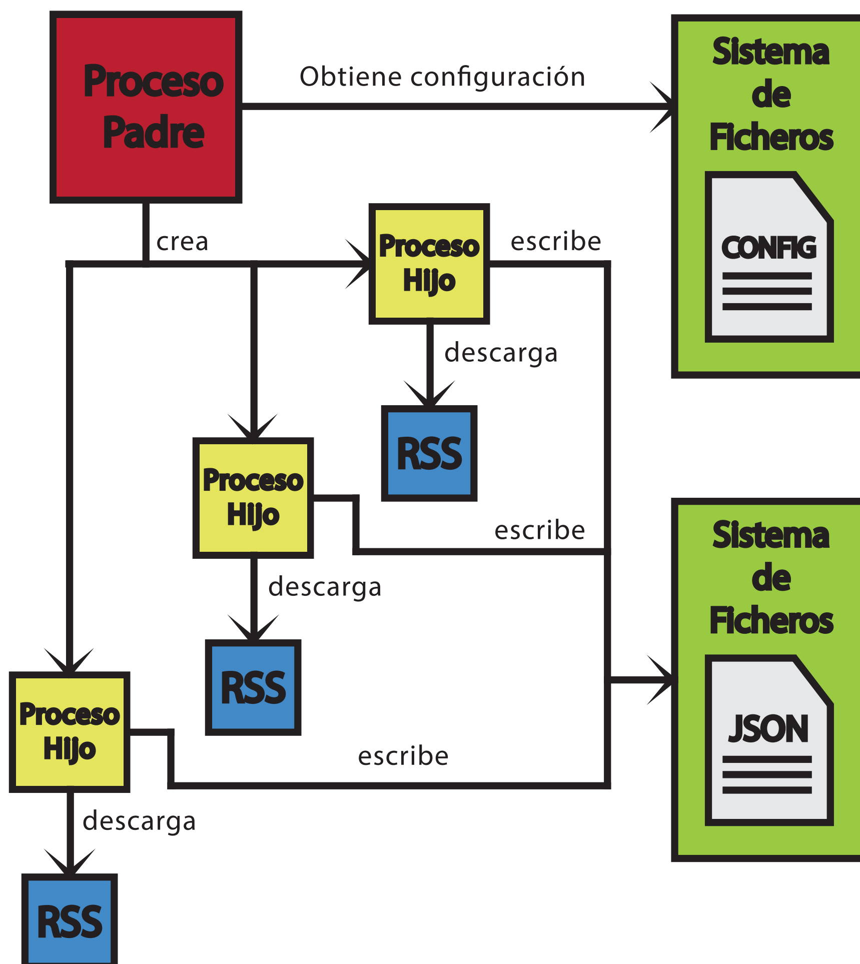


Figura 4.2: Arquitectura del sistema (Primera iteración)

4.2. Segunda iteración

La segunda iteración toma como base el diseño de la primera iteración e incluye dos módulos nuevos: una base de datos y un servidor que ofrece una API para acceder a los datos. Esto altera la arquitectura ligeramente (figura 4.3).

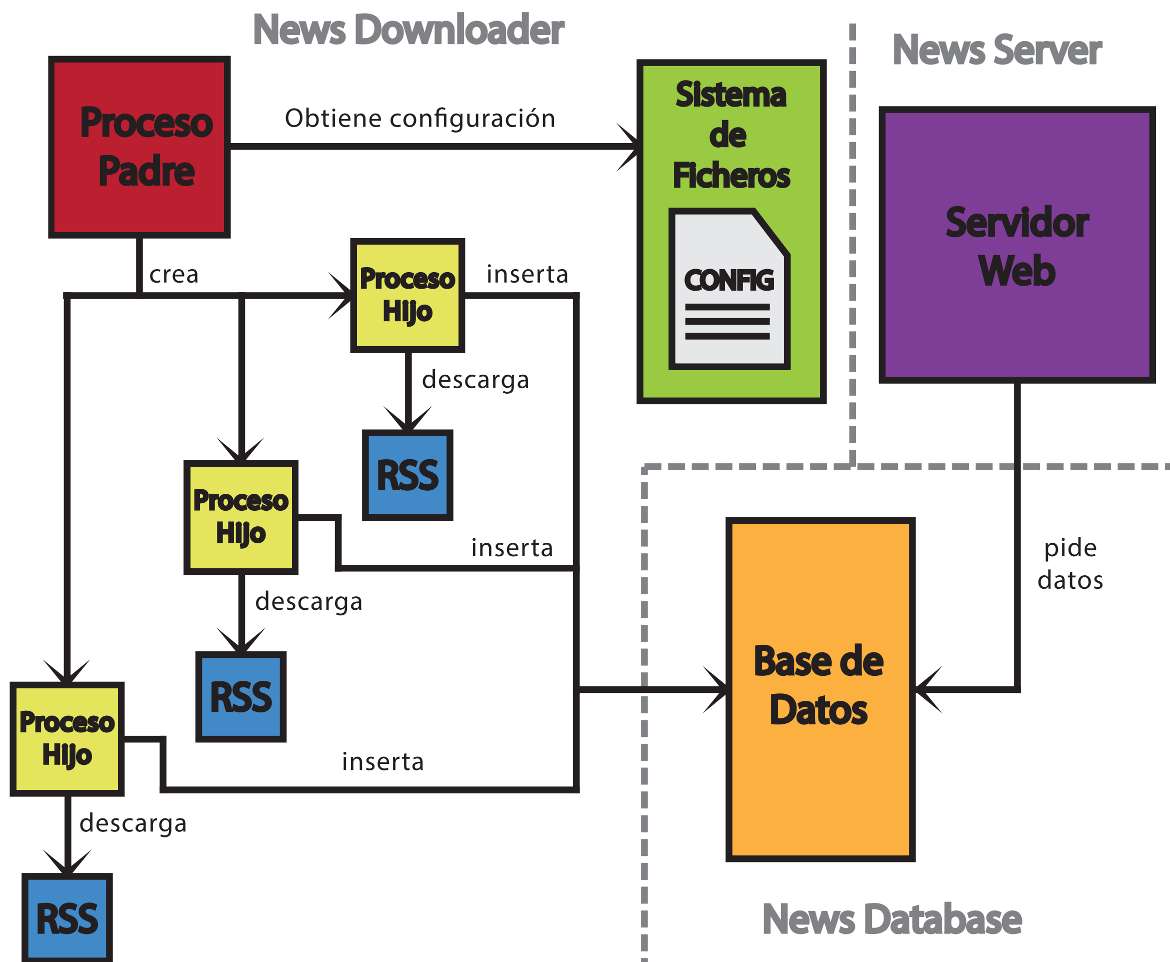


Figura 4.3: Arquitectura del sistema (Segunda iteración)

Lo más interesante que se puede contar sobre el diseño de la segunda iteración es especificar qué acciones debe permitir realizar la API:

URL /

Su propósito es dar información global a la aplicación sobre todos los periódicos que están disponibles. La información debe ser suficiente para que la aplicación pueda exponer los periódicos en una lista y mostrar sus nombres, logos, la noticia en portada y una URL para poder acceder a sus noticias.

En otras palabras, se trata de **Newspaper Index** (ver figura 2.1).

URL /:newspaper_id

Donde **newspaper_id** es la ID de un periódico. Su objetivo es dar toda su información, secciones y noticias.

Es **Newspaper Show** (ver figura 2.2).

Por otro lado, se encuentra la base de datos. Su esquema de datos se ha pensado explícitamente para cubrir las necesidades del backend. Se ha llegado a un diseño de esquema compuesto por tres tablas. Una tabla para periódicos, otra tabla para

secciones y una última para las noticias. La figura 4.4 presenta un diagrama UML representativo del sistema.

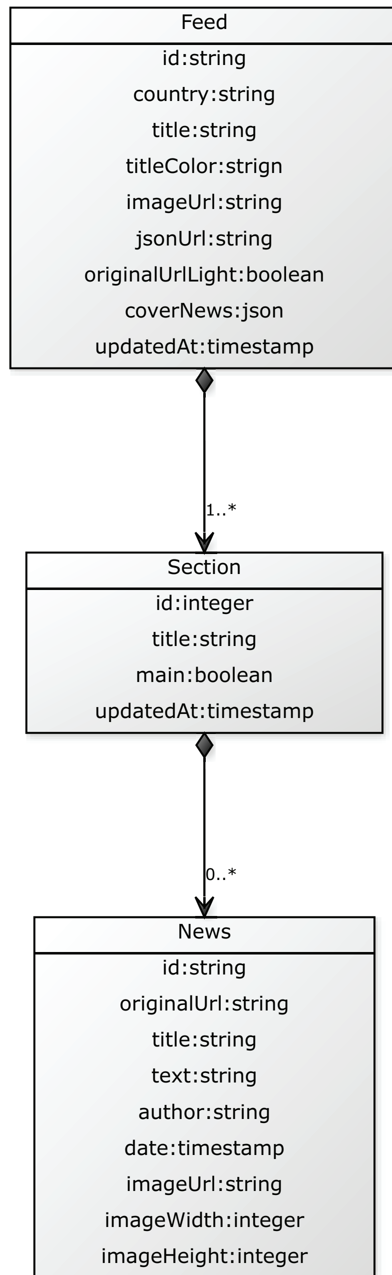


Figura 4.4: Diagrama UML de clases

5

Desarrollo

En este capítulo se explican los detalles principales de la implementación del diseño especificado en el capítulo 4.

Al igual que el capítulo de diseño, este capítulo se divide en dos partes, una por cada iteración.

5.1. Primera iteración

La implementación del News Downloader sigue exactamente el diseño especificado.

Hay un script maestro que se encarga de leer el listado de periódicos. El listado de periódicos al principio empezó siendo un fichero JSON. Sin embargo, cuando se implementaron los callbacks opcionales tuvo que pasar a ser un fichero JavaScript. Esto se debe a que no es posible representar funciones en JSON de una manera simple.

El script maestro crea un proceso hijo por cada periódico de la lista utilizando el siguiente comando:

```
1  childProcess.fork("./modules/feed_process." + feedProcessFormat, [
2    JSON.stringify({
3      "country": feed.country,
4      "uuid": feed.uuid
5    })
6  ]);
```

Figura 5.1: Comando fork() en Node.js

Este comando crea un proceso Node hijo y hace que empiece a ejecutar el contenido de otro fichero JavaScript. Además, el nuevo ejecutable recibe como argumentos de línea de comandos una cadena JSON con el país y la ID del periódico que tiene que descargar y procesar.

Cada proceso hijo instancia un objeto de la clase Periódico y le manda empezar a procesar. La clase Periódico lo que hace es, cada cierto tiempo (depende del periódico en sí, normalmente 15 minutos, pero en el caso de la prensa deportiva es tan sólo 5 minutos) descarga todos los datos de todas sus secciones. Una vez descargado, procesa todos los datos y al final los reúne y genera un objeto JSON que escribe en un fichero.

Por debajo, Periódico hace uso de otras dos clases: Sección y Noticia. Las secciones son las que realmente se corresponden con fuentes RSS. Por tanto, un periódico puede estar descargando de cinco o diez fuentes RSS simultáneamente.

Por su parte, las clases Sección y Noticias se encargan de realizar tareas que tienen sentido a su nivel de competencia. Una noticia puede generar un objeto JSON de su contenido. Una sección puede generar un objeto JSON que contenga los JSON de todas sus noticias, y un Periódico un JSON que contenga los JSON de todas sus secciones. Lo mismo ocurre en otras tareas como descargar (que en esencia se encargan las Secciones mientras que el periódico recoge resultados) y procesar (tarea de la que se encargan las Noticias.)

Otras curiosidades interesantes son:

- Para las IDs de los periódicos se han utilizado UUIDs. Esto permite definir los periódicos de forma no secuencial y que no colisionen nunca.
- Las IDs de las noticias, por otra parte, utilizan el resultado de aplicar SHA a sus URLs. Esto permite obtener siempre la misma ID para cada noticia, permitiendo no tener que volver a procesar del todo una noticia si se encuentra ya procesada en memoria. Aunque en realidad, se vuelve a procesar parcialmente para comprobar la última fecha de actualización; si la noticia ha sido actualizada se realiza el procesado completo.

5.2. Segunda iteración

En la segunda iteración se han agregado los módulos Database y Server. Además, el módulo Downloader ha sido modificado también.

5.2.1. Downloader

Las modificaciones realizadas sobre el Downloader han sido agregar métodos a Periódico, Sección y Noticia que les permitan insertarse o actualizarse a si mismas en la base de datos. Así cada objeto se encarga de su propia persistencia. No es

la manera más eficiente de insertar datos a una base de datos, ya que se realizan tantas operaciones de inserción como noticias haya, pero eso no resulta importante debido a que cada noticia sólo se va a insertar una vez.

5.2.2. Database

La base de datos utilizada, PostgreSQL, dispone de un adaptador para Node.js llamado *pg*.

Este adaptador permite conectarse y trabajar con la base de datos de forma asíncrona, lo que aprovecha el potencial de Node.js. Aunque es de bajo nivel; no ofrece ninguna API especial para realizar consultas. Sólo dos métodos `query(text, [callback])` y `query(config, [callback])`. En el primer método, el primer argumento es un string que debe contener una consulta SQL. En el segundo método, el primer argumento es un objeto con las claves text y values. Su propósito es permitir realizar consultas parametrizadas para protegerse de inyecciones SQL. En ambos casos, el segundo argumento es un *callback* opcional, que se ejecutará cuando responda la base de datos. Se muestran ejemplos de uso en las figuras 5.2 y 5.3. Se puede encontrar más información en la documentación de *pg*[22].

```
1 client.query('INSERT INTO feed (id) VALUES (' + uuid + ')', function(err,
      result) {
2   // Do something
3 });
```

Figura 5.2: Consulta normal realizada con *pg*

```
1 client.query({
2   text: 'INSERT INTO feed (id) VALUES ($1)',
3   values: [uuid]
4 }, function(err, result) {
5   // Do something
6 });
```

Figura 5.3: Consulta parametrizada realizada con *pg*

Por último, el código SQL utilizado para la creación de las tablas de la base de datos se muestra en las figuras 5.4, 5.5 y 5.6.

```
1  CREATE TABLE feed (
2      id TEXT NOT NULL,
3      country TEXT,
4      title TEXT,
5      titleColor TEXT,
6      imageUrl TEXT,
7      jsonUrl TEXT,
8      originalUrlLight BOOLEAN,
9      coverNews JSON,
10     updatedAt BIGINT
11 );
```

Figura 5.4: Definición de la tabla de periódicos

```
1  CREATE TABLE section (
2      id INTEGER NOT NULL,
3      title TEXT,
4      feed TEXT,
5      main BOOLEAN,
6      updatedAt BIGINT
7  );
```

Figura 5.5: Definición de la tabla de secciones

```
1  CREATE TABLE news (
2      id TEXT NOT NULL,
3      originalUrl TEXT,
4      title TEXT,
5      text TEXT,
6      author TEXT,
7      date BIGINT,
8      imageUrl TEXT,
9      imageWidth INTEGER,
10     imageHeight INTEGER,
11     section INTEGER
12 );
```

Figura 5.6: Definición de la tabla de noticias

5.2.3. Server

El desarrollo del Server resultó una de las tareas que requirió mas esfuerzo, no porque hubiera que escribir gran cantidad de código, sino, por la labor de investigación y aprendizaje que ha conllevado.

He invertido bastante tiempo en la búsqueda de un *framework* de desarrollo web para Node.js que me gustase y ofreciera un potencial de desarrollo semejante al que me ofrece *Ruby on Rails*[23] (*framework* que uso muy a menudo). Sorprendentemente, existen muchísimos. Sin embargo, ninguno es lo suficientemente bueno. Probablemente, debido a la inmadurez de Node.js.

Tras intentar implementar el servidor con *frameworks* tales como *Tower*[24], *Sails.js*[25], *Frappé*[26], *CompoundJS*[27] y probar ORMs como *Mongoose*[28] y *Waterline*[29]. Decidí que no estaba obteniendo ninguna ventaja de ellos y decidí escribir la aplicación partiendo desde lo más básico. Lo único que terminé usando fue *Express*[14] debido a lo minimalista que es y las herramientas tan básicas pero útiles que ofrece.

El resultado final es que el código del servidor no sobrepasa las 100 líneas, y cumple con su objetivo perfectamente.

6
Conclusiones

Este desarrollo me ha aportado un mayor conocimiento sobre tecnologías novedosas como Node.js[5] y bases de datos NoSQL[6]. Buscar una solución al problema de obtener correctamente los datos desde fuentes RSS variadas y a la vez conseguir que el sistema sea mantenible ha potenciado mis habilidades de análisis y diseño.

La solución desarrollada resulta estable, sólida y flexible; cumpliendo con las necesidades del proyecto. El Downloader utiliza múltiples procesos para aumentar la velocidad de procesado y reducir el daño en caso de fallo grave. El sistema de *callbacks* opcionales para cada fuente RSS ofrece un nivel excelente de flexibilidad, evitando tener que repetir casi ninguna línea de código, lo que facilita enormemente la mantenibilidad de la aplicación.

El Server, por otra parte, es minimalista, sencillo y cumple con su objetivo. Además, permite añadir nueva funcionalidad fácilmente. Esto es importante porque la mayoría de mejoras futuras (capítulo 7) se centran en a agregar funcionalidades al Server, en vez de al Downloader.

Finalmente, otras de las conclusiones que he obtenido de la realización de este Trabajo de Fin de Máster es que el uso de lenguajes de programación pesados, como por ejemplo Java, no siempre es la solución más eficiente para afrontar el desarrollo de una aplicación web. Sin embargo, en las clases que se han impartido de desarrollo web durante el máster, no se han tratado desarrollos con otro tipo de lenguajes. Aunque no fueran muchas porque el máster sólo dura un año. Sus enseñanzas se han enfocado principalmente al lenguaje de programación Java. Sin embargo, en proyectos del ”mundo real” o del entorno productivo, este tipo de lenguajes no siempre son la mejor opción para el desarrollo. En este sentido, quizá debería incluirse en el programa de alguna asignatura el uso de lenguajes dinámicos como, en este caso, JavaScript. No es un lenguaje que ofrezca un rendimiento

excepcional (aunque debido a las optimizaciones de Google V8 cada vez resulta ser más rápido). Lo que sí ofrece es flexibilidad a la hora de desarrollar. Esa flexibilidad permite a un desarrollador trabajar mucho más rápido, generar prototipos e implementar nuevas funcionalidades rápidamente. Esto en parte se debe a que suelen ofrecer herramientas de muy alto nivel, haciendo que sea más divertido de trabajar con ellos (esto se detalla más en la sección 3.1).

7
Mejoras futuras

7.1. Peticiones con timestamps

Durante la segunda iteración se ha añadido un problema debido al servidor y base de datos. Antes esto no ocurría porque el Downloader escribía a disco ficheros JSON que un servidor como Nginx o Apache pudieran servir directamente, en cambio ahora, hay una aplicación web que accede a base de datos para las respuestas lo que conlleva bastante procesamiento. El problema es que no se obtiene mayor ventaja *todavía* porque cuando una aplicación pide las noticias de cierto periódico, se le envía todo el objeto JSON con las noticias. En caso de que el móvil tuviera noticias actualizadas por última vez hace 5 minutos, en vez de enviársele los últimos 5 minutos de actualizaciones, se le vuelve a enviar el objeto JSON entero, resultando en un aumento innecesario de carga de la red y de tiempo de procesamiento tanto del dispositivo móvil como del servidor. Es necesario especificar un mecanismo para comunicar este tipo de operaciones. Esta mejora se planificará y afrontará en futuras aplicaciones junto con el equipo encargado del desarrollo de la aplicación móvil.

7.2. Utilización de caché

Resulta interesante la opción de añadir una caché (por ejemplo, Redis[30] o Memcached[31]) entre la base de datos y el servidor. En caso de tener datos todavía válidos, es posible obtenerlos de la caché en vez de recalcular. El problema es que hay que establecer un modo de operación para que se sepa cuando los datos dejan de ser válidos y haya que volver a recalcularlos desde la base de datos. Esta

medida será obligatoria cuando la aplicación salga a producción y empiece a ser utilizada por gran número de usuarios.

7.3. Posible Optimización de uso de memoria

El Downloader maneja alrededor de 25 procesos (actualmente). Cada proceso es una instancia de Node.js, que consume entre 50 y 100 MBs de memoria RAM. Si el número de periódicos creciera mucho sería necesario considerar agrupar múltiples bajo de un único proceso para reducir el consumo de memoria. Aunque esto podría conllevar un descenso en la estabilidad, porque si una fuente diera problemas, podría impedir el procesado de las otras que estuvieran asignadas al mismo proceso.

7.4. Comentarios en noticias

La razón principal por la que se ha modificado la arquitectura original para utilizar una base de datos la base de datos es para poder implementar funcionalidades como ésta en el futuro. Resultaría interesante permitir a la gente comentar las noticias desde la aplicación y que se formasen discusiones. Esto requiere persistencia, de ahí la necesidad de una base de datos que ha sido añadida en la segunda iteración.

Bibliografía y Referencias

[1] **RSS** : http://www.whatisrss.com/

[2] **JSON** : http://www.json.org/

[3] **Nginx** : http://nginx.org/

[4] **Apache Server** : http://httpd.apache.org/

[5] **Node.js** : http://nodejs.org/

[6] **NoSQL** : http://nosql-database.org/

[7] **Sarah Mei** : Why You Should Never Use MongoDB, http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/

[8] **CoffeeScript** : http://coffeescript.org/

[9] **Benchmarks Game (Javascript vs C)** : http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=all&lang=v8&lang2=gcc&data=u64

[10] **Benchmarks Game (Javascript vs Java)** : http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=all&lang=v8&lang2=java&data=u64

[11] **Benchmarks Game (Javascript vs Python)** : http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=all&lang=v8&lang2=python3&data=u64

[12] **Benchmarks Game (Javascript vs Ruby)** : http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=all&lang=v8&lang2=yarv&data=u64

[13] **Cheerio** : https://github.com/cheeriojs/cheerio

[14] **Express** : http://expressjs.com/

[15] **Feedparser** : https://github.com/danmactough/node-feedparser

[16] **Iconv** : https://github.com/bnoordhuis/node-iconv

[17] **Imagesize-ex** : https://github.com/qlalfqjs5/imagesize.js

[18] **UnderscoreJS** : `http://underscorejs.org/`

[19] **Winston** : `https://github.com/flatiron/winston`

[20] **Loggly** : `https://www.loggly.com/`

[21] **flock()** : `http://linux.die.net/man/2/flock`

[22] **pg wiki** : `https://github.com/brianc/node-postgres/wiki`

[23] **Ruby on Rails** : `http://rubyonrails.org/`

[24] **Tower** : `http://tower.github.io/`

[25] **Sails.js** : `http://sailsjs.org/`

[26] **Frappé** : `https://github.com/dweldon/frappe`

[27] **Compound.js** : `http://compoundjs.com/`

[28] **Mongoose** : `http://mongoosejs.com/`

[29] **Waterline** : `https://github.com/balderdashy/waterline`

[30] **Redis** : `http://redis.io/`

[31] **Memcached** : `http://memcached.org/`