Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Revisión de los paquetes para realizar web scraping en R: Análisis cualitativo y cuantitativo

Autor: Daniel, Francisco López

Tutor/es: Juan José, Cuadrado Gallego

2018

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Revisión de los paquetes para realizar web scraping en R: Análisis cualitativo y cuantitativo

Autor: Daniel, Francisco López

Tutor/es: Juan José, Cuadrado Gallego

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

FECHA:

Revisión de los paquetes para realizar web scraping en R: Análisis cualitativo y cuantitativo

Francisco López, Daniel

Índice general

Capítulo 1

Introducción. Análisis del mercado

1.1. Resumen

En este trabajo, se analiza el mercado de los paquetes de *web scraping* en R, escogiendo finalmente **XML**, **rvest** y **seleniumPipes** para su análisis en profundidad. Se crean diversos tests para probar los paquetes, verificando cómo se comportan de forma general y observando cómo actúan al trabajar con formularios, sesiones y con páginas que requieren ejecución de *JavaScript*. Posteriormente se hace *web scraping* a estos tests. Finalmente se realiza un test de *performance* para determinar en la medida de lo posible el rendimiento cuando hay un número apreciable de peticiones.

1.2. Abstract

In this paper, the market for *web scraping* packages in R is analyzed, choosing **XML**, **rvest** and **seleniumPipes** for an in-depth analysis. Tests are created to test the packages, verifying how they behave in a general way and observing how they work with forms, sessions and with pages that require *JavaScript* execution. Subsequently, *web scraping* is done for these tests. Finally, a performance test is executed to determine, as far as possible, the performance when there is an appreciable number of requests.

1.3. Palabras clave

web scraping; *bot*; *parsing*; *HTML*; *performance*;

1.4. Introducción

1.4.1. Qué es el web scraping

El *web scraping* es una técnica en la que mediante el uso de software, se puede navegar a través de la web para obtener datos concretos [1][2]. El *web scraping* pretende emular la navegación de un ser humano, accediendo a diversas páginas, obteniendo de forma automática la información deseada (ya sean encabezados, enlaces, etcétera) [1][2]. Así pues, se puede definir el *web scraping* como la construcción de un algoritmo para descargar, *parsear* (hacer un análisis léxico/sintáctico/semántico que permita obtener una estructura fácilmente accesible con selectores) y organizar datos de la web de forma automatizada. [3].

1.4.2. Posibilidades prácticas del web scraping

Se pueden distinguir muchos posibles ejemplos de uso de *web scraping* (basado en [3]):

- Pueden existir tablas interesantes en páginas como *REE* o *Wikipedia*, de las que obtener datos para realizar una análisis estadístico descriptivo.
- Se desean obtener comentarios u opiniones sobre un cierto producto, para hacer minería de texto, o construir modelos predictivos que detecten a los *haters*.
- Se puede obtener una lista de inmuebles para hacer una geovisualización sobre mapas de *Google Maps* u *OpenStreetView*.
- En general, se desea obtener más información para enriquecer u optimizar el proceso productivo de una empresa.
- Se podrían realizar análisis sociales.
- Se podrían monitorizar noticias para ver los temas de actualidad y actuar en consecuencia (gestionar redes sociales de acuerdo al tema que esté a la orden del día).
- Etcétera.

1.4.3. Tipologías de web scraping

A la hora de hacer *web scraping*, se pueden distinguir varios supuestos:

- Que la información se extraiga de una o varias páginas web, sin que como resultado de *scrapear* dichas páginas web se generen nuevas páginas que visitar. Esto se puede denominar *web scraping* de forma genérica. La idea de este tipo de enfoque es extraer información concreta de fuentes concretas ya verificadas, porque se sabe que en dichas webs están los datos y la información requeridos y se sabe cómo extraerlos para generar valor.

- Que entre la información que se extraiga se encuentren los enlaces contenidos en las páginas los cuales luego realimentan la entrada de enlaces (siguiendo cierta lógica o conjunto de reglas políticas), en cuyo caso se hablaría de una variedad concreta denominada *web crawling*, y el software que realiza estas tareas se suele denominar *spider* o *crawler* [4]. En general en este tipo de enfoque no necesariamente se suele conocer la estructura concreta de las páginas ni se suele investigar todo el universo de webs a *scrapear*. Este enfoque es propio de los buscadores, que suelen poner este tipo de programas o *bots* a funcionar, rastreando la web, con el objetivo de evaluar las webs puntuándolas con potentes métodos de *web scraping* y de *minería de datos* para ampliar su colección interna de páginas y que las páginas mostradas a la hora de buscar sean relevantes y acordes con los términos de búsqueda.

- Que lo que se desee hacer con la información obtenida sea programar pruebas de software (tests de unidad, de de regresión, etcétera). Se configurarían una serie de supuestos a probar, indicando las entradas y las salidas correctas, y verificando que realizando los pasos necesarios se llega de la entrada a la salida [5][6].

1.4.4. Cuándo utilizar web scraping

La web contiene muchas fuentes de datos muy diversas e interesante. Desafortunadamente, debido a que la web tiene una naturaleza desestructurada y no permite con facilidad recopilar o exportar los datos almacenados en ella de forma sencilla [3]. Para poder reunir un conjunto de datos completo de forma automática, entra en escena *web scraping* [3].

No obstante, se podría pensar que sería más interesante utilizar ficheros más estructurados (JSON, XML, Excel, CSV, etcétera) o bien utilizar API's que provean los datos deseados en vez de intentar extraer directamente los datos de la web. Hay sitios web que proveen dichas API's o en algunos casos alguno de estos ficheros, sin embargo, el uso de *web scraping* puede ser muy interesante y bastante preferible en los siguientes casos [3]:

- El sitio web del que se desea extraer la información no proporciona ninguna API y no hay forma de obtener otro tipo de ficheros con la información deseada.

- La API a la que se puede acceder no es gratis mientras que el acceso a la web es libre (y gratuito).

- La API tiene limitación de peticiones, y la extracción de información se ve entorpecida por los límites de la API.

- La API no provee todos los datos que se desean obtener mientras que el sitio web sí que lo hace.

La idea fundamental es que si los datos que se desean extraer se pueden observar en pantalla a la hora de navegar por el sitio web, se pueden extraer y recuperar mediante un programa, de tal manera que se pueden almacenar, limpiar y utilizar como se requiera [3].

No obstante, si existen alternativas viables como el uso de una API o el uso de ficheros estructuras, es muy aconsejable el uso de estos ya que el *web scraping* es una técnica que consume bastantes recursos y que tiene una cierta entidad.

1.4.5. Aspectos legales del web scraping

El uso del *web scraping* como técnicageneralmente es legal, pero hay que antender a varios factores. En una sentencia del Tribunal Supremo de 9 de octubre de 2012 número 572/2012, de Ryanair contra Atrápalo, se determinaron en qué circunstancias se puede considerar que dicha técnica es legal o ilegal [7]:

- Si la conducta que se lleva a cabo supone una competencia desleal, pudiendose generar una duda ante el consumidor dando a entender que la web de origen y la web de destino (esto aplica a casos en los que se hace una web con datos *scrapeados* de otra) tiene una vinculación inexistente o se usa dicha información para favorecer la web última, se puede considerar que la técnica se ha utilizado ilegalmente.

- Si se ha incumplido la propiedad intelectual hacidno *web scraping* sobre contenidos sujetos a dicha propiedad intelectual o bien sobre estructuras de datos (bases de datos) amparadas bajo esa propiedad intelectual, se considera que la técnica se ha utilizado de forma ilegal.

- Es ilegal hacer *web scraping* sobre datos amaprados por la ley orgánica de protección de datos.

En resumen, para tener una cierta garantía de que se está haciendo *web scraping* conforme a la legalidad, simplemente hay que atender a estas precisiones [8][9]:

- Obedece las directivas que indique el fichero `robots.txt` (el fichero de exclusión de robots).

- Realiza el algoritmo de *web scraping* teniendo en cuenta el respecto a los recursos del servidor: la actividad de *web scraping* puede afectar al rendimiento del servidor.

- Cumple todo lo mencionado en los términos de uso de la web relativo a *web scraping*.

- Respeta la legislación en materia de *copyright* y de propiedad intelectual.

1.5. Objetivos de este trabajo

Los objetivos de este trabajo de fin de grado básicamente consisten en analizar el estado de arte de los paquetes que permiten hacer *web scraping* en R y escoger los tres que pueden ser más interesantes de analizar (por su relevancia en el mercado, funcionalidad que aportan, curva de aprendizaje, etcétera).

Una vez que se han escogido estos tres paquetes, se procede a crear una batería de ejercicios o de supuestos prácticos donde se prueban las siguientes características:

- **Test de propósito general:** El propósito de este test es practicar la extracción de datos, realizando cierta variedad de ejercicios, extrayendo valores y atributos y utilizando selectores de todo tipo, para operar con el árbol que genera el *parser* concreto. Se finaliza extrayendo una tabla.

- **Test de envío de formularios:** El propósito de este test es ver si es viable (y en caso afirmativo de qué manera) el envío de un formulario y la posibilidad de observar la respuesta obtenida.

- **Test de gestión de *cookies*:** El propósito de este test es verificar si el paquete concreto puede mantener a lo largo de la navegación las *cookies* donde se almacena el identificador de sesión que debe ser enviado por el servidor para cada petición, de tal manera que el servidor dé una respuesta al cliente en base a la actividad de navegación que ha realizado anteriormente.

- **Test de renderizado de *JavaScript*:** El propósito de este test es verificar si el paquete concreto es capaz de renderizar la página ejecutando el código *JavaScript* contenido en ella antes de generar el árbol y permitir al desarrollador del algoritmo de *web scraping* la posibilidad de acceder a él una vez que ya se ha ejecutado dicha fase de renderizado.

Una vez realizados estos dos objetivos iniciales, simplemente hay que desarrollar los códigos que permiten extraer la información deseada de los supuestos prácticos con cada uno de los paquetes elegidos.

Para terminar y de cara a probar estos paquetes en una situación en la que el rendimiento puede ser determinante, se diseña una prueba en la que se realizan muchas peticiones y se extrae algún dato de las páginas descargadas con el fin de medir lo que tardan en realizarse estas operaciones. Con el resultado de los tests anteriores más este test de rendimiento, se deben obtener unas conclusiones que determinen en qué circunstancias es aconsejable utilizar uno u otro paquete.

Se asumirá que el lector tiene nociones básicas tanto de CSS como de *XPath*. No obstante, a lo largo de los capítulos se aconsejará diversa bibliografía para reforzar estos aspectos para aquellos lectores que los conozcan menos.

1.6. Proceso de búsqueda de potenciales paquetes candidatos a analizar

En primer lugar hay que buscar todos los elementos (o al menos los más representativos) que conforman la población de paquetes en el mercado para realizar *web scraping* en R. Se ha optado por realizar una búsqueda a través de los siguientes repositorios:

- The Comprehensive R Archive Network (CRAN) [10], búsqueda disponible a través de RDocumentation [11].
- R-Forge [12].
- Omegahat [13].
- Bioconductor [14], búsqueda disponible a través de RDocumentation.
- GitHub [15], búsqueda disponible a través de RDocumentation.

El uso de RDocumentation es aconsejable, ya que simplifica enormemente la búsqueda de los paquetes, siendo esta mucho más amigable y teniendo una certeza de encontrar casi la totalidad de los paquetes disponibles en los repositorios que cubre. RDocumentation permite hacer una búsqueda en más de 14.000 paquetes entre CRAN, GitHub y BioConductor.

1.6.1. Tipos de paquetes orientados a webscraping según su alcance

Existe una sección de CRAN que ha sido encontrada a través de RDocumentation, denominada *CRAN Task View: Web Technologies and Services* [16]. Además del buscador principal, RDocumentation tiene un área donde se pueden ver las *task view* que tiene CRAN. Esta *task view* en particular, expone diversos paquetes que permiten utilizar R para manejar los recursos que la World Wide Web provee [16]. La consulta de esta *task view* ha permitido entender que se puede hacer la tarea de *web scraping* mediante dos enfoques diferentes, en base a las herramientas existentes:

- **Enfoque de análisis sin renderizado previo:** Utiliza un paquete para hacer un análisis léxico y sintáctico (lo que se conoce como *parsear*) estructuras XML o HTML contenidas en una cadena y que crean una estructura que permite extraer la información que se requiere de dicha web descargada utilizando *XPath* o selectores CSS. Este enfoque es posible (o no) que requiera de otro paquete o utilidad que realice la petición HTTP y descargue la web.

  Además, este tipo de bibliotecas, si gestionan la descarga, en algunos casos permiten gestión de sesiones de navegación y/o envío de formularios, por lo que supone abstraer esas funcionalidades, pudiéndose olvidar la necesidad de una herramienta o algoritmo especializado que lo realice, simplemente se confía en la herramienta o paquete para estas necesidades.

- **Enfoque de análisis con renderizado previo:** Utilizar un paquete integrado que aporte más potencia respecto al anterior, introduciendo una "fase intermedia" de renderizado de la página descargada mediante un *browser engine* o motor de renderizado web. Esta fase intermedia de renderizado puede variar en función de la utilidad, ya que puede conllevar dirigir un navegador previamente instalado, tener la propia herramienta un renderizado propio, etcétera.

  Esta última opción es la más compleja, pero es necesaria si la página realiza carga asíncrona de elementos con JavaScript, puesto que estos no aparecerían si no se ejecuta el código JavaScript que alberga dicha página web. Cada vez más webs hacen uso de técnicas AJAX para carga de forma asíncrona ciertos elementos de la página, y cada vez se extiende más el uso de bibliotecas de desarrollo como Polymer, AngularJS o JQuery entre otras, que en resumen, pasan parte de la lógica del lado del servidor al lado del cliente y/o favorecen la interactividad. Estas páginas no podrán ser analizadas si no se renderizan antes.

1.6.2. Paquetes encontrados que realizan análisis sin renderizado previo

A continuación se exponen aquellos paquetes que permiten en base a una cadena dada (que se presupone que está conformada por una sintaxis válida HTML) o bien en base a una URL, parsearla para acceder a los elementos del DOM sin hacer ningún tipo de renderizado previo del código JavaScript que pudieran eventualmente contener:

- XML.
- xml2.
- rvest.
- selectr.
- scrapeR.
- rCrawler.
- htmltab.
- XML2R.
- boilerPipeR.

1.6.3. Paquetes encontrados que realizan análisis con renderizado previo

- rdom.

- relenium.
- splashr.
- RSelenium.
- seleniumPipes.
- webdriver.
- decapitated.

1.7. Sinopsis de los paquetes localizados

A continuación se hará una breve sinopsis de los paquetes localizados. Dicha sinopsis tiene como objetivo conocer brevemente las características de cada paquete, con el fin de tomar una decisión sobre cuales de estos paquetes analizar y comparar en mayor profundidad. Esta sección bajo ningún concepto pretende hacer un análisis profuso, sino que es una breve exposición orientada a hacer una selección de los candidatos.

Los paquetes se describirán a continuación en función de la categoría en la que fueron asignados al ser encontrados.

1.7.1. Paquetes encontrados que realizan análisis sin renderizado previo

XML

Este paquete permite tanto el análisis de documentos XML/HTML existentes así como la generación de nuevos con R y también con S mediante varios enfoques. También provee acceso a un intérprete XPath que permite consultar nodos concretos de los documentos [17]. Se puede acceder a los documentos mediante HTML, FTP y también de forma local [17][18].

Este paquete no permite utilizar selectores CSS para hacer consultas sobre los XML o HTML *parseados*. Adicionalmente, hay que destacar que tiene algunos problemas de liberación de memoria que no puede resolver adecuadamente el *garbage collector* de R, por lo que han creado un artículo en el que explican cómo hacer de forma adecuada la gestión de memoria [19].

Hay que resaltar por último, que parece que actualmente no hay nadie que mantenga el proyecto [20] y que al parecer hay algunos problemas en algunos usuarios de Windows, que aducen que hay pérdidas de memoria (por lo que el *heap* se va llenando progresivamente) [18].

Sin embargo según *CRAN* [20] y *RDocumentation* [21] la versión más reciente data de Agosto de 2018, por lo que es posible que estos errores sucedieran en el pasado y que esta página no haya sido convenientemente actualizada.

Además se puede observar que en las estadísticas de descarga de *RDocumentation* es un paquete muy descargado y además hay bastantes paquetes que lo

tienen como dependencia siendo estas descargas indirectas como consecuencia de que otros paquetes dependan de él muy significativas (en el mes de agosto de 2018 aproximadamente la mitad de las descargas son por descarga de dependencias y los días de menos descargas globales rondaba en torno a las 1000 descargas y los que más se acerca a 5000 descargas diarias).

xml2

xml2 es un paquete que permite trabajar con ficheros XML (extrapolable a HTML) utilizando una interfaz simple y consistente. Está basado sobre la biblioteca de C "libxml2" [22][23][24].

Además, según el autor, tiene los mismos objetivos que el paquete XML (paquete que se suele utilizar en R para "parsear" ficheros o cadenas XML) pero con algunas ventajas como las siguientes [23][24]:

- ***xml2*** cuida la gestión de la memoria de forma transparente, liberando la memoria utilizada por un documento XML tan pronto como se pierden todas las referencias que apuntan a él.
- ***xml2*** tiene una jerarquía de clases muy sencilla que permite no preocuparse por el tipo de objeto se está gestionando, haciendo xml2 lo correcto. Las tres clases clave que están definidas son las siguientes [23][24]:
  - `xml_node`: Un nodo cualquiera del documento.
  - `xml_doc`: El documento entero. Actuar sobre el nodo que representa el documento suele ser lo mismo que actuar sobre el nodo raíz del documento.
  - `xml_nodeset`: Un conjunto de nodos dentro del documento. Las operaciones sobre esta clase están vectorizadas, lo que implica que dichas operaciones se aplican a cada uno de los nodos contenidos en el conjunto.
- ***xml2*** tiene una gestión de nombre de espacio en expresiones **XPath** más adecuada.

Como principal inconveniente, es que no incorpora de forma "nativa" la posibilidad de seleccionar nodos o conjuntos de nodos utilizando selectores CSS, por lo que todas las selecciones de nodos se deben hacer con XPath.

rvest

rvest es un paquete cuya funcionalidad es hacer de *wrapper* de los paquetes ***xml2*** y ***httr***, haciendo que la cooperación de dichos paquetes se pueda hacer de forma muy útil y sencilla, facilitando con ello la descarga y manipulación de ficheros XML y HTML [25].

Entre las posibilidades que permite **rvest**, destacan las siguientes [26][27][25]:

- Crear un objeto con un documento HTML gestionable por **rvest** desde una URL, desde disco duro o desde una cadena.
- Seleccionar partes de un documento utilizando **XPath** o bien **CSS**.
- Obtener componentes de las partes seleccionadas, pudiendo acceder al nombre de las etiquetas, al contenido o a los atributos que las mismas contienen.
- Todo lo anterior se aplica también a documentos XML.
- *Parsear* tablas y convertirlas en *data frames*.
- Extraer, modificar y enviar formularios.
- Detectar y reparar problemas de codificación.
- Navegar a través de un sitio web como si de un navegador web se tratase, pudiendo hacer uso de secciones, ir hacia atrás y hacia adelante, ir a otra página, etcétera. Esta función está en progreso actualmente según el autor.

Además todas las funciones están diseñadas para ser encadenadas por el operador de tubería incluido en el paquete **magrittr**.

selectr

Selectr es un paquete cuya funcionalidad principal es transformar selectores CSS al equivalente en XPath, de tal manera que se puedan utilizar herramientas que solo trabajan con XPath haciendo una conversión previa desde CSS [28]. También se proveen ciertas funciones convenientes para trabajar con CSS sobre nodos XML [28].

En resumidas cuentas, la mayor parte de la funcionalidad que ***selectr*** provee se resume en los siguientes items [29]:

- Traduce un selector CSS a una expresión XPath con `css_to_xpath()`.
- Consultar nodos en un objeto ***XML*** o ***xml2***, obteniendo el primer nodo con `querySelector()` o todos con `querySelectorAll()`.

scrapeR

ScrapeR es un paquete que persigue hacer *web scraping* sobre documentos "basados en Web" [30][31]. Consta de una única función que realiza la obtención y crea el árbol DOM que se puede iterar posteriormente. No dispone de funciones propias de recorrido del DOM, por lo que recurre a las funciones y métodos del paquete ***XML*** para realizar su labor.

Tiene una ventaja significativa, y es que permite paralelizar la petición de varias URL's a la vez (aunque no se sabe si de forma subyacente realiza una paralelización efectiva) e indicar un campo personalizado de *User Agent* entre otros [32].

Rcrawler

Rcrawler permite realizar ***web scraping*** y ***web crawling*** (recorrer webs en profundidad para extraer información de ellas) de forma paralela. Está diseñado para rastrear, parsear y almacenar páginas web para producir datos que puede ser utilizados con posterioridad para aplicaciones de análisis [33][34].

Según el autor, la diferencia con **rvest** consiste en que **rvest** extrae datos de una página específica haciendo consultas mediante selectores, mientras que ***Rcrawler*** accede automáticamente a todas las páginas web del sitio y extrae los datos que se necesitan con **un solo comando** [35]. Además, ***Rcrawler*** permite ayudar en el **estudio de la estructura del sitio web** construyendo una red para representar la estructura de enlaces internos y externos que conforma el sitio web mediante nodos y aristas [35].

Con una sola invocación a la función principal de ***Rcrawler*** se pueden realizar las siguientes tareas[35]:

- Descargar todas las páginas HTML del sitio web.
- Cargar los ficheros HTML recolectados a la memoria para poder manejarlos con el entorno de R.
- Extraer datos estructurados de todas las páginas del sitio web (bien sea con *XPath* o bien con CSS).
- Filtrar los contenidos que se desean *scrapear* utilizando términos de búsqueda de tal manera que Rcrawler pueda ir a través de los enlaces y extraer solo las solo las páginas web que tienen interés por los términos de búsqueda elegidos.

Algunos sitios web al ser muy grandes, no se dispone de suficiente tiempo o recursos para recorrerlos. En caso de que solamente sea relevante una parte de la web en concreto, se disponen de parámetros de control para controlar el proceso de ir recorriendo las páginas del sitio web como son los siguientes [35]:

- Filtrar las URL's recolectadas o *scrapeadas* mediante ciertas palabras claves o bien mediante expresiones regulares.
- Controlar cuántos niveles de profundidad se quieren recorrer desde la página raíz que se ha empezado a *scrapear*.
- Se puede optar por ignorar ciertos parámetros de la URL durante el proceso de *crawling* (aquellos cuya presencia no afecta al contenido generado, para evitar que el mismo contenido sea analizado varias veces).

También permite como se indicaba anteriormente analizar la estructura del sitio web así como algunos detalles de configuración adicional a la hora de configurar el proceso de *web crawling*.

En conclusión, es un paquete muy completo a la hora de hacer *web scraping* y *web crawling* si se busca escalar dichas labores a un sitio web entero.

Como inconveniente a destacar, es que la curva de aprendizaje es mayor y la lógica de *scraping* debe ser igual para todas las páginas, por lo que podría no ser especialmente indicado para tests pequeños o lógicas que requieran descargar una sola página.

htmltab

Este paquete es diferente al resto, ya que está centrado fundamentalmente en la extracción y en la limpieza posterior de datos que se obtienen de tablas.

La filosofía de este paquete se basa en que las tablas HTML son una fuente de datos con gran valor, pero extraer su contenido y transformarlo a un formato que tenga utilidad puede ser tedioso. Este paquete ayuda en esa tarea aportando tres ventajas principales [36][37][38]:

1. La función expande los fragmentos los tramos combinados de celdas (ya sean columnas o filas) en el encabezado y en las celdas del cuerpo.
2. Los usuarios tienen más control para identificar las filas y las columnas que aparecerán en la tabla generada, incluyendo la información semántica que pudiera haber.
3. La función *preprocesa* el código de la tabla y corrige error de sintaxis que pudiera haber y elimina partes innecesarias, con el fin de reducir la complejidad de un proceso de limpieza que pudiera eventualmente ser necesario posteriormente.

XML2R

Este paquete consiste en un *framework* que según el autor reduce el esfuerzo para transforma el contenido XML en tablas que permiten mantener las realizaciones entre padres e hijos [39]. No obstante, el concepto parece ser que está orientado a generar tablas finales a partir de XMLs combinando las tablas que se van generando. No parece que este paquete nos facilite la labor de *web scraping* por lo que se descartará inmediatamente en la elección de candidatos.

boilerpipeR

Este paquete es un *wrapper* que permite extraer texto principal de ficheros HTML, eliminar anuncios, *sidebars* y *headers* utilizando la biblioteca de Java *boilerpipe* [40][41]. La extracción heurística de esta biblioteca tiene un gran rendimiento en gran cantidad de sitios web [40][41].

Los extractores ideados (que implementan cada uno un algoritmo heurístico) son los siguientes [41]:

- **ArticleExtractor**: Extractor de texto enfocado a la extracción de noticias.
- **ArticleSentencesExtractor**: Extractor de texto enfocado a la oraciones de artículos de noticias.

- **CanolaExtractor**: Extractor de texto entrenado en un *krdwrd*.
- **DefaultExtractor**: Extractor de texto genérico.
- **KeepEverythingExtractor**: Extractor que marca todo como contenido.
- **LargestContentExtractor**: Extractor de texto que extrae el componente de texto más largo de una página.
- **NumWordsRulesExtractor**: Extractor de texto bastante genérico basado únicamente en el número de palabras por bloque.

1.7.2. Paquetes encontrados que realizan análisis con renderizado previo

rdom

Rdom es un paquete bastante simple que utilizando *phantomjs* como navegador de forma subyacente, permitiendo acceder a las páginas, renderizarlas y finalmente extraer el elemento que se desea utilizar [42].

una función muy simple, que puede ser de ayuda si se desea extraer un único elemento. Si se desean extraer varios, ya habría que hacer un enfoque combinado con otros paquetes como ***rvest***, ***XML*** u otros [43]. Además, entre las cuestiones que se destacan para mejorar, consta el manejo de múltiples URL's, ya que para cada URL que se desea analizar se crea una instancia nueva de *phantomjs* [44].

relenium

Relenium es un *driver* de navegadores para *Selenium*. El objetivo es poder automatizar a través de *Selenium* un navegador, escribiendo las acciones que debe hacer así como el tratamiento posterior de los datos recopilados mediante R. El principal propósito es hacer *web scraping* de forma fácil y potente [45]. Es un desarrollo español liderado por Lluís Ramon y Aleix Riuz de Villa, que actualmente no está bajo mantenimiento, aconsejando dichos autores hacer uso de *RSelenium* [46].

splashr

Permite comunicarse con el servicio de renderizado denominado *Splash*. *Splash* es un servicio de renderizado de *JavaScript* que consiste en un navegador web ligero con una API implementada en Python y provee parte de la funcionalidad que puede proveer ***RSelenium*** y ***seleniumPipes*** [47][48]. Algunas de las funcionalidades que provee *Splash* son las siguientes:

- Procesar múltiples páginas web en paralelo.
- Obtener resultado HTML y realizar capturas de pantalla.
- Reglas tales como deshabilitar imágenes o activar el uso del bloqueador de anuncios con el fin de hacer el renderizado más rápido.

- Ejecutar código *JavaScript* personalizado en el contexto de la página.

Una particularidad que tiene este paquete es que para utilizar selectores de acceso para extraer determinada información, se debe utilizar ***xml2*** o ***rvest***.

RSelenium

El objetivo de ***RSelenium*** es facilitar la conexión a un servidor de *Selenium*. Provee enlace utilizando R a la API de Selenium WebDriver [49][50].

Selenium es un proyecto enfocado en automatizar navegadores web [49]. ***RSelenium*** permite hacer pruebas de unidad y de regresión en las aplicaciones web que se desarrollen (también permite hacer *web scraping*) y utilizando un gran rango de navegadores y de sistemas operativos [49][50].

Gracias a esto permite realizar entre otras las siguientes tareas [49][50]:

- Navegar a través de las página.
- Acceder a los elementos del DOM utilizando selectores de id, de clase, de nombre de etiqueta, selectores *XPath* o CSS.
- Enviar eventos de teclado y de ratón a los elementos de la página que se determinen.
- Inyectar *JavaScript* en la página.
- Utilizar marcos y ventanas diferenciadas.

seleniumPipes

Habiendo visto todas las utilidades previas, se extrae la conclusión de que ***SeleniumPipes*** es un paquete que es a ***RSelenium*** lo que ***rvest*** es a **xml2**. Es un cliente *webdriver* basado en la especificación del consorcio W3C [51] [52] (al igual que ***RSelenium***), pero que ha sido construido para una sintaxis mucho más ligera con **magrittr**, **xml2** y **httr** [53]. Gracias a esta implementación se puede utilizar de forma sencilla encadenamiento de instrucciones mediante tuberías, al igual que en **rvest**.

Entre las características básicas que se pueden ver en su manual de operación básica se pueden destacar las siguientes [54]:

- Permite conectarse a cierto conjunto de navegadores entre los que destaca **Firefox**, **Chrome**, **Edge** o **PhantomJS** entre otros.
- Utiliza notación de tuberías para facilitar la sintaxis encadenando funciones.
- Permite encontrar elementos e interactuar con ellos (accediendo a enlaces, rellenando cajas, etcétera).
- Permite ejecutar JavaScript a discreción dentro del navegador.
- Facilita la gestión de errores en la interacción con la web.
- Gestiona la casuística de varias ventanas y/o frames.

webdriver

Este paquete es una implementación más del API webdriver [51], que permite dirigir un navegador compatible y que puede ser utilizado para hacer pruebas a aplicaciones web [55][56]. Según subrayan los autores, en teoría funcionaría con cualquier navegador, pero ellos lo han probado solo con *PhantomJS* [55][56].

decapitated

Este paquete permite gestionar *Google Chrome* mediante programación en R [57].

Provee las siguientes principales funciones [57]:

- Imprimir página a PDF.
- Leer una página desde una URL y obtener los elementos del DOM renderizados (se requeriría otra herramienta como **xml2** para utilizar dicho árbol).
- Hacer una captura de pantalla.
- Obtener la versión de *Google Chrome*

1.8. Selección de potenciales paquetes a analizar

Finalmente, tras evaluar los paquetes expuestos y las ventajas y desventajas de los paquetes, se opta por elegir los paquetes **rvest**, **XML** y **SeleniumPipes** para profundizar en ellos, probarlos con más detalle, ver qué ofrecen más en detalle y hacer una cierta aproximación en la medida de su rendimiento.

Es necesario destacar previamente, que la no elección de un paquete particular, no indica que sea un paquete malo per se, si no que en comparación con sus competidores, es menos completo o bien no cumple con los requisitos que se esperan de él.

1.8.1. Paquetes elegidos que realizan análisis sin renderizado previo

Se han elegido dos paquetes del total de paquetes analizados en los que no se realiza un renderizado previo: **rvest** y **XML**. A continuación se expone para cada paquete, una explicación breve de las razones por la que no ha sido elegido (o sí, en caso de los elegidos), ya sea porque no cumplen determinados perfiles o bien porque pueden ser redundantes respecto a los candidatos elegidos (probablemente porque utilice a los candidatos como dependencia directa o inversa).

XML

Tal y como indica el autor en la web, en ciertos casos de uso en Windows hay pérdidas de memoria [18]. No se han encontrado referencias sobre como variar la compilación para que no aparezca este problema no tampoco consejos para evitarlo, lo que siembra dudas sobre su fiabilidad. Además, hay un manual de buenas prácticas para la gestión de memoria ya que al parecer XML no es del todo eficiente en su uso [18]. Es por estas razones por las que en principio se aconsejaría no utilizar este paquete.

Sin embargo la página de *OmegaHat* en la que se ha observado esta información data del año 2015 y el software también mientras que la última versión en *CRAN* [20] y *RDocumentation* es de es de Agosto de 2018 [21] por lo que posiblemente estos problemas hayan quedado resueltos. Además se puede observar que en las estadísticas de descarga de *RDocumentation* es un paquete muy descargado y además hay bastantes paquetes que lo tienen como dependencia siendo estas descargas indirectas como consecuencia de que otros paquetes dependan de él muy significativas (en el mes de agosto de 2018 aproximadamente la mitad de las descargas son por descarga de dependencias y los días de menos descargas globales rondaba en torno a las 1000 descargas y los que más se acerca a 5000 descargas diarias). Es por la cantidad de uso junto con la versatilidad que ofrece que es interesante probarlo, ya que es un paquete muy potente.

xml2

xml2 es un paquete mucho más simple que XML a tenor de lo que se ha podido observar. Sin embargo no se ha elegido porque hay otro paquete que tiene mejoras en manejo de formularios, sesiones o selectores CSS (que este paquete no incluye) que es *rvest*. Sin embargo, cuando se vea la arquitectura de *rvest*, se observará que indirectamente se utiliza *xml2* y este último paquete aporta algo más en materia de extracción de información de los documentos HTML y en funcionalidades de navegación (en detrimento cierto es de la parte de creación de documentos).

Es por estas razones por las que se ha optado por rechazar el uso de este paquete, para evitar redundancias.

rvest

Dado que **rvest** permite utilizar selectores CSS y XPath, encadenar las funciones y además se puede navegar entre páginas, enviar formularios, *parsear* tablas, etcétera; y estas funcionalidades no se han localizado todas juntas en el resto de los paquetes que no aportan renderizado previo, se ha decidido que este paquete sea uno de los candidatos. Además, se espera que el uso de tuberías agilice el desarrollo de los algoritmos para hacer *web scraping* que se requieran.

selectr

Existen varias razones por las que **selectr** no ha sido incluido como candidato en los paquetes a analizar. En primer lugar, las dos funciones que ofrece son reducidas de por sí: la transformación CSS-XPath no permite realizar por sí misma de ninguna manera *web scraping* mientras que la segunda, no permite más que hacer una búsqueda inicial en el árbol y si se desea profundizar o hacer nuevas búsquedas sobre el árbol se debe utilizar **XML** o **xml2**.

La otra razón principal es la de evitar redundancia a la hora de hacer los ejemplos, ya que otro de los candidatos (**rvest**) utiliza por debajo **selectr** y **xml2** proporcionando más funcionalidad.

scrapeR

EN el caso de este paquete, la determinación de no incluirlo viene dada por razones de redundancia, ya que simplemente crea el árbol DOM pero luego para operar a través de él (utilizando selectores *XPath* o *CSS*) es necesario el uso de la biblioteca **XML**, que ya ha sido incluida, por lo que se considera que las aportaciones adicionales que puede hacer esta biblioteca no son especialmente significativas.

Sería interesante no obstante, verificar las posibilidades que tiene a la hora de indicar varias URL's en una página, ya que podría ser que este paquete aportara en ese sentido una mejora de rendimiento.

rCrawler

Este paquete no ha sido elegido debido a que el propósito que tiene difiere un poco de lo que se desea realizar. La configuración que hay que hacer para descargar una sola página tiene bastante más complejidad, dado que el paquete está diseñado para realizar por defecto *web crawling* (habría que decirle que recorriera los enlaces de tal manera que solo se recorra el primer nivel de profundidad, contrariando así el concepto de realizar *web crawling* que provee esta herramienta). Además esta herramienta tiene montones de parámetros de configuración que desaconsejan utilizarla en los ejercicios propuestos, ya que los dificultaría mucho.

No obstante, dado su enfoque en el rendimiento (paralelización de consultas) y grado de refinamiento de la configuración sí que puede ser aconsejable probarlo en el caso de que se requiera aplicarlo a un problema resoluble mediante técnicas de *web crawling*.

htmltab

Este paquete ha sido descartado por su alto grado de especialización en extracción y "limpieza" de tablas, que no permitiría superar la mayor parte de los tests de supuestos prácticos preparados. Sin embargo sí que sería aconsejable el uso de este paquete si la extracción de tablas requiere un cierto tratamiento de limpieza o reordenación más allá de lo que se hará en el ejemplo del supuesto

práctico de extracción de una tabla expuesto para las herramientas en este trabajo.

XML2R

Como se indicaba anteriormente, este paquete va a ser rechazado debido a que el concepto de extracción y transformación que hay en este paquete no se ajusta al *web scraping* que se desea realizar.

boilerpipeR

En este caso, el paquete no se ajusta a las necesidades, ya que estriba en la elección de lo que se desea obtener en base a algoritmos heurísticos prestablecidos (lo que ata a lo que esos algoritmos permiten hacer) pero no permite extraer información determinada por un selector *XPath* o *CSS*.

1.8.2. Paquetes elegidos que realizan análisis con renderizado previo

A diferencia del apartado de análisis sin renderizado donde se han elegido dos paquetes, en este caso simplemente se ha elegido uno: **seleniumPipes**.

De igual manera, se va a proceder a explicar para cada paquete las razones que han llevado a ser rechazados (o en el caso de **seleniumPies** a ser elegido).

rdom

Este paquete se ha rechazado por razones de redundancia, ya que tras la fase de obtención y de renderizado, se debe utilizar un enfoque combinado con **XML** o **rvest** para poder trabajar más allá de hacer una consulta sobre el árbol con un solo selector.

No obstante, este paquete es aconsejable si ya se ha realizado el algoritmo requerido con alguna de estas utilidades y se ha observado que alguna de las páginas sobre las que se quiere hacer *web scraping* no funciona correctamente (porque para extraer la información necesaria se requiera una fase de renderizado para terminar de cargar el DOM con JavaScript).

relenium

Este paquete de origen español ha sido rechazado por la simple razón de que los autores indicaron que ya no mantendrían más el paquete y que aconsejaban utilizar la alternativa de **RSelenium** [46].

splashr

No se va a proceder a incluir a este paquete como candidato porque se requiere **XML** o **rvest** para poder acceder a selectores, lo que implicaría nuevamente ser redundante en los ejemplos y explicaciones.

No obstante, en su momento se determinó intentar probarlo ya que el autor indica que el enfoque es más ligero [47][48], pero no se logró concluir la instalación por problemas con el *Docker*.

RSelenium

Este paquete se planteó en su momento como uno de los candidatos potenciales, pero se ha descartado porque tiene un hermano **seleniumPipes** que realiza prácticamente lo mismo además de introducir una sintaxis en forma de tuberías (que a priori facilita enormemente la labor para realizar los algoritmos para extraer la información deseada).Ambos paquetes han sido desarrollados por John Harrison.

seleniumPipes

Este paquete ha sido el elegido como candidato con capacidades de renderizado previo, debido a las posibilidades que ofrece en torno a su funcionalidad y gracias a la posibilidad de operar con varios navegadores. Además, ha sido un gran punto a su favor el hecho de permitir notación mediante tuberías, ya que esto (a mi criterio) permite incrementar la legibilidad del código, su mantenibilidad y facilita el desarrollo de los algoritmos necesarios para extraer la información deseada.

webdriver

Este paquete ha sido rechazado debido a que es muy similar a **RSelenium**, aportando alguna característica menos. Además, otro de los factores relevantes ha sido que los autores indican que solamente ha sido probado para *PhantomJS*. Todos estos aspectos, añadidos al hecho de que el trabajo adquiría unas dimensiones significativas, han determinado que este componente no haya sido probado.

No obstante, en futuras líneas de trabajo o en ampliaciones de este trabajo sería interesante incluir este paquete.

decapitated

Se ha determinado que este paquete no es candidato a un análisis más profundo ya que solamente provee la fase de renderizado y creación del árbol, pero para realizar consultas sobre dicho árbol habría que utilizar otra herramienta.

Capítulo 2

Definición de tests de supuestos prácticos

2.1. Introducción

Antes de proceder a probar los paquetes elegidos en el capítulo anterior, se deben definir qué supuestos se van a probar, y para ello se crean una serie de tests.

Se ha creado un sitio web de ejemplo, que a través de diversas páginas pretende probar los siguientes tests:

- Propósito general
- Envío de formularios
- Gestión de cookies de sesión
- Renderizado de JavaScript
- Performance

Los tests se pueden encontrar en la web personal http://tfg.daniel-fl.com/.

A continuación se van a explicar simplemente los cuatro primeros supuestos, ya que el quinto se expondrá en un capítulo final destinado a comparara el rendimiento entre las diversas herramientas que se prueben, mientras que los cuatro primeros no tienen un objetivo comparativo, si no que dicho objetivo permite observar las características y funcionalidades que aporta cada herramienta aplicada a unos ejemplos dados.

2.2. Test de propósito general

La página para la ejecución de este test se puede visualizar accediendo con la herramienta de *web scraping* a través de la dirección http://tfg.daniel-fl.com/general/.

Este test tiene como propósito extraer contenido sencillo que permita probar las diversas características de selección y de recorrido del DOM que proveen las herramientas de *web scraping*. No pretende ser una sección que pruebe todas la posibilidades, si no que debe probar aquellas generales y que son comunes a toda o a casi cualquier) tarea o de *web scraping*. Adicionalmente, y con el fin de facilitar la primera aproximación al *web scraping*, no se hará especial hincapié en el rendimiento de las soluciones adoptadas.

Dicha página web tiene una barra de enlaces, una imagen, una tabla, enlaces junto a textos importantes, etcétera, con el objetivo de realizar los siguientes ejercicios previstos (en la medida en que se puedan hacer y que la herramienta de *web scraping* permita hacerlos):

1. Ejemplo de petición HTTP.
2. Obtener metaetiqueta del título de la página, descripción y palabras clave.
3. Obtener todos los enlaces de la barra de navegación.
4. Obtener enlaces por su clase y por su identificador.
5. Obtener los atributos de una imagen.
6. Obtener diversas etiquetas de una clase y diferenciarlas.
7. Obtener para cada enlace del cuerpo, a qué párrafo pertenece.
8. Ver los enlaces hermanos de los textos importantes.
9. Obtener la tabla que hay en el cuerpo.

A continuación se puede observar una captura de la página web que contiene la información a *scrapear* en este test. Dicha captura se desarrolla en las ilustraciones 2.1, 2.2 y 2.3.

Además, se observa en las ilustraciones 2.4 y 2.5 que la web es acorde a los estándares de la *World Wide Web Consortium* relativos a HTML5 y CSS3.

2.3. Test de envío de formularios

La página para la ejecución de este test se puede visualizar accediendo con la herramienta de *web scraping* a través de la dirección http://tfg.daniel-fl.com/forms/.

Este test tiene como objetivo verificar si la herramienta de *web scraping* puede gestionar envíos de formularios y recoger las respuestas que se generen en base a dichos envíos. Se va a solicitar un nombre a través del formulario. Al darle a enviar, se volverá a realizar una petición donde la página se mostrará de nuevo mostrando el nombre.

En la ilustración 2.6 se puede observar el formulario antes de enviar el nombre. Tras escribir el nombre y enviarlo, dicho nombre aparecerá en pantalla tal y como se muestra en la ilustración 2.7. A continuación se puede observar una

Ilustración 2.1: Página web destinada al test de propósito general (I)

Ilustración 2.2: Página web destinada al test de propósito general (II)

Ilustración 2.3: Página web destinada al test de propósito general (III)

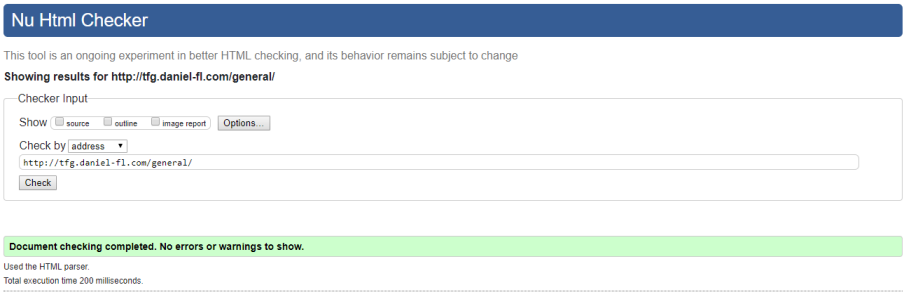


Ilustración 2.4: Resultado de validación HTML5 de la página web con los recursos necesarios para el test de propósito general

Ilustración 2.5: Resultado de validación CSS3 de la página web con los recursos necesarios para el test de propósito general

captura de la página web que contiene la información a *scrapear* en este test. Dicha captura se observa en la ilustración 2.6.

Ilustración 2.6: Página web destinada al test de envío de formularios (I)

Ilustración 2.7: Página web destinada al test de envío de formularios (II)

Si se hace una inspección del código fuente en tiempo real con las herramientas de desarrollador (se suele poder acceder a través de la opción "Inspeccionar elemento" haciendo clic en el botón derecho sobre el elemento que se quiere inspeccionar o bien mediante F12), se puede observar que se ha planificado la respuesta del servidor al introducir el nombre aislado dentro de un elemento

span de forma que dicho nombre se pueda extraer fácilmente. Para ello simplemente habrá que buscar con la herramienta de *web scraping* un elemento cuyo identificador sea *nombre*. Se puede ver en la ilustración 2.8 una captura detallada de las herramientas de desarrollador donde se puede observar el identificador utilizado en la etiqueta *span* que contiene el nombre.

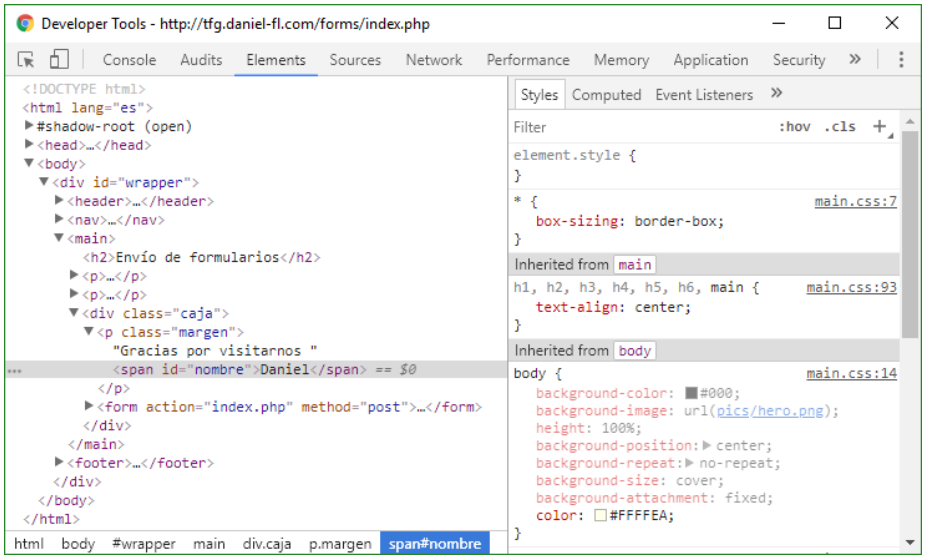


Ilustración 2.8: Ubicación del nombre en la respuesta del envío de formularios visto desde las herramientas de desarrollador

En resumen, simplemente habrá que verificar con la herramienta de *web scraping* si dicho elemento existe, y en caso afirmativo recuperarlo.

Además, se observa en las ilustraciones 2.9 y 2.10 que la web es acorde a los estándares de la *World Wide Web Consortium* relativos a HTML5 y CSS3.

2.4. Test de gestión de cookies

La página para la ejecución de este test se puede visualizar accediendo con la herramienta de *web scraping* a través de la dirección http://tfg.daniel-fl.com/cookies/.

El objetivo de este test es demostrar si la herramienta de *web scraping* es capaz de gestionar las *cookies* que se generan en el navegador, que permiten mantener un identificador de las sesiones que están alojadas en el servidor. En caso afirmativo, se observará que el siguiente número de visitas previas crecerá con cada petición. En caso negativo, siempre se observará que hay cero visitas previas.

La idea básica es que si dicho identificador de las sesiones no se mantiene

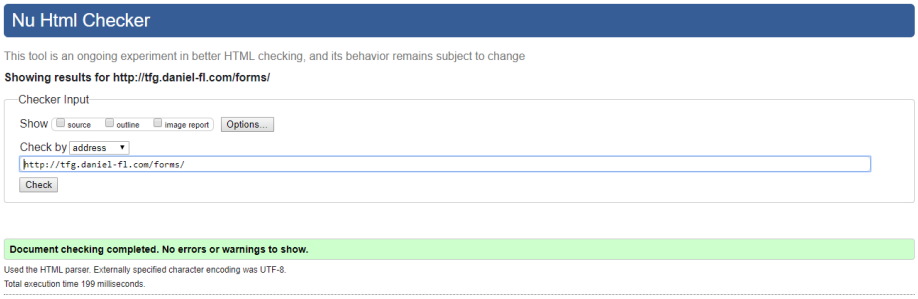


Ilustración 2.9: Resultado de validación HTML5 de la página web destinada al test de envío de formularios

Ilustración 2.10: Resultado de validación CSS3 de la página web destinada al test de envío de formularios

(esto es, la *cookie*), el servidor creará una sesión nueva cada vez que se acceda a la página. Hay herramientas de *web scraping* que permiten bien sea de forma transparente, o bien haciendo alguna configuración adicional, el mantenimiento de esta *cookie*, mientras que otras herramientas no lo permiten.

En la ilustración 2.11 se puede observar la página web. Si se revisita la página, se puede observar que el número de veces que aparece en la pantalla indicando que la web ha sido visitada se incrementa (ilustración 2.12).

Ilustración 2.11: Página web destinada al test de gestión de cookies (I)

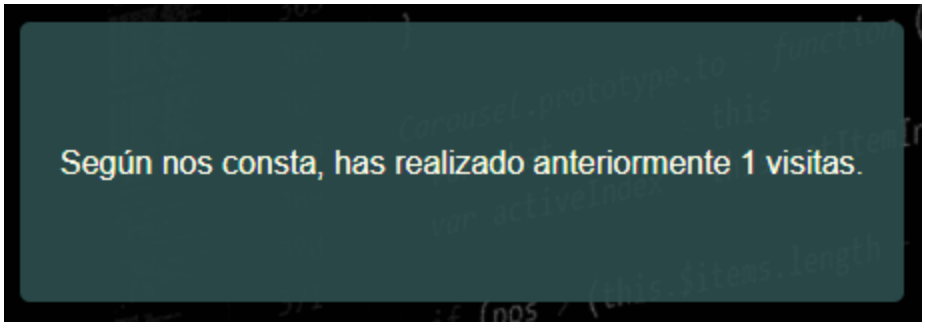


Ilustración 2.12: Página web destinada al test de gestión de cookies (II)

En la ilustración 2.13 se puede observar cómo el servidor PHP al inicializar una sesión, devuelve su identificador que el navegador almacena en forma de *cookie*. Esta información se ha obtenido con el complemento "Web Developer" de Google Chrome.

Si se hace una inspección del código fuente en tiempo real con las herramientas de desarrollador (se suele poder acceder a través de la opción "Inspeccionar

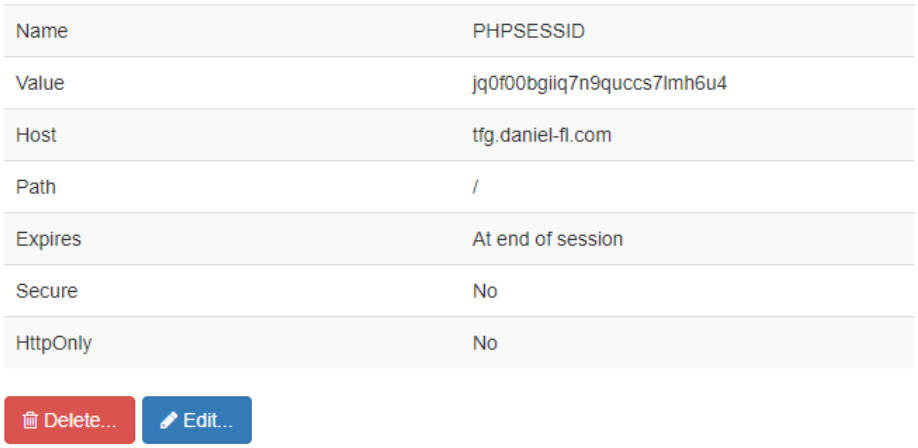


Ilustración 2.13: Visión de la *cookie* almacenando el identificador de sesión

elemento" haciendo clic en el botón derecho sobre el elemento que se quiere inspeccionar o bien mediante F12), se puede observar que se ha planificado la respuesta del servidor al introducir el número de visitas previas aislado dentro de un elemento *span* de forma que dicho número se pueda extraer fácilmente mediante un selector CSS (o XPath si así se requiriera). Para ello simplemente habrá que buscar con la herramienta de *web scraping* un elemento cuyo identificador sea *visitas*. Se puede ver en la ilustración 2.14 una captura detallada de las herramientas de desarrollador donde se puede observar el identificador utilizado en la etiqueta *span* que contiene el número de visitas.

Además, se observa en las ilustraciones 2.15 y 2.16 que la web es acorde a los estándares de la *World Wide Web Consortium* relativos a HTML5 y CSS3.

2.5. Test de renderizado de JavaScript

La página para la ejecución de este test se puede visualizar accediendo con la herramienta de *web scraping* a través de la dirección http://tfg.daniel-fl.com/test-js/.

El objetivo de este test es verificar si realmente la herramienta de *web scraping* renderiza la página web o solamente la descarga. Para ello se ha creado una página, que tiene una parte de la misma que se carga de forma asícrona (los profesionales), realizando una petición AJAX de un contenido en JSON y parseando el mismo, para finalmente insertarlo en el DOM mediante JQuery. En caso de que dicha herramienta renderice la página, no solamente se descargará la página si no que se ejecutará el código en JQuery que realiza la descarga de los profesionales en JSON (véase la ilustración 2.17 para ver el JSON que se descarga) y se insertará en el DOM tal y como está detallado en JavaScript.

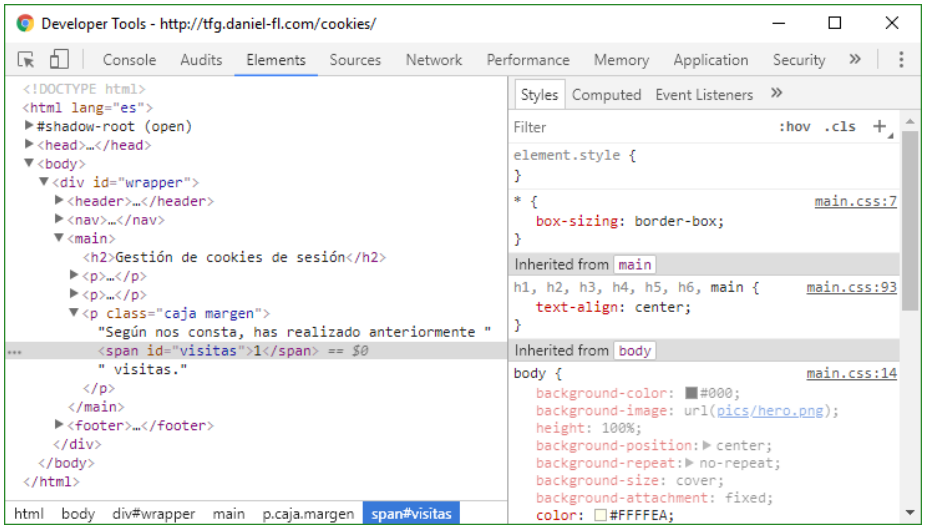


Ilustración 2.14: Ubicación del número de visitas en la respuesta al cargar la página web del test de gestión de *cookie*

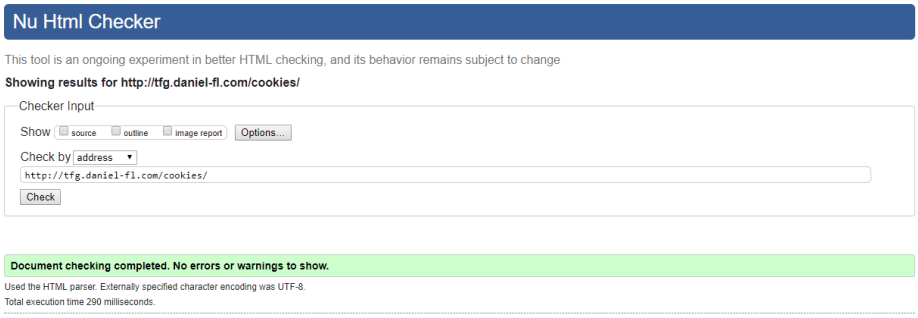


Ilustración 2.15: Resultado de validación HTML5 de la página web destinada al test de gestión de *cookies*

Ilustración 2.16: Resultado de validación CSS3 de la página web destinada al test de gestión de *cookies*

En caso de que solamente se permita hacer *web scraping* sobre la página decargada, sin ser esta renderizada previamente (esto es, no se ejecutan los scripts que aparecen en la misma), se obtendrá otra información diferente en la que no aparecerán los profesionales.

```
{"profesionales":[
    {"nombre":"Max Power",
     "edad":22,
     "avatar":"pics/person1.jpeg",
     "descripcion": "Persona con una gran formación pero poca experiencia. Puede aportar ideas muy
disruptivas. Graduado en ingeniería informática con máster en desarrollo de videojuegos."},
    {"nombre":"Emma Mc'Calister",
     "edad":26,
     "avatar":"pics/person2.jpeg",
     "descripcion":"A pesar de su juventud, es una de las científicas más prominentes del ámbito nacional,
con gran potencial en I+D. Graduada en matemáticas y física. Doctora en física de partículas."},
    {"nombre":"John Doe",
     "edad":36,
     "avatar":"pics/person3.jpeg",
     "descripcion":"Persona curtida en entornos altamente competitivos. Habiendo estudiado economía, con
especialización en la escuela de Chicago, la abandonó en pro de la teoría austriaca del ciclo económico."}
]
}
```

Ilustración 2.17: JSON con información a cargar dinámicamente

En la ilustración 2.18 se puede observar la página web renderizada correctamente por el navegador. Se puede observar que aparecen tres elementos que corresponden a tres personas (los profesionales), que son los que se cargan de forma dinámica en la página.

Con la ayuda del complemento "Web Developer" de Google Chrome, se va a proceder a proceder a emular el efecto que se observaría con una herramienta que no tenga un motor de navegación (y que por lo tanto, entre otras cosas no podría procesar JavaScript). El contenido que se observaría en dicho caso, sería el que se muestra en la ilustración 2.19, en el que se puede ver que no hay pista alguna de los profesionales apareciendo en su lugar una imagen que indica que la sección de profesionales se está cargando.

Si se hace una inspección del código fuente en tiempo real con las herramientas de desarrollador (se suele poder acceder a través de la opción "Inspeccionar

Ilustración 2.18: Página web destinada al test de renderizado de JavaScript con el renderizado activado

Ilustración 2.19: Página web destinada al test de renderizado de JavaScript con el renderizado desactivado

elemento" haciendo clic en el botón derecho sobre el elemento que se quiere inspeccionar o bien mediante F12), en ambas páginas, se puede observar que en el caso de la página que ha sido correctamente renderizada, dentro del elemento *div* (elemento de caja) con identificador *dinamico*, donde hay un elemento *ul* (lista de elementos) cuyo identificador es *profesionales*, que dentro contiene tres elementos. Estos tres elementos son los profesionales. Esta caja dinámica, siempre aparecerá tanto si la página se renderiza correctamente como si no. Los profesionales así como su contenedor *ul* solo aparecerán si el código JavaScript se ejecuta (lo que implica que la página se ha de renderizar). Si la página no es renderizada y no se ejecuta el código JavaScript, simplemente se mostrará un elemento *div* con identificador *cargando*, en cuyo interior hay una imagen GIF que indica que se está cargando el contenido (esto se suele hacer por si el navegador tarda mucho en ejecutar el código JavaScript, en caso de páginas que son muy grandes. Es obvio que si el navegador no hace el renderizado y no ejecuta el código, el mensaje de carga persistirá en pantalla). Estos detalles indicados se pueden observar en las ilustraciones 2.20 y 2.21.

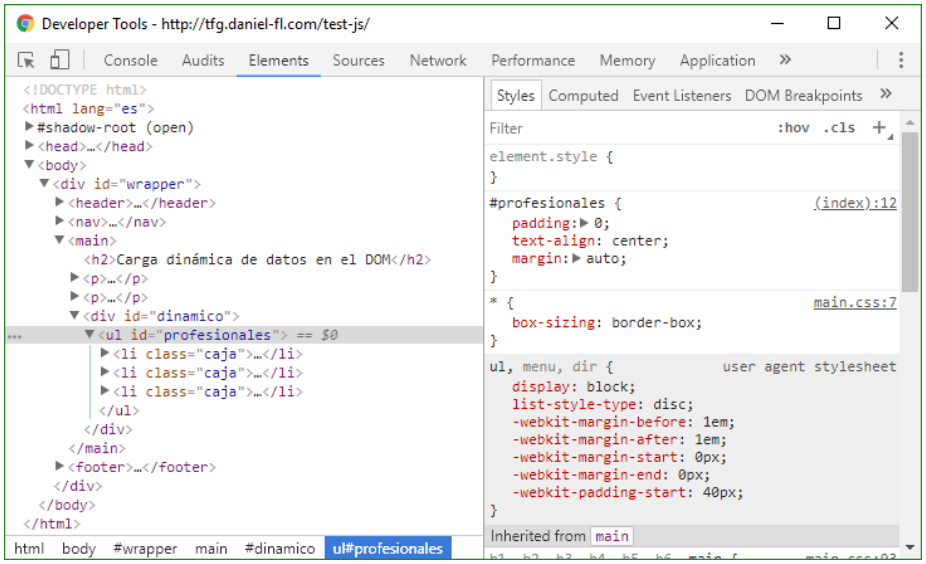


Ilustración 2.20: Código HTML en relación a la carga de profesionales en el contexto de ejecución de JavaScript habilitada

Además, se observa en las ilustraciones 2.22 y 2.23 que la web es acorde a los estándares de la *World Wide Web Consortium* relativos a HTML5 y CSS3.

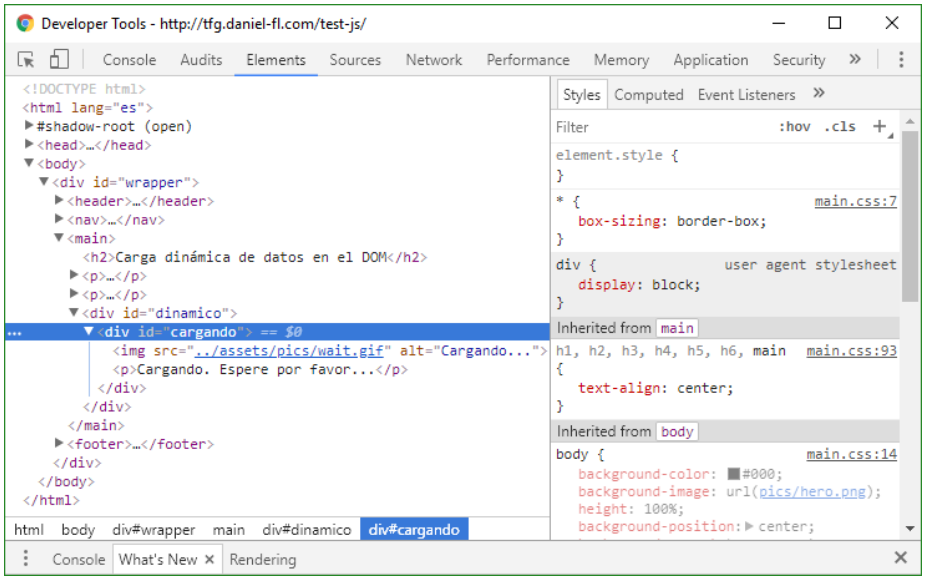


Ilustración 2.21: Código HTML en relación a la carga de profesionales en el contexto de ejecución de JavaScript deshabilitada

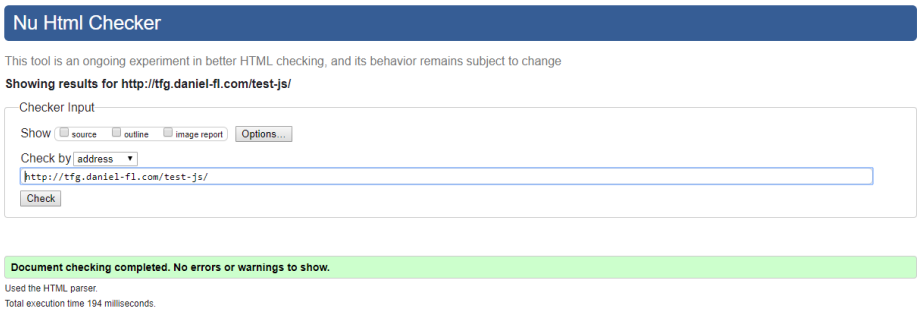


Ilustración 2.22: Resultado de validación HTML5 de la página web destinada al test de renderizado de JavaScript

Ilustración 2.23: Resultado de validación CSS3 de la página web destinada al test de renderizado de JavaScript

Capítulo 3

Web scraping con XML

3.1. Introducción a XML

XML es una biblioteca de R cuyo principal cometido es el de manipular documentos XML y HTML [21][20]. Es un proyecto iniciado por el profesor Duncan Temple Lee, de la universidad de California del campus Davis [21][20].

Se puede considerar que *XML* es un *wrapper* de **libxml2** [18][21][20], que es una biblioteca de C que permite básicamente crear y leer los documentos XML deseados.

Es un paquete bastante versátil y es uno de los primeros en realizar esta labor en R. Es ampliamente utilizado tal y como se puede observar en las referencias inversas y en las estadísticas de descarga [20][18] y tiene una documentación bastante profusa.

A la hora de desarrollar las funciones, se observarán las más esenciales para realizar *wbe scraping*, ya que hay gran cantidad de funciones de las que un alto grado sirve poco o nada para realizar esta labor (lo que no quiere decir que no tengan otra utilidad) y explicarlas todas implicaría un esfuerzo significativo orientado a un objetivo que no es el prefijado en este trabajo de fin de grado.

3.2. Visión general de las funcionalidades de XML

A continuación se procede a enunciar brevemente las diversas funcionalidades que posee el paquete [18]:

- *Parsear* ficheros XML, URLs y cadenas utilizando dos enfoques:
  - Enfoque basado en el árbol DOM.
  - Mecanismo SAX (Simple API for XML) dirigido por eventos.
- *Parsear* documentos XML-
- Realizar consultas *XPath* en un documento.

- Generar contenido XML para buffers, ficheros, URLs y árboles XML internos.
- Leer DTDs (definiciones de tipo de documentos).

En resumidas cuentas, XML permite tanto leer como escribir ficheros XML y HTML [58].

Si se desea conocer sobre las labores de *parsing* con otros ejemplos y de forma muy gráfica y simplificada (con documentos XML simplificados y mostrados en forma de árbol), aconsejo sustancialmente los apuntes de Gaston Sanchez ([58]), ya que explica este paquete de forma muy simplificada (no hace *web scraping* como tal, si no que utiliza ficheros XML) y además hace una introducción bastante amena del uso de *XPath*.

3.3. Instalación de XML

Para la ejecución de los ejercicios que se exponen a continuación, se asume que se dispone del entorno de ejecución para R instalado. Si no se dispusiera de él, simplemente habría que descargárselo desde alguno de los espejos existentes como [59].

Para instalar dicho paquete simplemente se debe abrir la consola de R e introducir el comando de instalación que se muestra en el código fuente 1.

```
1 install.packages("XML")
```

Código fuente 1: Instalación del paquete *XML*

3.4. Detalle pormenorizado de las funciones de XML

Se observa verificando el manual de referencia del paquete y las webs recopilatorias que tiene decenas de funciones [21], lo que es indicador de la gran versatilidad que tiene. Una posible justificación al número de funciones estriba en que este paquete no solamente permite *parsear* documentos XML y HTML, si no que permite manipularlos y crearlos de cero [21][20].

Debido a la enorme cantidad de funciones que hay, simplemente se expondrán las más significativas para la extracción de información de un documento HTML. Además, es probable que no se expongan todos los parámetros en algunas de las funciones, debido a la gran *aridad* de las mismas, en donde la gran mayoría de los parámetros son opcionales.

3.4.1. Obtención de un documento HTML/XML

En primer lugar, se debe obtener el documento HTML del que se desea extraer cierta información. Para realizar este cometido se debe utilizar la función `htmlParse()` [60]. Se pueden observar los argumentos a continuación junto al uso de la función indicada en el código fuente 2.

```
1 htmlParse(file, ignoreBlanks = TRUE, handlers = NULL,
  ↪  replaceEntities = FALSE, asText = FALSE, trim = TRUE,
  ↪  validate = FALSE, getDTD = TRUE, isURL = FALSE, asTree =
  ↪  FALSE, addAttributeNamespaces = FALSE, useInternalNodes =
  ↪  TRUE, isSchema = FALSE, fullNamespaceInfo = FALSE, encoding
  ↪  = character(), useDotNames = length(grep("^\\.",
  ↪  names(handlers))) > 0, xinclude = TRUE, addFinalizer = TRUE,
  ↪  error = htmlErrorHandler, isHTML = TRUE, options =
  ↪  integer(), parentFirst = FALSE)
```

Código fuente 2: Función de obtención de un documento HTML con *XML*

Argumentos principales

Debido a que la función tiene una gran cantidad de argumentos opcionales que probablemente no se utilicen nunca, se deja al lector la posibilidad de consultarlos todos si requiere un uso más refinado de la función [60] [60]. Se van a explicar los argumentos más usuales así como todos los obligatorios.

Retorno

Si no se ha modificado el parámetro `useInternalNodes` y se mantiene el valor por defecto (que es cierto), se obtiene un objeto de clase `XMLInternalDocument` que se podrá utilizar posteriormente. En caso de que se haya variado algún parámetro opcional es aconsejable ver la documentación para contrastar la compatibilidad con las posteriores funciones [60].

3.4.2. Extracción de nodos de un documento

Tras obtener el documento HTML deseado, se tendrán que obtener los nodos que se desean. Para ello se debe utilizar la función `getNodeSet()` [62]. Se pueden observar los argumentos a continuación junto al uso de la función indicada en el código fuente 3.

`file` Nombre del fichero que contiene el documento HTML. Si se indica `\~` quiere decir que se debe indicar la carpeta principal del usuario. También puede ser una URL (este será el uso que se le dé en este documento). El fichero utilizado podría estar comprimido (compresión *gzip*) y ser leído directamente por el *parser* sin necesidad de hacer una descompresión previa. Este parámetro es obligatorio.

`asText` Valor lógico que indica si el argumento `file` debe ser tratado directamente como el contenido del documento HTML en vez del nombre de un fichero. Esto permite obtener el documento de otras fuentes diferentes (dirección web, *sockets*, procedimientos remotos, servicios web, etcétera.) y que el *parser* aún pueda ser utilizado. Este parámetro es opcional.

`trim` Valor booleano que permite determinare si se suprimen los caracteres en blanco que puedan haber al principio y al final de las cadenas de texto. Este argumento es opcional.

`error` Función opcional que puede ser invocada si el *parser* genera un error durante su ejecución. Esta función es llamada con 7 argumentos, en caso de que sea utilizada (en los ejemplos realizados no se utiliza la gestión de errores para simplificar los ejemplos) se debe consultar con más detalle el uso de este parámetro concreto [61].

```
1 getNodeSet(doc, path, namespaces = xmlNamespaceDefinitions(doc,
  ↪  simplify = TRUE), fun = NULL, sessionEncoding = CE_NATIVE,
  ↪  addFinalizer = NA, ...)
```

Código fuente 3: Función de obtención de nodos de un documento HTML con *XML*

Argumentos principales

Debido a que la función tiene una gran cantidad de argumentos opcionales que probablemente no se utilicen nunca, se deja al lector la posibilidad de consultarlos todos si requiere un uso más refinado de la función 3 [62]. Se van a explicar los argumentos más usuales así como todos los obligatorios.

`doc`	Un objeto de clase XMLInternalDocument. Esto se puede obtener tras *parsear* el documento.
`path`	Una cadena que contenga la expresión *XPath* que se desea evaluar, esto es, esta cadena es la que permite determinar el selector *XPath* que determina el criterio para obtener los nodos deseados.
`fun`	Un objeto de función que permite ser ejecutada para cada nodo del conjunto de nodos extraído, una vez que la extracción ha finalizado. El nodo se pasa en el primer parámetro de la función. Este argumento es opcional.
`...`	Argumentos adicionales para pasar a *fun* en cada nodo. Este parámetro es opcional.
`sessionEncoding`	Funcionalidad y parámetros experimentales relativos a la codificación. Este parámetro es opcional.

Retorno

En función del tipo de dato que genera la expresión *XPath*, se puede devolver un tipo u otro de datos, tal y como se observa a continuación 3:

- **lista:** Si la expresión genera una lista, el resultado devolverá un set de nodos.
- **número:** Si la expresión genera un número, este será devuelto como `numeric`.
- **valor lógico:** Si la expresión genera un valor lógico, la función devolverá un valor booleano.
- **cadena de caracteres:** Se retornará una cadena de caracteres.

Además, si el parámetro fun es proporcionado a la función y el resultado de la consulta *XPath* es un conjunto de nodos, el resultado proporcionado por R será una lista.

3.4.3. Obtención del contenido de un nodo

Para obtener el valor o contenido de un nodo hay que utilizar la función `xmlValue()` [63]. La interfaz de esta función se puede observar en el código fuente 4 junto a sus argumentos.

```
1 xmlValue(x)
```

Código fuente 4: Función de obtención del valor de un nodo de un documento HTML con *XML*

Argumentos

`x` Un nodo XML (objeto de clase XMLNode) que contiene el contenido que se desea obtener.

Retorno

El resultado de la ejecución de esta función es el valor del nodo (generalmente una cadena). En concreto lo que se obtiene es el atributo value del objeto de tipo XMLNode recibido. Este valor generalmente suele el que está comprendido entre la apertura y el cierre de la etiqueta que representa el nodo.

3.4.4. Obtención del valor de un atributo de un nodo

Para obtener el valor de un atributo concreto de un nodo concreto se debe utilizar la función `xmlGetAttr()` [64]. La interfaz de esta función se puede observar en el código fuente 5 junto a sus argumentos.

```
1 xmlGetAttr(node, name, default = NULL, converter = NULL,
  ↪  namespaceDefinition = character(), addNamespace =
  ↪  length(grep(":", name)) > 0)
```

Código fuente 5: Función de obtención del valor del atributo de un nodo de un documento HTML con *XML*

Argumentos principales

Debido a que la función tiene una gran cantidad de argumentos opcionales que probablemente no se utilicen nunca, se deja al lector la posibilidad de consultarlos todos si requiere un uso más refinado de la función 5 [64]. Se van a explicar los argumentos más usuales así como todos los obligatorios.

Retorno

Si el atributo indicado está presente en el nodo, el valor retornado será la cadena asignada a dicho atributo (el valor). Por el contrario, si no está presente, se retornará el valor indicado en el parámetro `default`.

3.4.5. Obtención de todos los atributos

Para obtener el valor de todos los atributos de un nodo concreto se debe utilizar la función `xmlAttrs()` [65]. La interfaz de esta función se puede observar en el código fuente 6 junto a sus argumentos.

`node`	Un nodo XML (objeto de clase XMLNode que contiene el atributo se desea obtener.
`name`	Nombre del atributo que se desea obtener.
`default`	Valor para devolver por defecto si el atributo no está presente en el nodo XML. Este parámetro es opcional.
`converter`	Función que si se proporciona es invocada con el valor del atributo y el valor del nodo. Se puede utilizar para convertir el valor del atributo a otro tipo de valor, como puede ser un número. Solamente se invoca si el atributo existe en el nodo. Si no, esta función no es aplicada al valor por defecto.

```
1 xmlAttrs(node, ...)
```

Código fuente 6: Función de obtención de todos los atributos de un nodo de un documento HTML con *XML*

Argumentos

`node`	Un nodo XML (objeto de clase XMLNode que contiene los atributos que se desean obtener.
`...`	Parámetros opcionales adicionales avanzados, que no se utilizarán en los casos de documentos HTML. Sirve para gestionar los *namespace*, indicando si se desean obtener a través de un parámetro o si se desea que haya prefijación en el nombre de los atributos. Estos detalles son más avanzados y se utilizan en el caso de documentos XML. En la página de la función se detalla más este parámetro.

Retorno

Se obtiene un vector asociativo de caracteres donde los nombres son cada uno de los nombres de los atributos y los elementos corresponden al valor de cada uno de los atributos. De esta forma se pueden representar como pares de elementos clave-valor.

3.4.6. Obtención del nombre de un elemento

A veces puede ser necesario identificar cual es el nombre de un elemento/nodo/etiqueta dado (por ejemplo, si se obtienen distintos elementos a través de una clase pero se quiere discriminar una acción a realizar en función del tipo de elemento). Para realizar esta función se debe utilizar la función `xmlName()`

[66]. La interfaz de esta función se puede observar en el código fuente 7 junto a sus argumentos.

```
1 xmlName(node, full = FALSE)
```

Código fuente 7: Función de obtención del nombre de un nodo de un documento HTML con *XML*

Argumentos

A continuación se pueden observar los argumentos requeridos para esta función:

- `node` Un nodo XML (objeto de clase XMLNode cuyo nombre de etiqueta se desea obtener.
- `full` Valor lógico que indica si se debe indicar en el nombre del elemento el prefijo asociado a su *namespace*. El valor cierto indica que sí se debe realizar esta acción. Esta configuración es de utilidad en los documentos XML pero no en los documentos HTML (ya que no tienen *namespace*.

Retorno

Se obtiene una cadena con el nombre de la etiqueta del nodo indicado por parámetro.

3.4.7. Obtención de una tabla

Una de las operación más complejas que hay a la hora de hacer *web scraping* es obtener una tabla. Para este cometido, *XML* provee de una función bastante versátil denominada `readHTMLTable`.

Con esta función se pueden leer todas las tablas de un documento localizado en el equipo o en la red, o bien utilizar un documento previamente parseado con `htmlParse()`. Adicionalmente se puede indicar un solo nodo donde esté la tabla que se desea *parsear*.

interfaz de esta función se puede observar en el código fuente 8 junto a sus argumentos.

```
readHTMLTable(doc, header = NA, colClasses = NULL, skip.rows =
↪  integer(), trim = TRUE,elFun = xmlValue, as.data.frame =
↪  TRUE, which = integer(), ...)
```

Código fuente 8: Función de obtención de tablas con *XML*

Argumentos

A continuación se pueden observar los distintos parámetros que admite esta función:

`doc`	El documento HTML, pudiendo ser un nombre de fichero, una dirección URL, un documento previamente *parseado* o una cadena de caracteres que contenga el contenido a *parsear* y procesar.
`header`	Puede ser un valor booleano que indique que la tabla tiene nombres de columna (bien sea en la primera fila o en una fila thead) o bien puede ser un vector de caracteres indicando los nombres deseados para las columnas. Este parámetro es opcional.
`colClasses`	Lista o un vector que permite proporcionar los nombres de los tipos de datos para las diferentes columnas en la tabla o, alternativamente, una función utilizada para convertir los valores de cadena al tipo apropiado. Un valor de NULL significa que se debe descartar esa columna del resultado. Este parámetro es opcional y se puede observar algunos detalles adicionales en [67]
`skip.rows`	Vector de enteros donde s epuede indicar qué filas se desean descartar. Este parámetro es opcional.
`trim`	Valor booleano donde se indica si se desean borrar los caracteres en blanco sobrantes a la izquierda y a la derecha de cada celda. Este parámetro es opcional.
`elFun`	Una función donde se puede transformar el valor de cada celda. Actualmente solamente se pasa a dicha función el nodo, aunque en un futuro es posible que se proporcione más información (fila, columna) que permita tener algo más de información sobre el contexto en el que se ubica la celda. Este parámetro es opcional.
`as.data.frame`	Valor booleano que indica si se devuelve la tabla como *data frame* (TRUE) o como matriz (FALSE). Este parámetro es opcional.
`which`	Vector de enteros que permite determinar de las tablas que se han encontrado en el documento cuáles se desean obtener. Esto solamente aplica en el caso de que se haya intentado *parsear* un documento XML, no para tablas individuales. Este parámetro es opcional.
`...`	Paramétros adicionales que habría que proporcionar si se indicara `as.data.frame = TRUE`.

Retorno

Si se proporciona un documento HTML completo, el valor obtenido sería una lista de *data frames* o de matrices. Si solo se proporciona un nodo tabla concreto, entonces no se percibirá una lista, si no solamente el *data frame* o matriz concreto.

3.4.8. Miscelánea: otras funciones

Como se indicó al principio de esta sección, este paquete tiene un montón de funcionalidades que permiten manipular los documentos XML y HTML y hacer todo tipo de operaciones. Una gran parte de esas operaciones no sirven para hacer *web scraping* o bien aunque sirven, no suelen ser necesarias (hay funciones para desplazarse a través del árbol DOM que no son estrictamente necesarias, ya que se puede navegar a través del árbol con *XPath*, otras funciones para extraer enlaces, etcétera). A continuación se expone un listado con las funciones y clases que están en distintos epígrafes en la documentación de *XML* en CRAN (y que se muestran de forma diferenciada en *RDocumentation*) [20][21]. Esto no quiere decir que dentro de cada apartado no se expliquen más funciones, por lo que se deja al usuario la labor de buscar de forma más concreta en el manual si se requiere alguna funcionalidad muy específica (el manual en la versión 3.98-1.16 tiene 170 páginas, por lo que sería inabarcable explicarlo todo en este trabajo de fin de grado).

- `addChildren`: Añade nodos hijo a un nodo XML.
- `addNode`: Añade un nodo al árbol.
- `append.xmlNode`: Añade un nodo al árbol.
- `asXMLNode`: Convierte objetos de nodo no-XML a objetos `XMLTextNode`.
- `asXMLTreeNode`: Convierte un nodo XML regular en uno para ser utilizado en un árbol "plano".
- `catalogLoad`: Manipular contenidos del catálogo XML.
- `catalogResolve`: Buscar un elemento a través del mecanismo de catálogo XML.
- `coerceNodes`: Transforma entre representaciones XML.
- `compareXMLDocs`: Indica las diferencias entre dos documentos XML.
- `docName`: Accesores para el nombre del documento XML.
- `Doctype`: Constructor para la referencia DTD (DOCTYPE).
- `Doctype-class`: Clase para describir una referencia a un DTD de un XML.
- `dtdElement`: Obtiene la definición de un elemento o entidad de un DTD.

- `dtdElementValidEntry`: Determina si un elemento XML permite un tipo particular de subelemento.
- `dtdIsAttribute`: Consulta si un nombre es un atributo válido de un elemento DTD.
- `dtdValidElement`: Determina si una etiqueta XML es válida dentro de otra.
- `ensureNamespace`: Asegura que el nodo tenga una definición para *namespaces* de XML particulares.
- `findXInclude`: Encuentra el nodo XInclude asociado con un nodo XML.
- `free`: Libera el objeto especificado y limpia la región de memoria ocupada por el mismo.
- `genericSAXHandlers`: Lista de manejadores de llamadas genéricas SAX.
- `getChildrenStrings`: Concatena todos los textos de sus hijos y los descendientes de los mismos.
- `getEncoding`: Determina la codificación de un documento XML o un nodo.
- `getHTMLLinks`: Obtiene los enlaces o los nombre de ficheros externos en un documento HTML.
- `getLineNumber`: Determina el número de línea del fichero en el que se encuentra el nodo XML (interno).
- `getRelativeURL`: Determina el nombre de una URL relativa en base a una URL base.
- `getSibling`: Manipula nodos XML hermanos.
- `getXIncludes`: Encuentra los documentos que son "XInclude" en un documento XML.
- `getXMLErrors`: Obtiene los errores de *parseado* de un documento XML o HTML.
- `isXMLString`: Utilidades para trabajar con cadena XML.
- `length.XMLNode`: Determina el nímero de hijos en un objeto XMLNode.
- `libxmlVersion`: Consulta la versión y las características disponibles de la biblioteca subyacente *libxml*.
- `makeClassTemplate`: Crea una definición de clase S4 basada en nodos XML.
- `names.XMLNode`: Obtiene los nombre de un hijo de nodos XML.

- `newXMLDoc`: Crea un nodo XML interno o un objeto de documento.
- `newXMLNamespace`: Añade la definición de un *namescape* a un nodo XML.
- `parseDTD`: Lee una definición de tipo de documento (DTD).
- `parseURI`: *Parsea* una cadena URI en sus elementos.
- `parseXMLAndAdd`: *Parsea* contenido XML y lo añade a un nodo.
- `print.XMLAttributeDef`: Métodos para mostrar objetos XML.
- `processXInclude`: Realiza las sustituciones XInclude.
- `readHTMLList`: Lee datos de una lista HTML o de todas las listas de un documento.
- `readKeyValueDB`: Leer un documento de estilo de lista de propiedades XML.
- `readSolrDoc`: Lee los datos de un documetno *Solr*.
- `removeXMLNamespaces`: Elimina las definiciones de *namespace* de un nodo XML o un documento.
- `saveXML`: Salida del árbol interno XML.
- `SAXState-class`: Una clase base virtual definiendo métodos para el *parseo SAX*.
- `schema-class`: Clases para trabajar con esquemas XML.
- `setXMLNamespace`: Establece el *namespace* de un nodo.
- `startElement.SAX`: Métodos genéricos para las llamadas SAX.
- `supportsExpat`: Determina si los *parsers* nativos de XML están siendo utilizados.
- `toHTML`: Crea una representación del objeto de R obtenido, utilizando nodos internos creados en C de forma subyacente.
- `toString.XMLNode`: Crea una representación en cadena de caracteres de un nodo XML.
- `xmlApply`: Aplica una función a cada uno de los hijos de un nodo XMLNode.
- `XMLAttributes-class`: Clase XMLAttributes.
- `xmlAttributeType`: El tipo de un atributo XML para un elemento del DTD.

- `xmlChildren`: Obtiene los subnodos dentro del objeto XMLNode proporcionado.
- `xmlCleanNamespaces`: Elimina *namespaces* redundantes de un documento XML.
- `xmlClone`: Hace una copia de un nodo o un documento XML (ambos internos).
- `XMLCodeFile-class`: Clases simples para la identificación de un docuemto XML que contiene código en R.
- `xmlContainsEntity`: Verifica si una entidad está definida dentro de un DTD.
- `xmlDOMApply`: Aplica la función determinada a los nodos en un árbol XML o un DOM.
- `xmlElementsByTagName`: Obtiene los hijos de un nodo XML con un nombre de etiqueta específico.
- `xmlElementSummary`: Tabla de frecuencias de nombres de elementos y atributos en el contenido XML.
- `xmlEventHandler`: Manejadores predeterminados para el analizador XML de eventos de estilo SAX.
- `xmlFlatListTree`: Constructores para árboles guardados como listas planas de nodos con información sobre padres e hijos.
- `xmlHandler`: Ejemplo de funciones manejadoras del *parser* de eventos XML.
- `XMLInternalDocument-class`: Clase para representar referencias a estructuras de datos a nivel de C para un documento XML.
- `xmlNamespace`: Permite obtener el *namespace* de un nodo XML indicado.
- `xmlNamespaceDefinitions`: Permite obtener las definiciones de cualquiera de los *namespaces* definidos en el nodo XML indicado.
- `xmlNode`: Crea un nodo XML.
- `XMLNode-class`: Clases para describir un objeto de nodo XML.
- `xmlOutputBuffer`: *Streams* de salida de XML.
- `xmlParent`: Obtiene el padre de un nodo XMLInternalNode o los nodos antecesores.
- `xmlParseDoc`: *Parsea* un documento XML con opciones controlando dicho *parser*.

- `xmlParserContextFunction`: Identifica una función como se requiere en un argumento *xmlParserContext*.
- `xmlRoot`: Obtiene el nodo XML más alto (el nodo raíz).
- `xmlSchemaValidate`: Valida un documento XML en relación a un esquema XML.
- `xmlSearchNs`: Encuentra un objeto de definición de *namespace* buscando en nodos antecesores.
- `xmlSerializeHook`: Funciones que ayudan a *serializar* y *deserializar* objetos XML internos.
- `xmlSize`: Permite obtener el número de subelementos dentro de un nodo XML.
- `xmlSource`: Permite ejecutar código en R, ejemplos, etcétera; desde un documento XML.
- `xmlStopParser`: Detener el *parser* XML.
- `xmlStructuredStop`: Funciones de manejo de condiciones de error para el *parseado* de XML.
- `xmlToDataFrame`: Extrae datos de un documento XML simple a *data frame*.
- `xmlToList`: Convierte un nodo o documento XML a una lista de R.
- `xmlToS4`: Mecanismo general para mapear un nodo XML a un objeto *S4*.
- `xmlTree`: Objeto DOM interno y actualizable para construir árboles XML.

3.5. Realización del test de propósito general con XML

3.5.1. Ejemplo de petición HTTP

En primer lugar se tiene que descargar la página web que se desea (la de propósito general) y *parsearla*, esto es, pasarla a una estructura en forma de árbol (lo que se conoce como DOM) que permita acceder de forma selectiva a los nodos que se deseen.

Para realizar esta acción se debe utilizar la función `htmlParse()` [60] indicando la URL deseada. Posteriormente se imprime el resultado de dicha función por pantalla.

El código fuente completo se puede observar en el código fuente 9.

Al aplicar la función de impresión sobre la estructura devuelta, se observa en la ilustración 3.1 que se obtiene en forma texto el código HTML correspondiente a la página. También se puede observar que el título de la página está

```r
library(XML)

pagina <- htmlParse("http://tfg.daniel-fl.com/general/")

print(pagina)
```

Código fuente 9: Ejercicio de petición HTTP con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> print(pagina)
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Tests de propÃ³sito general | Daniel Francisco LÃ³pez</title>
<meta name="description" content="Tests para hacer diversas pruebas de selecto$
<meta name="keywords" content="web, scrapping, utilidad, CSS, XPath, DOM">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link href="../assets/main.css" rel="stylesheet">
<style>
```

Ilustración 3.1: Resultado de ejecución del ejercicio de petición HTTP con *XML*

mal codificado al llevar tilde. Posteriormente se observará como esto no sucede normalmente cuando se utilizan selectores y se obtiene el valor de los nodos extraídos.

3.5.2. Obtener metaetiqueta del título de la página, descripción y palabras clave

Para realizar este ejercicio, se va a partir del ejercicio anterior, en el que se obtenía la página y se generaba la estructura de datos en forma de árbol.

El objetivo es obtener el elemento *title* de la página así como las metaetiquetas de descripción y de palabras clave.

La obtención de todos estos elementos parte de una extracción inicial de nodos mediante la función `getNodeSet()` [62], en la que partiendo de la página obtenida y utilizando el selector *XPath* correspondiente se pueden obtener de 1 a n nodos que se podrán tratar posteriormente. Tanto en el caso del título como en el caso de la metadescripción y las palabras clave, se ha utilizado un selector *XPath* directo, que va desde la raíz hasta el nodo deseado. Esto resulta en una gran eficiencia a la hora de realizar la búsqueda, ya que el intérprete no ha de realizar la búsqueda en todo el árbol.

Posteriormente, en el caso del título, ya que lo que se desea es el contenido de su etiqueta (también se podría denominar *valor*) se utiliza la función

`xmlValue()` [63] que realiza dicho cometido. Esta función se aplica sobre el primer y único nodo de la lista obtenida al realizar la selección.

Para el caso de las metaetiquetas el contenido que es de interés se encuentra en el atributo *content*. Es por ello que tras seleccionarlas (seleccionando la etiqueta meta que tiene como nombre el que corresponde a palabras clave o a descripción) la función que se debe utilizar es `xmlGetAttr()` [64] recibiendo el nodo el nodo del que se desea extraer el atributo concreto y el nombre de dicho atributo.

El código fuente completo se puede observar en el código fuente 10. Un extracto del resultado de dicha ejecución donde se puede contrastar lo explicado anteriormente se puede observar en la ilustración 3.2.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5  #Se imprime el titulo
6  tituloNodo <- getNodeSet(pagina, "/html/head/title")[[1]]
7  titulo <- xmlValue(tituloNodo)
8  print(titulo)
9
10 #Se imprime la descripcion:
11 descrNodo <- getNodeSet(pagina,
   ↪  "/html/head/meta[@name='description']")[[1]]
12 descr <- xmlGetAttr(descrNodo, "content")
13 print(descr)
14
15 #Se imprimen las palabras clave:
16
17 kwNodo <- getNodeSet(pagina,
   ↪  "/html/head/meta[@name='keywords']")[[1]]
18 kw <- strsplit(xmlGetAttr(kwNodo, "content"), ', ')
19 print(kw)
```

Código fuente 10: Ejercicio de obtención de metaetiquetas con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se imprime el titulo
> tituloNodo <- getNodeSet(pagina, "/html/head/title")[[1]]
> titulo <- xmlValue(tituloNodo)
> print(titulo)
[1] "Tests de propósito general | Daniel Francisco López"
>
> #Se imprime la descripcion:
> descrNodo <- getNodeSet(pagina, "/html/head/meta[@name='description']"$
> descr <- xmlGetAttr(descrNodo, "content")
> print(descr)
[1] "Tests para hacer diversas pruebas de selectores, para ver las posib$
>
> #Se imprimen las palabras clave:
>
> kwNodo <- getNodeSet(pagina, "/html/head/meta[@name='keywords']")[[1]]
> kw <- strsplit(xmlGetAttr(kwNodo, "content"), ', ')
> print(kw)
[[1]]
[1] "web"       "scrapping" "utilidad"  "CSS"       "XPath"
[6] "DOM"
```

Ilustración 3.2: Resultado de ejecución del ejercicio de obtención de metaetiquetas con *XML*

3.5.3. Obtener todos los enlaces de la barra de navegación

Para continuar se obtienen todos los enlaces que hay en la barra de navegación. Para ello se va volver a utilizar la función `getNodeSet()` [62]. En este caso se ha utilizado un selector que no parte de la raíz. Esto se hace así ya que aunque hay algo menos de eficiencia, se logra una sintaxis más sencilla ya que se omite la ruta para ir a nav.

A continuación se crean dos listas para acabar formando un *data frame*. La primera lista contendrá el texto de los enlaces, para lo que se utilizará nuevamente la función `xmlValue()` [63] aplicada con `sapply()` a cada uno de los elementos de la lista de nodos encontrados.

Por otra parte se hace la misma operación pero para obtener los enlaces con la función `xmlGetAttr()` [64] que permite obtener el atributo que contiene el enlace.

Por último se crea un *data frame* y se visualiza en pantalla con la función `View()`.

El código fuente completo se puede observar en el código fuente 11. Un extracto del resultado de dicha ejecución donde se puede contrastar lo explicado anteriormente se puede observar en la ilustración 3.3 y la tabla generada en la ilustración 3.4.

```r
library(XML)

pagina <- htmlParse("http://tfg.daniel-fl.com/general/")

#Se obtienen los enlaces
links <- getNodeSet(pagina, "//nav//a")

textos <- sapply(links, function(x){xmlValue(x)})
enlaces <- sapply(links, function(x){xmlGetAttr(x,"href")})

tabla <- data.frame(textos, enlaces)
View(tabla)
```

Código fuente 11: Ejercicio de obtención de enlaces de navegación con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces
> links <- getNodeSet(pagina, "//nav//a")
>
> textos <- sapply(links, function(x){xmlValue(x)})
> enlaces <- sapply(links, function(x){xmlGetAttr(x,"href")})
>
> tabla <- data.frame(textos, enlaces)
> View(tabla)
>
```

Ilustración 3.3: Resultado de ejecución del ejercicio de obtención de enlaces de navegación con *XML*

	textos	enlaces
1	Propósito general	#
2	Envío de formularios	../forms
3	Gestión de cookies de sesión	../cookies
4	Renderizado de JavaScript	../test-js
5	Performance	../performance

Ilustración 3.4: Tabla de la ejecución del ejercicio de obtención de enlaces de navegación con *XML*

3.5.4. Obtener enlaces por su clase y por su identificador

El objetivo de este ejercicio es buscar enlaces utilizando como selectores identificadores o clases con *XML*. Para ello se generan dos códigos fuente.

Ambos apartados empiezan haciendo uso de la función `getNodeSet()` [62]. Simplemente se varía el selector *XPath* para obtener los nodos según los identificadores o las clases deseadas. Es importante destacar que *XPath* no dispone de un selector natural para elegir clases, por lo que para verificar una clase, se debe poner el atributo entre espacios y ver si la cadena contiene una subcadena formada por la clase deseada utilizando espacio al principio y al final. La utilidad de esto es porque dentro del mismo atributo puede haber varias clases, por lo que no serviría la comparación mediante el igual (que verificaría si ambas cadenas son estrictamente iguales).

Posteriormente el proceso es común al ejercicio anterior. Se extrae el nombre de los enlaces con `xmlValue()` [63] y los enlaces con `xmlGetAttr()` [64].

En primer lugar se va a realizar la obtención de enlaces mediante su identificador, lo que se puede observar en el código fuente 12 y en las ilustraciones 3.5 y 3.6.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5  #Se obtienen los enlaces por su id
6
7  links <- getNodeSet(pagina, "//*[@id='link_seccion_formularios'
   ↪  or @id='link_seccion_js']")
8
9  #Ahora se obtiene por un lado los identificadores, por otros los
   ↪  textos de los enlaces y por ultimo las direcciones
10
11 ids <- sapply(links, function(x){ xmlGetAttr(x, "id") })
12 textos <- sapply(links, function(x){ xmlValue(x) })
13 direcciones <- sapply(links, function(x){ xmlGetAttr(x, "href")
   ↪  })
14
15 #Se introduce en un dataframe y se muestra en pantalla
16
17 tabla <- data.frame(ids, textos, direcciones)
18 View(tabla)
```

Código fuente 12: Ejercicio de obtención de enlaces por su identificador con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces por su id
>
> links <- getNodeSet(pagina, "//*[@id='link_seccion_formularios' or @id='link_$
>
> #Ahora se obtiene por un lado los identificadores, por otros los textos de lo$
>
> ids <- sapply(links, function(x){ xmlGetAttr(x, "id") })
> textos <- sapply(links, function(x){ xmlValue(x) })
> direcciones <- sapply(links, function(x){ xmlGetAttr(x, "href") })
>
> #Se introduce en un dataframe y se muestra en pantalla
>
> tabla <- data.frame(ids, textos, direcciones)
> View(tabla)
>
```

Ilustración 3.5: Resultado de ejecución del ejercicio de obtención de enlaces por su identificador con *XML*

	ids	textos	direcciones
1	link_seccion_formularios	Envío de formularios	../forms
2	link_seccion_js	Renderizado de JavaScript	../test-js

Ilustración 3.6: Tabla con el resultado de ejecución del ejercicio de obtención de enlaces por su identificador con *XML*

En segundo lugar se va a realizar la obtención de enlaces mediante su clase, lo que se puede observar en el código fuente 13 y en las ilustraciones 3.7 y 3.8.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5  #Se obtienen los enlaces por su clase
6
7  xpath <- "//*[contains(concat(' ', @class, ' '), '
   ↪  dificultad_media ') or contains(concat(' ', @class, ' '), '
   ↪  dificultad_alta ')]"
8
9  links <- getNodeSet(pagina, xpath)
10
11 #Ahora se obtiene por un lado las clases, por otros los textos
   ↪  de los enlaces y por ultimo las direcciones
12
13 clases <- sapply(links, function(x){ xmlGetAttr(x, "class") })
14 textos <- sapply(links, function(x){ xmlValue(x) })
15 direcciones <- sapply(links, function(x){ xmlGetAttr(x, "href")
   ↪  })
16
17 #Se introduce en un dataframe y se muestra en pantalla
18
19 tabla <- data.frame(clases, textos, direcciones)
20 View(tabla)
```

Código fuente 13: Ejercicio de obtención de enlaces por su clase con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces por su clase
>
> xpath <- "//*[contains(concat(' ', @class, ' '), ' dificultad_media ') or con$
>
> links <- getNodeSet(pagina, xpath)
>
> #Ahora se obtiene por un lado las clases, por otros los textos de los enlaces$
>
> clases <- sapply(links, function(x){ xmlGetAttr(x, "class") })
> textos <- sapply(links, function(x){ xmlValue(x) })
> direcciones <- sapply(links, function(x){ xmlGetAttr(x, "href") })
>
> #Se introduce en un dataframe y se muestra en pantalla
>
> tabla <- data.frame(clases, textos, direcciones)
> View(tabla)
>
```

Ilustración 3.7: Resultado de ejecución del ejercicio de obtención de enlaces por su clase con *XML*

	clases	textos	direcciones
1	dificultad_media	Envío de formularios	../forms
2	dificultad_media	Gestión de cookies de sesión	../cookies
3	dificultad_alta	Renderizado de JavaScript	../test-js
4	dificultad_alta	Performance	../performance

Ilustración 3.8: Tabla con el resultado de ejecución del ejercicio de obtención de enlaces por su clase con *XML*

3.5.5. Obtener los atributos de una imagen

Se van a obtener los atributos básicos de una imagen. Los atributos que se han utilizado a la hora de crear este test de propósito general son los que suelen estar presentes, esto es, un atributo de ubicación (*src*). Además, es aconsejable que tenga un texto alternativo (atributo *alt*).

Para ello simplemente hay que localizar la imagen mediante su identificador y utilizar la función `getNodeSet()` [62]. Posteriormente se extraen todos los atributos requeridos utilizando `xmlAttrs()` [65].

El código fuente completo se puede observar en el código fuente 14 y el resultado de su ejecución puede ser observado en la ilustración 3.9.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4  #Se obtiene la imagen
5  img <- getNodeSet(pagina, "//*[@id='imagen']")[[1]]
6
7  #Se obtienen todos los atributos de la imagen
8
9  atributos <- iconv(xmlAttrs(img, "alt"), "UTF-8")
10
11 print(atributos)
```

Código fuente 14: Ejercicio de obtención de los atributos de una imagen con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
> #Se obtiene la imagen
> img <- getNodeSet(pagina, "//*[@id='imagen']")[[1]]
>
> #Se obtienen todos los atributos de la imagen
>
> atributos <- iconv(xmlAttrs(img, "alt"), "UTF-8")
>
> print(atributos)
                                  id                                   src
                            "imagen"        "pics/original-compressor.jpeg"
                                 alt
"Fotografía de carretera convencional"
>
```

Ilustración 3.9: Resultado de ejecución del ejercicio de obtención de los atributos de una imagen con *XML*

3.5.6. Obtener diversas etiquetas de una clase y diferenciarlas

A la hora de crear el test de propósito general algunos elementos HTML en los párrafos fueron coloreados estéticamente con una coloración roja (que suele ser interpretado por el usuario como elementos importantes). Dicha coloración se establece mediante una clase denominada ïmportante".

El objetivo es extraer todos los elementos que tengan esta clase, indicando a cual de los siguientes tres apartados corresponde: uno primero, con los elementos en línea `strong`, que es un indicador de que es un texto al que se le quiere dar relevancia según los estándares del W3C [68], esto es, un texto importante; un segundo apartado en donde la clase es aplicada a un enlace (a) y un tercer apartado a modo de cajón de sastre donde se incluyen los elementos que no están bajo el paraguas de los dos apartados anteriores.

Para proceder con este ejercicio, en primer lugar se buscan todos los elementos que tengan como clase ïmportante" haciendo uso de la función `getNodeSet()` [62]. Es importante volver a destacar que *XPath* no dispone de un selector natural para elegir clases, por lo que para verificar una clase, se debe poner el atributo entre espacios y ver si la cadena contiene una subcadena formada por la clase deseada utilizando espacio al principio y al final. La utilidad de esto es porque dentro del mismo atributo puede haber varias clases, por lo que no serviría la comparación mediante el igual (que verificaría si ambas cadenas son estrictamente iguales).

Posteriormente se recorren en bucle los elementos encontrados y son clasificados verificando para cada elemento localizado qué etiqueta es con la función `xmlName()` [66].

En función de qué tipo de etiqueta sea, se extrae solo el texto con `xmlValue()` [63] o también se extrae en el caso de un enlace su dirección con `xmlGetAttr()` [64]. Si no se puede catalogar el elemento en ninguno de los dos casos mencionados anteriormente, se indicaría en un mensaje por pantalla.

Se puede observar de forma detallada este algoritmo en el código fuente 15. Además se puede observar la salida de ejecución en la ilustración 3.10 donde se puede observar que se han localizado dos elementos con la clase ïmportante", de los cuales uno es un enlace a una página del sitio web referida a *performance*, y la otra es simplemente un texto que se ha deseado destacar.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5
6  #Se toman los elementos de importancia
7  importantes <- getNodeSet(pagina, "//*[contains(concat(' ',
   ↪  @class, ' '), ' importante ')]")
8
9  for(imp in importantes){
10         tag <- xmlName(imp)
11         if(tag == "a"){
12                 print("ENLACE-------------------------")
13                 print(paste("    contenido: ",xmlValue(imp)))
14                 print(paste("    enlace:
                    ↪  ",xmlGetAttr(imp,"href")))
15         }else if(tag == "strong"){
16                 print("STRONG-------------------------")
17                 print(paste("    contenido: ",xmlValue(imp)))
18         }else{
19                 print("No es enlace ni strong")
20         }
21 }
```

Código fuente 15: Ejercicio de obtención de diversas etiquetas/elementos de una clase y su diferenciación con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
>
> #Se toman los elementos de importancia
> importantes <- getNodeSet(pagina, "//*[contains(concat(' ', @class, ' '), ' i$
>
> for(imp in importantes){
+ tag <- xmlName(imp)
+ if(tag == "a"){
+ print("ENLACE-------------------------")
+ print(paste("    contenido: ",xmlValue(imp)))
+ print(paste("    enlace: ",xmlGetAttr(imp,"href")))
+ }else if(tag == "strong"){
+ print("STRONG-------------------------")
+ print(paste("    contenido: ",xmlValue(imp)))
+ }else{
+ print("No es enlace ni strong")
+ }
+ }
[1] "STRONG-------------------------"
[1] "    contenido:  esto es un texto importante"
[1] "ENLACE-------------------------"
[1] "    contenido:  esto es un enlace importante relativo al performance"
[1] "    enlace:  ../performance"
>
```

Ilustración 3.10: Resultado de ejercicio de obtención de diversas etiquetas/elementos de una clase y su diferenciación con *XML*

3.5.7. Obtener para cada enlace del cuerpo, a qué párrafo pertenece

El objetivo de este ejercicio es determinar para cada enlace que hay dentro del cuerpo `main` de la página, determinar a qué párrafo corresponde (o a qué elemento superior si este no fuera un párrafo, aunque en el test diseñado se tienen párrafos en todo momento).

Para realizarlo se utilizará un selector *XPath* que permita acceder a cualquier enlace que esté dentro de `main` y posteriormente se utilizará un selector para acceder al antecesor (el padre).

En primer lugar, se accede a cada uno de enlaces que contiene el cuerpo con la función `getNodeSet()` [62]. Haciendo uso de `sapply()` y de la función `xmlGetAttr()` [64] se obtienen las direcciones URL de los enlaces en una lista.

Posteriormente nuevamente mediante `getNodeSet()` [62] y `sapply()` se accede por cada nodo al padre para obtener el párrafo. Del párrafo simplemente se requiere el texto que se extraerá con la función `xmlValue()` [63].

Por último se genera un *data frame* con la lista de las direcciones URL y el párrafo en el que se encuentran.

El código fuente 16 destaca cómo se realiza el algoritmo explicado de forma más detallada. Así mismo, se puede observar el resultado de la ejecución en la ilustración 3.11. Por último se puede observar la tabla generada con los enlaces y los párrafos padre en la ilustración 3.12.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5  #Se obtienen los enlaces del cuerpo
6  enlaces <- getNodeSet(pagina,"//main//a")
7  urls <- sapply(enlaces, function(x){xmlGetAttr(x,"href")})
8
9  #Se obtienen los padres
10
11 parrafos <- sapply(enlaces, function(x){
12                 xmlValue(getNodeSet(x,"..")[[1]])
13                 })
14
15 #Se genera la tabla con enlaces y sus parrafos
16 tabla <- data.frame(urls, parrafos)
17 View(tabla)
```

Código fuente 16: Ejercicio de obtención de enlaces y párrafos con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces del cuerpo
> enlaces <- getNodeSet(pagina,"//main//a")
> urls <- sapply(enlaces, function(x){xmlGetAttr(x,"href")})
>
> #Se obtienen los padres
>
> parrafos <- sapply(enlaces, function(x){
+ xmlValue(getNodeSet(x,"..")[[1]])
+ })
>
> #Se genera la tabla con enlaces y sus parrafos
> tabla <- data.frame(urls, parrafos)
> View(tabla)
>
```

Ilustración 3.11: Resultado de ejecución del ejercicio de obtención de enlaces y párrafos con *XML*

	urls	parrafos
1	http://tfg.daniel-fl.com/performance	Mientras que esto es un texto importante que tien>
2	http://tfg.daniel-fl.com/forms	Tras la finalización de este supuesto, se debe em>

Ilustración 3.12: Tabla con resultado de ejecución del ejercicio de obtención de enlaces y párrafos con *XML*

3.5.8. Ver los enlaces hermanos de los textos importantes

Para empezar el test se extrae con el selector *XPath* conveniente todos los nodos que tengan simultáneamente al menos un hijo enlace a y un hijo importante strong, contando con la ayuda de la función `getNodeSet()` [62].

Posteriormente para cada nodo se obtienen por un lado los enlaces (de los que se extrae el nombre con la función `xmlValue()` [63] y el enlace con la función `xmlGetAttr()` [64]) y por el otro los textos importantes (de los que se extrae simplemente el nombre).

Se puede observar el algoritmo desarrollado en R en el código fuente 17 y una muestra del resultado de ejecución en la ilustración 3.13.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5  #Se extraen aquellos padres que tengan como hijos a la vez nodos
   ↪  importantes y nodos enlaces
6  padres <- getNodeSet(pagina, "//*[strong][a]")
7
8  #Para cada nodo de los seleccionados,
9  #se extrae su contenido y luego las partes de enlaces y textos
   ↪  importantes
10 for(padre in padres){
11
12         print(paste("+Padre: ", xmlValue(padre)))
13
14         print("-Textos importantes:")
15         sapply(getNodeSet(padre,"strong"), function(x){
16                 print(xmlValue(x))
17         })
18
19         print("-Enlaces:")
20         sapply(getNodeSet(padre,"a"), function(x){
21                 print(paste(xmlValue(x), " - ",
                   ↪  xmlGetAttr(x,"href")))
22         })
23 }
```

Código fuente 17: Ejercicio de obtención de de enlaces hermanos de textos importantes con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se extraen aquellos padres que tengan como hijos a la vez nodos importantes $
> padres <- getNodeSet(pagina, "//*[strong][a]")
>
> #Para cada nodo de los seleccionados,
> #se extrae su contenido y luego las partes de enlaces y textos importantes
> for(padre in padres){
+
+ print(paste("+Padre: ", xmlValue(padre)))
+
+ print("-Textos importantes:")
+ sapply(getNodeSet(padre,"strong"), function(x){
+ print(xmlValue(x))
+ })
+
+ print("-Enlaces:")
+ sapply(getNodeSet(padre,"a"), function(x){
+ print(paste(xmlValue(x), " - ", xmlGetAttr(x,"href")))
+ })
+ }
[1] "+Padre:  Mientras que esto es un texto importante que tiene una clase impo$
[1] "-Textos importantes:"
[1] "esto es un texto importante"
[1] "-Enlaces:"
[1] "esto es un enlace importante relativo al performance  -  ../performance"
>
```

Ilustración 3.13: Resultado de ejecución de ejercicio de obtención de de enlaces hermanos de textos importantes con *XML*

3.5.9. Obtener la tabla que hay en el cuerpo

A continuación se va a extraer de la página de propósito general una tabla (la única que hay). Para ello, se va a hacer uso de la función `readHTMLTable()` [67] sobre una tabla determinada que se debe obtener utilizando un selector *XPath* la función `getNodeSet()` [62].

Como solo hay una tabla, dicho elemento se puede extraer directamente indicando el nombre de la etiqueta/elemento fboxtable. En caso de ser varias tablas, en caso de desear extraer una, habría que utilizar un selector más concreto.

En la función de extracción de tablas se ha asignado el valor verdadero al argumento header. Este argumento indica que los nombres de columnas a utilizar se indican en los elementos th o en su defecto si no los hubiera, en la primera fila.

Por último, se hace uso de la función *iconv()* aplicada a los nombres de columnas, ya que desafortunadamente *XML* no codifica bien los nombres de las columnas con tildes (en este caso la palabra *Miércoles*) por lo que utilizando esta función e indicando la codificación correcta por parámetro (UTF-8) el problema queda solucionado.

El código fuente 18 muestra cómo se obtiene dicha tabla, generando un *data frame* que es mostrado en pantalla posteriormente. El contenido del *data frame* se puede observar en las ilustraciones 3.15 y 3.16, mientras que el resultado de la ejecución (no devuelve nada más allá que la impresión de los frames de datos) en la ilustración 3.14.

```
1  library(XML)
2
3  pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
4
5  #Se obtienen la tabla que hay (solo hay una)
6  tablaNodo <- getNodeSet(pagina, "//table")[[1]]
7  tabla <- readHTMLTable(tablaNodo, header = TRUE)
8
9  #Se corrige la codificacion de la cabecera para el miercoles
10
11 colnames(tabla) <- iconv(colnames(tabla), "UTF-8")
12
13 View(tabla)
```

Código fuente 18: Ejercicio de obtención de tabla con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen la tabla que hay (solo hay una)
> tablaNodo <- getNodeSet(pagina, "//table")[[1]]
> tabla <- readHTMLTable(tablaNodo, header = TRUE)
>
> #Se corrige la codificación de la cabecera para el miercoles
>
> colnames(tabla) <- iconv(colnames(tabla), "UTF-8")
>
> View(tabla)
>
```

Ilustración 3.14: Resultado de ejecución de ejercicio de obtención de tabla con *XML*

	Lunes	Martes	Miércoles
1	Ensalada	Paella	Sopa de arroz
2	Pasta con tomate	Guisantes	Albóndigas
3	Lentejas con chorizo	Plato combinado	Croquetas
4	Lomo con patatas	Redondo de ternera	Zarzuela de pescado

Ilustración 3.15: Tabla de ejecución del ejercicio de obtención de tabla con *XML* (I)

	Miércoles	Jueves	Viernes
1	Sopa de arroz	Espaguetis a la carbonara	Judías con jamón
2	Albóndigas	Arroz con conejo	Cocido completo
3	Croquetas	Fabada	Pescado
4	Zarzuela de pescado	Guisado de costillas	Croquetas

Ilustración 3.16: Tabla de ejecución del ejercicio de obtención de tabla con *XML* (II)

3.6. Realización del test de envío de formularios con XML

El objetivo de este ejercicio es observar cómo se puede interactuar con el servidor mediante *XML* haciendo un envío de un formulario y observando si la respuesta obtenida es la esperada.

Desafortunadamente no se ha encontrado ninguna funcionalidad en el paquete que se está analizando que permita el envío de parámetros al servidor mediante POST o GET (que sería la aproximación más sencilla a este ejercicio). Por lo tanto se puede concluir que *XML* de forma nativa no permite hacer este supuesto.

No obstante, una posible aproximación sería utilizar *RCurl* [69] o *httr* [70] para hacer la petición inicial enviando los parámetros GET o POST y luego redirigir la respuesta a XML. La realización de esta aproximación se deja a criterio del lector, que podrá determinar utilizar otro paquete (como *rvest*) o realizar este supuesto. El motivo de la no realización de esta aproximación estriba en que el objetivo del presente trabajo es ver qué funcionalidades provee cada paquete y qué funcionalidades no provee, pero no se busca realizar alternativas ante las funcionalidades no provistas por los paquetes.

3.7. Realización del test de gestión de cookies con XML

La utilidad de este ejercicio estriba en saber como actúa *XML* cuando se necesita almacenar y enviar al servidor un identificador de *sesión* en base al que el contenido proporcionado en la navegación varía, esto es, que la información mostrada en la web visitada no es siempre la misma para todo el mundo, sino que varía en función del usuario y de las peticiones previas que haya hecho navegando por distintas páginas o autenticándose, entre otros. Lo que permite el almacenamiento de estos estados para un cliente dado se conoce como sesiones y el acceso a la sesión del cliente se produce porque se envía su identificador. Los navegadores almacenan este identificador en lo que se conoce como *cookie*.

En este supuesto, se van a probar si *XML* es capaz de gestionar de forma nativa estas *cookies*. Para ello simplemente se va a solicitar la página. Esta acción se va a realizar varias veces (en bucle) y se va a observar si la respuesta varía en cada petición, o si por el contrario no lo hace.

La página web creada para este supuesto, permite ver en la sesión actual cuántas veces se ha visitado. El número de veces que se ha visitado la página está aislado con un identificador, para poder *scrapearlo* fácilmente.

Para ello, todo dentro de un bucle `for`, se obtiene la página con la función `htmlParse()` [60]. Tras ello se extrae el nodo que contiene el número concreto de peticiones previas realizadas según el servidor con la función `getNodeSet()` [62] y por último se extrae su valor con la función `xmlValue()` [63] y se inserta en el *data frame* que se mostrará posteriormente.

El algoritmo utilizado para realizar esta prueba se puede observar en el código fuente 19. La salida producida se puede observar en la ilustración 3.17. Además se ha generado una tabla (ilustración 3.18) que permite observar que todo el rato se muestra que previamente se habían realizado 0 visitas. Esto quiere decir, que realizando las peticiones como se venía haciendo, *XML* no guarda la *cookie de sesión* y por lo tanto no la envía al servidor, por lo que el servidor no la puede obtener en cada petición y asume que no había una sesión previa generada, por lo que opta por crear una nueva en cada petición.

```
1  library(XML)
2
3  df <-data.frame(Visitas=integer())
4
5  for(i in 1:5){
6          pagina <- htmlParse("http://tfg.daniel-fl.com/cookies/")
7          elemVis <- getNodeSet(pagina,
           ↪  "//span[@id='visitas']")[[1]]
8          visitas <- xmlValue(elemVis)
9          df[nrow(df) + 1,] = c(visitas)
10 }
11
12 View(df)
```

Código fuente 19: Verificación (fallida) de mantenimiento transparente de sesiones con *XML*

```
> library(XML)
>
> df <-data.frame(Visitas=integer())
>
> for(i in 1:5){
+ pagina <- htmlParse("http://tfg.daniel-fl.com/cookies/")
+ elemVis <- getNodeSet(pagina, "//span[@id='visitas']")[[1]]
+ visitas <- xmlValue(elemVis)
+ df[nrow(df) + 1,] = c(visitas)
+ }
>
> View(df)
>
```

Ilustración 3.17: Resultado de ejecución de la verificación (fallida) de mantenimiento transparente de sesiones con *XML*

Tras verificar la documentación de XML y ver las distintas opciones que se ofrecen ante las consultas de ciertos desarrolladores en la comunidad, se observa que *XML* no provee por sí mismo de capacidad para gestionar la navegación

	Visitas
1	0
2	0
3	0
4	0
5	0

Ilustración 3.18: Tabla generada en el resultado de ejecución de la verificación (fallida) de mantenimiento transparente de sesiones con *XML*

almacenando los identificadores de sesión, ya que se limita a descargar la URL indicada y a parsearla, siendo cada petición totalmente independiente. Para realizar este supuesto con *XML* habría que renunciar a realizar la descarga con *XML*, pasando a ser realizada la descarga utilizando *RCurl* [69] o *httr* [70], manteniendo el objeto correspondiente que represente a la sesión, que variará posiblemente según el paquete que se utilice, y redirigiendo los documentos HTML descargados mediante estas herramientas a *XML*.

3.8. Realización del test de renderizado de JavaScript con XML

El objetivo de este ejercicio es determinar si *XML* renderiza JavaScript tras descargar las páginas web o no.

Se ha optado por utilizar el test de renderizado destinado a la verificación de renderizado de JavaScript para la incorporación de profesionales al DOM. Por tanto, se va a proceder a descargar dicha página y a verificar si la caja cuyo identificador es *cargando* existe. En caso de existir, ello implica por como se ha realizado el test que la página no ha sido renderizada. Por el contrario, si no existiera, ello implicaría que *XML* ha descargado la página y la ha renderizado (o al menos lo ha intentado).

Para ello se debe intentar extraer la caja con `getNodeSet()` [62]. A continuación se verifica el número de nodos extraídos.

Como se puede observar en la ilustración 3.19, *XML* no renderiza las páginas tras descargarlas, sino que *parsea* directamente la respuesta obtenida. El algoritmo diseñado que prueba esto se puede encontrar en el código fuente 20

```r
library(XML)

pagina <- htmlParse("http://tfg.daniel-fl.com/test-js/")

cargando <- getNodeSet(pagina, "//*[@id='cargando']")

if(length(cargando) > 0){
        print("Div de carga encontrado. El navegador no admite
        ↪  carga de JavaScript")
}else{
        print("Caja de carga no se muestra. Eso implica que el
        ↪  JS se ha ejecutado.")
}
```

Código fuente 20: Test de renderizado de JavaScript con *XML*

```
> library(XML)
>
> pagina <- htmlParse("http://tfg.daniel-fl.com/test-js/")
>
> cargando <- getNodeSet(pagina, "//*[@id='cargando']")
>
> if(length(cargando) > 0){
+ print("Div de carga encontrado. El navegador no admite carga de JavaScript")
+ }else{
+ print("Caja de carga no se muestra. Eso implica que el JS se ha ejecutado.")
+ }
[1] "Div de carga encontrado. El navegador no admite carga de JavaScript"
>
```

Ilustración 3.19: Resultado de ejecución del test de renderizado de JavaScript con *XML*

3.9. Conclusiones de XML

Tras explicar la funcionalidad de *XML* y una vez utilizada una parte de la misma para realizar los tests diseñados, se pueden obtener varias conclusiones. En la tabla que aparece a continuación se puede observar una serie de datos de interés, tales como la dificultad de los supuestos planteados así como algunos datos de carácter general.

Sobre este paquete se pueden hacer las siguientes afirmaciones, en base a la experiencia adquirida con él.

- La instalación es bastante sencilla.
- La curva de aprendizaje es baja/media. Quizás el número de funciones que hay sea excesivo (este paquete además de leer XML/HTML también permite crear XML).
- Más allá de parsear, no permite realizar ningún tipo de interacción con el servidor (sesiones, formularios, renderizado y ejecución de *JavaScript*).

PARÁMETROS GENERALES	
Dificultad de instalación	Fácil
Complejidad de la sintaxis	Media
Pendiente de aprendizaje	Media (muchas funciones)
TEST DE PROPÓSITO GENERAL	
Ejemplo de petición HTTP	Fácil
Obtener metaetiqueta del título de la página, descripción y palabras clave	Fácil
Obtener todos los enlaces de la barra de navegación	Fácil
Obtener enlaces por su clase y por su identificador	Medio (dificultad en XPath)
Obtener los atributos de una imagen	Medio (problemas codificación)
Obtener diversas etiquetas de una clase y diferenciarlos	Fácil
Obtener para cada enlace del cuerpo, a qué párrafo pertenece	Medio
Ver los enlaces hermanos de los textos importantes	Medio (dificultad en XPath)
Obtener la tabla que hay en el cuerpo	Fácil (algún problema codificación)
TEST DE ENVÍO DE FORMULARIOS	
Envío de información a través de formularios	No es posible
TEST DE GESTIÓN DE COOKIES DE SESIÓN	
Mantenimiento de sesiones	No lo supera
TEST DE RENDERIZADO DE JAVASCRIPT	
Webscraping ante situaciones de carga asíncrona del DOM	Imposible - no renderiza JS

Capítulo 4

Web scraping con rvest

4.1. Introducción a rvest

Rvest es una biblioteca de R destinada a facilitar la descarga y manipulación de páginas web (HTML) y documentos XML, y para la realización de *web scraping*. Está diseñada para trabajar con *magrittr*, biblioteca que facilita el trabajo, en particular del paso de resultados de funciones, implementando un operador tubería que se explicará profusamente a continuación. Esta biblioteca está inspirada en *Beautiful Soup*, homóloga en Python [25].

Esta biblioteca básicamente es un *wrapper* de dos bibliotecas o paquetes dependientes, que son las siguientes [25]:

- **xml2:** Trabaja con ficheros XML utilizando una simple y consistente interfaz. Es un *wrapper* de la biblioteca *libxml2* que trabaja sobre C [71].
- **httr:** Son utilidades para trabajar con comunicaciones HTTP, con los métodos GET, POST, PUT y DELETE entre otros [70].

Además, otro de los componentes importantes es *selectr* que da capacidad a *rvest* para utilizar selectores CSS.

Se puede observar en la ilustración 4.1 un esquema de los componentes principales de *rvest*

4.2. Visión general de las funcionalidades de rvest

A continuación se procede a enunciar brevemente las diversas funcionalidades que posee el paquete [25].

- Crear un documento HTML desde una URL, un archivo en el disco duro o directamente desde una cadena, utilizando la función `read_html()`.
- Seleccionar partes de un documento utilizando selectores CSS (o bien si se prefiere, XPath) gracias a la función `html_nodes()`.

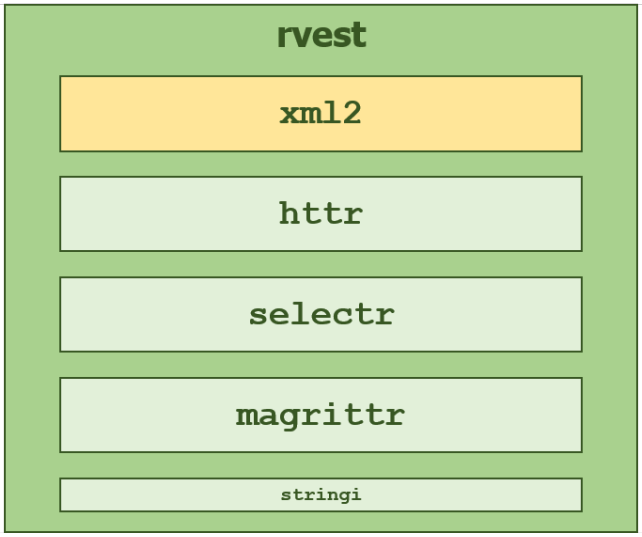


Ilustración 4.1: Principales componentes que conforman el *wrapper* de *rvest*

- Se puede utilizar *rvest* para extraer la información de los nodos que se van iterando, el nombre (con `html_tag()`), el texto que hay dentro de la etiqueta (con `html_text()`), o los atributos gracias a `html_attr()`. y `html_attrs()`, con los que podrá obtener la primera o todas las apariciones del atributo escogido.
- También se puede utilizar *rvest* con ficheros XML, ya que existen funciones hermanas para dicho propósito con idéntico propósito a las de HTML, tal y como se muestra en la tabla siguiente:

Funciones para HTML	**Funciones para XML**
`html_node()`	`xml_node()`
`html_nodes()`	`xml_nodes()`
`html_attr()`	`xml_attr()`
`html_attrs()`	`xml_attrs()`
`html_text()`	`xml_text()`
`html_tag()`	`xml_tag()`

- Parsear una tabla a un *data frame* con `html_table()`.
- Extraer, modificar y enviar formularios con `html_form()`, `set_values()` y `submit_form()`
- Detectar y reparar problemas de codificación con `guess_encoding()` y con `repair_encoding()`

- Navegar a través de un sitio web como si se realizara a través de un explorador web, utilizando `html_session()`, `jump_to()`, `follow_link()`, `back()`, `forward()`, etcétera. Estas funcionalidades de navegación están en desarrollo en el momento de la redacción de este documento.

4.3. Instalación de rvest

Para la ejecución de los ejercicios que se exponen a continuación, se asume que se dispone del entorno de ejecución para R instalado. Si no se dispusiera de él, simplemente habría que descargárselo desde alguno de los espejos existentes como puede ser RedIRIS [59].

4.3.1. Instalación de la versión estable de rvest

Para instalar la versión estable o en producción de **rvest** simplemente se debe abrir la consola de R e introducir el comando que se muestra en el código fuente 21 [26]. Esta opción instalará el paquete estable más reciente. Posiblemente se requiera la elección de un servidor espejo, se elige el más reciente y se continúa con la instalación.

```
1 install.packages("rvest")
```

Código fuente 21: Instalación de la versión estable del paquete *rvest*

Automáticamente se descargarán todas las dependencias necesarias (véase la ilustración 4.2), por lo que no será necesario realizar ninguna actuación adicional.

4.3.2. Instalación de la versión en desarrollo de rvest

En el presente documento se ha trabajado con la versión estable. No obstante, si el autor de *rvest* tuviera en el futuro alguna funcionalidad en la versión de desarrollo que pueda ser de su interés, puede descargarla siguiendo los siguientes pasos.

1. Descargue el paquete *webtools* si no dispone de él. En el código fuente 22 se puede observar la invocación necesaria para la instalación (comentada, debe ser descomentada si se requiere).

2. Mediante el paquete *webtools* invocar a la función que permite descargar el paquete de desarrollo que permite su descarga desde GitHub (código fuente 22) [26].

```
downloaded 309 KB

probando la URL 'https://cran.rediris.es/bin/windows/contrib/3.4/stringr_1.2.0.zip'
Content type 'application/zip' length 149108 bytes (145 KB)
downloaded 145 KB

probando la URL 'https://cran.rediris.es/bin/windows/contrib/3.4/xml2_1.1.1.zip'
Content type 'application/zip' length 3529661 bytes (3.4 MB)
downloaded 3.4 MB

probando la URL 'https://cran.rediris.es/bin/windows/contrib/3.4/httr_1.3.1.zip'
Content type 'application/zip' length 301771 bytes (294 KB)
downloaded 294 KB

probando la URL 'https://cran.rediris.es/bin/windows/contrib/3.4/selectr_0.3-1.zip'
Content type 'application/zip' length 168835 bytes (164 KB)
downloaded 164 KB

probando la URL 'https://cran.rediris.es/bin/windows/contrib/3.4/magrittr_1.5.zip'
Content type 'application/zip' length 155420 bytes (151 KB)
downloaded 151 KB

probando la URL 'https://cran.rediris.es/bin/windows/contrib/3.4/rvest_0.3.2.zip'
Content type 'application/zip' length 853729 bytes (833 KB)
downloaded 833 KB

package 'stringi' successfully unpacked and MD5 sums checked
package 'Rcpp' successfully unpacked and MD5 sums checked
```

Ilustración 4.2: Principales dependencias de *rvest*

```
1 #install.packages("devtools") #Solo si se requiere
2 devtools::install_github("hadley/rvest")
```

Código fuente 22: Instalación la versión en desarrollo del paquete *rvest*

4.4. Detalle pormenorizado de las funciones de rvest

4.4.1. El operador tubería (%>%)

Debido a que el encadenamiento de funciones mediante el operador de tubería es utilizado en varios capítulos, ha sido ubicado en el anexo A. Es aconsejable leerlo antes de empezar a manejar el paquete *rvest* ya que en él aprenderá la sintaxis para encadenar funciones, muy útil porque aligera enormemente la sintaxis.

4.4.2. Obtención de un fichero HTML/XML

A continuación se explica en el código fuente 23 y en la tabla de argumentos el uso de la función que permite obtener un fichero HTML (o bien XML) [72], para posteriormente poder aplicar el resto de funcionalidades que *rvest* provee.

```
1 read_html(x, ...)
2 read_html(x, ..., encoding = "")
3 read_xml(x, ..., as_html = FALSE)
4 read_xml(x, ..., encoding = "", as_html = FALSE)
```

Código fuente 23: Función de obtención de un fichero HTML/XML con *rvest*

Argumentos

`x`	Una URL, una ruta local, una cadena que contenga HTML o XML o bien una respuesta de una petición *httr*.
`...`	Parámetro opcional. En caso de que `x` sea una función, se obtendrán los parámetros adicionales en la función GET().
`encoding`	Parámetro opcional. Codificación ISO de la página.
`as_html`	Parámetro opcional. Permite parsear un fichero XML como si fuera HTML.

4.4.3. Extracción de nodos

Una vez que se ha obtenido (*parseado*) el documento HTML, la opción más usual suele ser seleccionar un nodo o grupo de nodos. Para ello se utilizan las funciones que se pueden observar en el código fuente 24 y en la tabla de argumentos que se expone a continuación [73].

```
1 html_nodes(x, css, xpath)
2 html_node(x, css, xpath)
```

Código fuente 24: Función de extracción de nodos de un documento HTML/XML previamente parseado con *rvest*

Argumentos

`x`	Un documento, un conjunto de nodos o bien un solo nodo.
`css, xpath`	Nodos a seleccionar. Se debe suministrar uno u otro en función del tipo de selector a utilizar. Si se utiliza el selector CSS, no es necesario indicarlo con una asignación, mientras que con XPath sí que es necesario (xpath = 'selector XPath').

La selección se puede hacer utilizando selectores CSS o bien selectores XPath 1.0. Los selectores CSS son muy usuales. Además, si se utiliza http://selectorgadget.com/, esta herramienta indica qué selector debes utilizar exactamente para la sección de la página que deseas obtener [73].

Además, si nunca se ha utilizado CSS, el autor de *rvest* recomienda acceder al recurso ubicado en la web http://flukeout.github.io/. Personalmente, aconsejo el manual ubicado en https://librosweb.es/libro/css/. Aunque no es tan interactivo, es un recurso muy sencillo que expone los principios de CSS de forma guiada y en español.

A la hora de elegir entre las dos funciones, hay que tener en cuenta un criterio: cuántos nodos se desean obtener utilizando la función indicada. La función `html_node()` solo permite obtener el primer nodo, mientras que `html_nodes()` permite obtener todos los nodos. Realmente ambas funciones obtienen una lista, lo único que en la primera opción solo se obtiene el primer elemento (esto es útil por razones de eficiencia, puesto que es probable que la función singular al buscar en el árbol, al encontrar el primer nodo deje de seguir buscando).

Respecto al soporte para selectores CSS, la compatibilidad es provista por un paquete que se denomina *selectr*. Este paquete permite extraer la mayor parte de los selectores CSS3, salvo las siguientes excepciones:

- No se pueden utilizar pseudoselectores que requieran interactividad: `:hover`, `:active`, `:focus`, `:target`, `:visited`.
- Aquellos selectores en cuya composición se utilice el elemento comodín * (este elemento permite seleccionar todos los elementos, según como se utilice, puede indicar a solas absolutamente todos los elementos o bien permite hacer selecciones más compuestas) no pueden ser utilizados.
- Se puede utilizar el operador `!=` siendo `[foo!=bar]` lo mismo que `:not([foo=bar])`

- `:not()` acepta una secuencia de selectores simples, no solo un selector simple.
- Se soporta el selector `:contains(texto)`.

4.4.4. Obtener información concreta de los nodos

Una vez que ya se han obtenido uno o varios nodos, se debe extraer información concreta de esos nodos. La información más esencial que se extraer son los atributos, el texto y el nombre de la etiqueta [74]. En la ilustración 4.3 se puede observar un ejemplo de un párrafo en HTML indicando lo que es un atributo y dónde se ubica el texto y el nombre de la etiqueta.

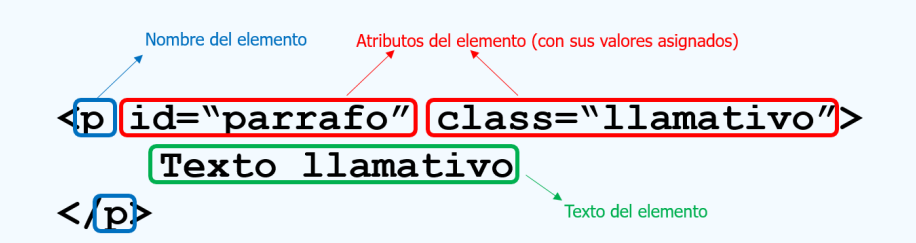


Ilustración 4.3: Indicación de atributos, texto y nombre en un párrafo HTML

Además, se pueden ver en el código fuente 25 las funciones a utilizar y a continuación sus argumentos.

```
1 html_text(x, trim = FALSE)
2 html_name(x)
3 html_children(x)
4 html_attrs(x)
5 html_attr(x, name, default = NA_character_)
```

Código fuente 25: Extracción de información de nodos con *rvest*

Argumentos

A continuación se indica el uso de las distintas funciones:

- `html_text()`: Permite obtener el texto del elemento o de los elementos pasados por parámetro. Si se establece el parámetro indicado, se puede limpiar de espacios en blanco en el inicio y el fin de la cadena.

`x`	Un documento, un conjunto de nodos o bien un solo nodo.
`trim`	Parámetro booleano en el que si se indica como cierto en la invocación, elimina los espacios en blanco sobrantes al principio y al final de la cadena.
`name`	Nombre del atributo a obtener.
`default`	Cadena utilizada como valor por defecto cuando el atributo no existe en cada nodo.

- html_name(): Permite obtener el nombre del nodo o los nodos pasados por parámetro.
- html_children(): Permite obtener los hijos del elemento o conjunto de elementos.
- html_attrs(): Permite obtener todos los atributos y sus valores del elemento o elementos indicados por parámetro.
- html_attr(): Permite obtener el valor de un solo atributo, indicando por parámetro su nombre.

En el caso de las funciones html_text(), html_attr(), html_name() se obtiene una vector de caracteres (una cadena), en el caso de la función html_attrs() una cadena. En caso de haber pasado un conjunto de nodos, se recibirá una lista con una posición por cada uno de los elementos que están en el conjunto, conteniendo cada posición un dato del tipo que correspondería si solo se hubiera llamado con un solo elemento.

Una última apreciación consiste en que si se desea extraer datos de un fichero XML, las funciones cambiarán su prefijo de *html* a *xml*.

4.4.5. Obtener una tabla

En ciertas ocasiones se presentará la necesidad de obtener información de una tabla, por lo que no valdrán las funciones de tratamiento de nodos anteriormente vistas. Para ello existe la función html_table() de la que se van a explicar sus argumentos a continuación y de la que se puede ver un ejemplo de su sintaxis en el código fuente 26 [75]. Esta función lo que hace básicamente es convertir la tabla o tablas extraídas como nodos con las funciones html_node() o html_nodes() en un *data frame*, de tal manera que se pueden manejar los datos contenidos en dicha tabla de forma sencilla (limpiarlos, operar con ellos, etcétera) [75].

```
1 html_table(x, header = NA, trim = TRUE, fill = FALSE, dec = ".")
```

Código fuente 26: Extracción de tabla con *rvest*

Argumentos

`x`	Un documento, un conjunto de nodos o bien un solo nodo.
`header`	Parámetro de tipo booleano que permite determinar si se utiliza la primera fila como encabezado de tabla (valor verdadero) o no. En caso de que se indique el valor indefinido (NA), se utilizará la primera fila como encabezados solamente si está formada por etiquetas de celda de encabezado
`trim`	Parámetro booleano en el que si se indica como cierto en la invocación, elimina los espacios en blanco sobrantes al principio y al final de la cadena.
`fill`	Parámetro booleano que si toma valor cierto rellena las filas que tienen menos número de columnas que el máximo con NA's. La idea de esto es que las tablas sean cuadradas, esto es, que todas las filas tengan el mismo número de celdas.
`dec`	Parámetro que permite establecer el separador de decimales a utilizar en caso de que se detecten cadenas que representen números decimales.

Precondiciones

La función html_table() tiene una serie de precondiciones (asunciones según el autor) que hay que tener en cuenta para utilizarla y que su comportamiento sea el adecuado [75]:

- No hay uniones de celdas entre filas (esto es, ninguna celda ocupa más de dos filas).
- Los encabezados de las columnas están en la primera fila.

4.4.6. Trabajando con formularios: parsing previo

Para trabajar con formularios, es necesario parsearlos previamente, esto es, utilizar una función en la que pasado un nodo cuya raíz es un elemento formulario *form* (obtenido previamente con html_node() o html_nodes()) analice no solo como un nodo HTML dicho formulario, si no que internamente pueda tener en cuenta todos los elementos con los que se puede interactuar, para luego hacer uso de ellos con funciones que se utilizarán con posterioridad.

La función que realiza esta labor se denomina html_form() [76]. En el código fuente 27 se puede observar un ejemplo de su uso y tras él, se explican sus argumentos.

```
1 html_form(x)
```

Código fuente 27: Parsing inicial de formulario con *rvest*

Argumentos

`x`	Un documento, un conjunto de nodos o bien un solo nodo. En este caso, el nodo que se transfiera debe ser uno solo y debe ser un elemento *form*.

4.4.7. Trabajando con formularios: completar campos

Una vez que ya se ha *parseado* previamente el formulario, antes de realizar su envío, hay que completar dicho formulario rellenando y seleccionando los diversos campos que pudiera tener el formulario (en caso de que los hubiera y/o se requiriera rellenarlos).

Ese es el objetivo de la función set_values(). [77]. En el código fuente 28 se puede observar un ejemplo de su uso y tras él, se explican sus argumentos.

```
1 set_values(form, ...)
```

Código fuente 28: Completar campos de formulario con *rvest*

Argumentos

Esta función puede ser un poco más compleja de utilizar, ya que es de las pocas de *rvest* en las que su *aridad* puede ser infinita (en concreto de 2 a n).

4.4.8. Trabajando con formularios: enviar formulario

Para finalizar el posible tratamiento del formulario, tras *parsearlo* con html_form() y completarlo con set_values(), hay que enviar dicho formulario con submit_form(). Esta última función se explica de forma un poco más profusa viendo un ejemplo

`form`	Formulario a modificar (obtenido previamente con html_form().
`...`	Pares nombre-valor de los campos que se desean modificar.

de uso en el código fuente 29 y viendo sus argumentos inmediatamente después [78].

```
1 submit_form(session, form, submit = NULL, ...)
```

Código fuente 29: Envío de formulario con *rvest*

Argumentos

`session`	Sesión a través de la que enviar el formulario. El concepto de sesión aún no se ha visto en *rvest*, pero a continuación se podrá observar. Este parámetro es opcional por la experiencia que se ha tenido al utilizarla.
`form`	Formulario que se desea enviar.
`submit`	Nombre del botón al que se desea invocar para realizar el envío. Si no se suple, se utilizará por defecto el primer botón destinado al envío que se encuentre en el formulario.
`...`	Argumentos adicionales que se deseen enviar ya sea mediante método GET o POST.

Si se logra enviar el formulario correctamente, se obtiene la respuesta HTML parseada. En caso de que haya un error en la petición se lanzará un error.

4.4.9. Trabajando con sesiones: simular una sesión

Cuando se navega por Internet, en ciertas ocasiones se almacena el estado de la navegación (cuando se hace una autenticación o un filtro de búsqueda). Este almacenamiento se produce en el servidor y se identifica el cliente gracias a un identificador de sesión almacenado en una *cookie*. Por defecto (tal y como se verá en los ejercicios que se realizarán con posterioridad) *rvest* no lleva la gestión de las sesiones correctamente, ya que no guarda el identificador y no lo gestiona correctamente (no lo envía en la cabecera). Para ello, el autor ha previsto una serie de funciones que permiten prever esta eventualidad. La primera de ellas es la función html_session() que genera un objeto que indicado a las funciones de navegación que se expondrán a continuación, permite simular una sesión [79]. Además existe la función is_session() que permite verificar si para un objeto

dado, es una sesión o no lo es. En el código fuente 30 se puede observar el orden de los parámetros y a continuación una descripción de los mismos. Hay que tener en cuenta que el propio autor indica que estas funcionalidades de sesión están en desarrollo por lo que agradece el *feed back* que se le pueda hacer llegar [25][26].

```
1 html_session(url, ...)
2
3 is.session(x)
```

Código fuente 30: Simular una sesión con *rvest*

Argumentos

`url`	Ubicación desde la que se quiere iniciar la sesión.
`...`	Cualquier configuración orientada a textithttr adicional que se desee usar durante la sesión.
`x`	Objeto que se desea verificar para ver si es una sesión.

Métodos para trabajar con una sesión

Un objeto de sesión puede ser manipulado combinando métodos de *httr* y métodos HTML [79]:

- Hay que utilizar las funciones cookies(), headers() y status_code() para acceder a las propiedades de la petición [70].
- La función html_nodes() permite acceder a los nodos deseados (igual que si no se utilizara una sesión, salvo que se proporciona la sesión en vez del documento como primer parámetro) [73].

4.4.10. Trabajando con sesiones: saltar a otra página

Una vez se ha creado la sesión, para navegar entre páginas y poder hacerlo manteniendo, no hay que utilizar read_html(), sino que hay que utilizar las funciones jump_to() y follow_link(). En concreto, la primera de las opciones permite ir a la dirección indicada por parámetro, mientras que la segunda permite ir a una dirección de un enlace (elemento ¡a¿) seleccionado en la página actual de la sesión mediante un selector (ya sea CSS o XPath) [80][81].

En el código fuente 31 se puede observar el orden de los parámetros y a continuación la descripción de los parámetros para ambas funciones.

```
1 jump_to(x, url, ...)
2
3 follow_link(x, i, css, xpath, ...)
```

Código fuente 31: Navegar manteniendo una sesión con *rvest*

Argumentos

`x`	Una sesión previamente creada.
`url`	Una dirección web, ya sea absoluta o relativa, para acceder a ella navegando.
`...`	Cualquier configuración adicional de *httr* que aplicar a esta petición (indicación de parámetros por POST, cookies, etcétera).
`i`	Este parámetro puede ser (en caso de que se utilice, es opcional): ▪ Un entero, que permite navegar a través de la URL del *iésimo* enlace que hay en la página actual ▪ Una cadena, que permite navegar a través de la URL del primer enlace cuyo texto coincide con la cadena suministrada.
`css`	Selector CSS (opcional) a utilizar para seleccionar el enlace que se desea utilizar para navegar a través de su URL. Este parámetro es excluyente con un selector XPath.
`xpath`	Selector XPath 1.0 (opcional) a utilizar para seleccionar el enlace que se desea utilizar para navegar a través de su URL. Este parámetro es excluyente con un selector CSS.

4.4.11. Trabajando con sesiones: gestionar el historial de la sesión

Para finalizar, se indican las dos últimas propuestas del autor para el uso de sesiones: consultar el historial de navegación a través de la sesión con la función session_history() y volver a la página visitada anteriormente con back() [82]. En ambas funciones se requiere simplemente el objeto de sesión tal y como se puede observar en el código fuente 32 y en los argumentos indicados a continuación.

```
session_history(x)

back(x)
```

Código fuente 32: Historial de sesión y vuelta atrás con *rvest*

Argumentos

`x`	Una sesión previamente creada.

4.4.12. Resolución de errores de codificación

En los supuestos prácticos que se han desarrollado no será necesario el uso de esta función, pero es posible que se encuentre con páginas que declaren una codificación y utilicen caracteres que pertenezcan a otra codificación, por lo que dichos caracteres no se representarían bien al utilizar la función de descarga read_html(). Para resolver esta eventualidad se pueden utilizar las funciones guess_encoding() y repair_encoding() [83]. La diferencia entre ambas es que la primera es de carácter preventivo mientras que la segunda es de caracter correctivo. Así pues, si se detecta que se ha descargado una página en la que los caracteres que aparecen son extraños, se puede utilizar guess_encoding() para saber la codificación real y tomar de las posibles la que mayor confianza aporte (aparece una tabla con las posibles codificaciones y su nivel confianza) y volver a descargar dicha página indicando a read_html() la codificación correcta, o bien tomar esa codificación de mayor confianza e invocar repair_encoding() indicándola junto al objeto que contiene la página descargada, de tal forma que intente arreglar la codificación errónea. A continuación se puede observar en el código fuente 33 el uso de ambas funciones y la descripción de los argumentos requeridos.

```
guess_encoding(x)

repair_encoding(x, from = NULL)
```

Código fuente 33: Resolver problemas de codificación con *rvest*

Argumentos

`x`	Un vector de caracteres (la salida de html_read() normalmente)
`from`	La codificación realmente tiene la cadena. En caso de valor nulo, el autor no indica nada [83], pero se podría deducir con el contexto (no se ha probado) que lo que hará será intentar hacer una reparación deduciendo internamente la codificación.

Precondición

Estas funciones son *wrappers* que utilizan herramientas del paquete *stringi*. Por lo tanto para poder utilizar estas herramientas es necesario que esté instalada previamente.

4.4.13. Miscelánea: extraer un elemento de una lista

Para extraer un elemento de una lista, se puede utilizar [[ ]]. No obstante, el autor ha creado una pequeña función denominada pluck() [84]. El uso de esa función se explica en el código fuente 34 y en los argumentos que aparecen a continuación.

```
1 pluck(x, i, type)
```

Código fuente 34: Extraer elementos de una lista con *rvest*

Argumentos

`x`	Una lista.
`i`	Una cadena o un entero.
`type`	Tipo del elemento a devolver, si se conoce.

4.5. Realización del test de propósito general con rvest

4.5.1. Ejemplo de petición HTTP

En primer lugar hay que descargar la página de la que se desean extraer los datos y crear la estructura que permita recorrer el DOM. Esto se puede hacer

con la misma biblioteca *rvest*, en particular con la función `read_html()` [72], que entre las múltiples posibilidades, recibe la URL por parámetro, tal y como se muestra en el código fuente 35, donde se obtiene la página para imprimirla en pantalla a continuación.

```
1 library(rvest)
2
3 url <- "http://tfg.daniel-fl.com/general/"
4
5 page <- read_html(url)
6
7 print(page)
```

Código fuente 35: Ejercicio de petición HTTP con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
>
> page <- read_html(url)
>
> print(page)
{xml_document}
<html lang="es">
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 .$
[2] <body>\n        <div id="wrapper">\n            <header><h1><a href="../" .$
>
```

Ilustración 4.4: Resultado de ejecución del ejercicio de petición HTTP con *rvest*

Al aplicar la función de impresión sobre la estructura devuelta, se observa en la ilustración 4.4 que es un objeto de tipo documento XML, en el que hay dos nodos hijo (que corresponden a *head* y a *body*).

4.5.2. Obtener metaetiqueta del título de la página, descripción y palabras clave

Para realizar este ejercicio, se va a partir del primer ejercicio de *rvest* y se va a utilizar además el operador tubería. El objetivo es obtener el elemento *title* de la página así como las metaetiquetas de descripción y de palabras clave.

Para ello se hará uso de la función `html_node()` [73] que permite obtener el primer nodo que cumple un patrón determinado. Dicho patrón es determinado por un selector CSS (podría ser XPath sí así se deseara) para cada uno de los elementos que se desea obtener. En el caso del título, el selector es fácil, ya que solamente puede haber un elemento *title* en un documento HTML. Sin embargo,

en el caso de las palabras clave o de la descripción, ambas utilizan una etiqueta o elemento *meta*. En el caso de estas etiquetas, ambas se diferencian en el atributo *name* que tienen. Así pues, solamente habrá una para palabras clave y otra para la descripción. Por esta razón se utiliza un selector CSS más concreto. Además, se utiliza tras obtener cada uno de estos nodos la funcion `html_attr()` ya que la información en el caso de las etiquetas no se guarda en el cuerpo (las metaetiquetas no llevan cuerpo), si no que almacenan la información en el atributo *content*.

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/general/"
4
5  page <- read_html(url)
6
7  #Se imprime el título
8  titulo <- page %>% html_node("title") %>% html_text
9  print(titulo)
10
11 #Se imprime la descripción:
12 descr <- page %>% html_node("meta[name=description]") %>%
   ↪  html_attr("content")
13 print(descr)
14
15 #Se imprimen las palabras clave:
16 kw <- page %>% html_node("meta[name=keywords]") %>%
   ↪  html_attr("content") %>% strsplit(", ")
17 print(kw)
```

Código fuente 36: Ejercicio de obtención de metaetiquetas con *rvest*

Tal y como se puede observar en el código fuente 36, así como en la ilustración 4.5, se ha utilizado además la función `html_text()` [74] para el título, ya que permite extraer el texto del nodo o nodos seleccionados (el contenido propiamente dicho). Adicionalmente se ha utilizado el operador tubería [85]. Por último, cabe destacar, que se ha hecho una pequeña operación adicional con la metaetiqueta de palabras clave (ya que éstas vienen separadas por punto y coma en el atributo *content*) con el fin de separarlas convenientemente (por si en un futuro se quisieran tratar o almacenar de forma desagregada).

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
>
> page <- read_html(url)
>
> #Se imprime el título
> titulo <- page %>% html_node("title") %>% html_text
> print(titulo)
[1] "Tests de propósito general | Daniel Francisco López"
>
> #Se imprime la descripción:
> descr <- page %>% html_node("meta[name=description]") %>% html_attr("content")
> print(descr)
[1] "Tests para hacer diversas pruebas de selectores, para ver las posibilidades de acceder al DOM"
>
> #Se imprimen las palabras clave:
> kw <- page %>% html_node("meta[name=keywords]") %>% html_attr("content") %>% strsplit(", ")
> print(kw)
[[1]]
[1] "web"       "scrapping" "utilidad"  "CSS"       "XPath"     "DOM"

>
```

Ilustración 4.5: Resultado de ejecución del ejercicio de obtención de metaetiquetas con *rvest*

4.5.3. Obtener todos los enlaces de la barra de navegación

A continuación se obtienen todos los elementos enlace que hay en la barra de navegación (que en HTML5 se denota con *nav*). Para ello se utiliza nuevamente CSS para acceder a los elementos enlace (*a*) que estén en cualquier profundidad dentro de la barra de enlace (*nav*).

Como se puede observar en el código fuente 37, se obtienen en primer lugar todos los nodos, haciendo uso de la variante plural de `html_node()`, que es `html_nodes()`, que permite obtener todas las apariciones que coinciden con el patrón CSS indicado a partir del nodo sobre el que se aplica y no solamente con la primera aparición encontrada [73]. A continuación se extraen en primer lugar los textos de los enlaces con la función `html_text()` [74] y por el otro lado se obtienen los enlaces propiamente dichos obteniendo el atributo *href* con la función `html_attr()` [74], que es el que indica el estándar HTML que contiene la URL del enlace. Por último, se crea un *data frame*, que es una estructura de datos de R que permite almacenar datos de forma bidimensional sin necesidad de que el tipo de datos de las columnas coincidan [86]. Esto se realizará uniendo los vectores de textos (lo que pone en el enlace) y direcciones (dónde lleva el enlace). La ejecución sucede como se puede observar en la ilustración 4.6.

Además, gracias al uso de la función `View()` aplicada al *data frame* se imprime en forma tabulada dichas direcciones, que se muestra en la ilustración 4.7.

```r
library(rvest)

url <- "http://tfg.daniel-fl.com/general/"
#Se obtienen los enlaces que estan dentro de la barra de
 ↪  navegacion
enlaces <- read_html(url) %>% html_nodes("nav a")

#Ahora se obtiene por un lado los textos de los enlaces...

textos <- enlaces %>% html_text

# y por el otro las direcciones

direcciones <- enlaces %>% html_attr("href")

#Se introduce en un dataframe y se muestra en pantalla

tabla <- data.frame(textos, direcciones)

View(tabla)
```

Código fuente 37: Ejercicio de obtención de enlaces de navegación con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
> #Se obtienen los enlaces que estan dentro de la barra de navegacion
> enlaces <- read_html(url) %>% html_nodes("nav a")
>
> #Ahora se obtiene por un lado los textos de los enlaces...
>
> textos <- enlaces %>% html_text
>
> # y por el otro las direcciones
>
> direcciones <- enlaces %>% html_attr("href")
>
> #Se introduce en un dataframe y se muestra en pantalla
>
> tabla <- data.frame(textos, direcciones)
>
> View(tabla)
>
```

Ilustración 4.6: Resultado de ejecución del ejercicio de obtención de enlaces de navegación con *rvest*

	textos	direcciones
1	Propósito general	#
2	Envío de formularios	../forms
3	Gestión de cookies de sesión	../cookies
4	Renderizado de JavaScript	../test-js
5	Performance	../performance

Ilustración 4.7: Tabla de ejecución del ejercicio de obtención de enlaces de navegación con *rvest*

4.5.4. Obtener enlaces por su clase y por su identificador

El objetivo de este ejercicio es hacer dos tipos de obtención de enlaces. En el cuerpo del test general se han puesto enlaces identificados por clase e identificados por identificador.

En primer lugar se va a hacer la obtención de enlaces por identificador, pudiéndose observar dicho proceso en el código fuente 38 y su resultado de ejecución en la ilustración 4.8. Para ello, se utiliza la función `html_nodes()` vista con anterioridad [73] en la que se van a utilizar los selectores CSS de identificador [87]. Tras ello, se obtienen tres listas, la primera que contiene el nombre del identificador asignado a cada enlace (obtenido como atributo *id*, con la función `html_attr()` [74]), la segunda que indica el nombre del enlace y la tercera que indica la dirección del enlace (atributo *href*). Para terminar se utilizan las tres listas para generar un *data frame* que se puede observar en la ilustración 4.9.

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
>
> #Se obtienen los enlaces obtenidos por id
> enlaces <- read_html(url) %>% html_nodes("#link_seccion_formularios, #link_seccion_js")
>
> #Ahora se obtiene por un lado los identificadores, por otros los textos de los enlaces y por ultimo las direcciones
>
> ids <- enlaces %>% html_attr("id")
> textos <- enlaces %>% html_text
> direcciones <- enlaces %>% html_attr("href")
>
> #Se introduce en un dataframe y se muestra en pantalla
>
> tabla <- data.frame(ids, textos, direcciones)
> View(tabla)
>
```

Ilustración 4.8: Resultado de ejecución del ejercicio de obtención de enlaces por su identificador con *rvest*

En segundo lugar, tal y como se puede observar en el código fuente 39, se repite el mismo proceso pero esta vez con los identificadores de clase. Para

```r
library(rvest)

url <- "http://tfg.daniel-fl.com/general/"

#Se obtienen los enlaces obtenidos por id
enlaces <- read_html(url) %>%
 ↪  html_nodes("#link_seccion_formularios, #link_seccion_js")

#Ahora se obtiene por un lado los identificadores, por otros los
 ↪  textos de los enlaces y por ultimo las direcciones

ids <- enlaces %>% html_attr("id")
textos <- enlaces %>% html_text
direcciones <- enlaces %>% html_attr("href")

#Se introduce en un dataframe y se muestra en pantalla

tabla <- data.frame(ids, textos, direcciones)
View(tabla)
```

Código fuente 38: Ejercicio de obtención de enlaces por su identificador con *rvest*

	ids	textos	direcciones
1	link_seccion_formularios	Envío de formularios	../forms
2	link_seccion_js	Renderizado de JavaScript	../test-js

Ilustración 4.9: Tabla de ejecución del ejercicio de obtención de enlaces por su identificador con *rvest*

ello se modifican los identificadores de id por identificadores de clase [87] y se modifica el selector de atributo de la primera columna de *id* a *class*. El proceso de ejecución se puede observar en la ilustración 4.10 y la tabla resultante en la ilustración 4.11.

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/general/"
4
5  #Se obtienen los enlaces obtenidos por clase
6  enlaces <- read_html(url) %>% html_nodes(".dificultad_media,
   ↪  .dificultad_alta")
7
8  #Ahora se obtiene por un lado la clase a la que pertenece el
   ↪  enlace, por otros los textos de los enlaces y por ultimo las
   ↪  direcciones
9
10 clases <- enlaces %>% html_attr("class")
11 textos <- enlaces %>% html_text
12 direcciones <- enlaces %>% html_attr("href")
13
14 #Se introduce en un dataframe y se muestra en pantalla
15
16 tabla <- data.frame(clases, textos, direcciones)
17 View(tabla)
```

Código fuente 39: Ejercicio de obtención de enlaces por su clase con *rvest*

```
> url <- "http://tfg.daniel-fl.com/general/"
>
> #Se obtienen los enlaces obtenidos por id
> enlaces <- read_html(url) %>% html_nodes(".dificultad_media, .dificultad_alta")
>
> #Ahora se obtiene por un lado la clase a la que pertenece el enlace, por otros los textos de los enlaces y por ultim?
>
> clases <- enlaces %>% html_attr("class")
> textos <- enlaces %>% html_text
> direcciones <- enlaces %>% html_attr("href")
>
> #Se introduce en un dataframe y se muestra en pantalla
>
> tabla <- data.frame(clases, textos, direcciones)
> View(tabla)
>
```

Ilustración 4.10: Resultado de ejecución del ejercicio de obtención de enlaces por su clase con *rvest*

Por último, cabe destacar que esto sólo se puede realizar con elementos enlace de los que se sabe que tienen el identificador o clase particular indicado en el ejercicio.

	clases	textos	direcciones
1	dificultad_media	Envío de formularios	../forms
2	dificultad_media	Gestión de cookies de sesión	../cookies
3	dificultad_alta	Renderizado de JavaScript	../test-js
4	dificultad_alta	Performance	../performance

Ilustración 4.11: Tabla de ejecución del ejercicio de obtención de enlaces por su clase con *rvest*

4.5.5. Obtener los atributos de una imagen

Se van a obtener los atributos básicos de una imagen. Una imagen siempre tiene un atributo de ubicación (*src*). Además, es aconsejable que tenga un texto alternativo (atributo *alt*).

Siguiendo esta premisa, es como se ha hecho el ejemplo. Utilizando la función `html_attrs()` [74] se han introducido en un *data frame* todos y cada uno de los atributos que tiene la imagen. Esto se hace así para garantizar que se obtienen todos los atributos. Se puede obtener un solo atributo con la función `html_attr()`, indicando por parámetro el nombre del atributo, o bien, obtener todos en una tabla, que es lo que finalmente se ha realizado.

El código fuente 40 detalla en detalle la obtención de estos atributos. El resultado de la ejecución se puede observar en la ilustración 4.12 y el *data frame* con los atributos de la imagen en la ilustración 4.13.

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
>
> #Se obtiene la imagen (ya se sabe que el identificador es imagen)
> imagen <- read_html(url) %>% html_node("#imagen")
>
> #Ahora se obtienen todos los atributos de la imagen
>
> atributos <- imagen %>% html_attrs()
>
> #Y se introducen en un data frame
>
> tabla <- data.frame(atributos)
> View(tabla)
>
```

Ilustración 4.12: Resultado de ejecución del ejercicio de obtención de los atributos de una imagen con *rvest*

```r
library(rvest)

url <- "http://tfg.daniel-fl.com/general/"

#Se obtiene la imagen (ya se sabe que el identificador es
↪  imagen)
imagen <- read_html(url) %>% html_node("#imagen")

#Ahora se obtienen todos los atributos de la imagen

atributos <- imagen %>% html_attrs()

#Y se introducen en un data frame

tabla <- data.frame(atributos)
View(tabla)
```

Código fuente 40: Ejercicio de obtención de los atributos de una imagen con *rvest*

	row.names	atributos
1	id	imagen
2	src	pics/original-compressor.jpeg
3	alt	Fotografía de carretera convencional

Ilustración 4.13: Tabla con resultado de ejecución del ejercicio de obtención de los atributos de una imagen con *rvest*

4.5.6. Obtener diversas etiquetas de una clase y diferenciarlas

A la hora de crear el test de propósito general, se han creado algunos elementos HTML en los párrafos con una coloración roja (que da lugar a la interpretación de que son elementos importantes). Dicha coloración roja se establece mediante una clase denominada ïmportante".

El objetivo es extraer todos los elementos que tengan esta clase, indicando a cual de los siguientes tres apartados corresponde: uno primero, con los elementos en línea *strong* que es un indicador de que es un texto al que se le quiere dar relevancia según los estándares del W3C [68], esto es, un texto importante; un segundo apartado en donde la clase es aplicada a un enlace (*a*) y un tercer apartado a modo de cajón de sastre donde se incluyen los elementos que no están bajo el paraguas de los dos apartados anteriores.

Para obtener todos los elementos con la clase *importante*, se utiliza junto a la función `html_nodes()`, la función `html_name()` que devuelve el tipo de elemento (esto es, el nombre) sobre el que se ejecuta la función [74].

La resolución de este ejercicio se muestra en el código fuente 41. La salida producida por la ejecución de dicho código fuente se puede observar en la ilustración 4.14, donde se puede observar que se han localizado dos elementos con la clase ïmportante", de los cuales uno es un enlace a una página del sitio web referida a *performance*, y la otra simplemente es un texto que se ha deseado destacar.

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/general/"
4
5  #Se obtienen los elementos de importancia
6  importantes <- read_html(url) %>% html_nodes(".importante")
7
8  for(imp in importantes){
9          tag <- imp %>% html_name()
10         if(tag == "a"){
11                 print("ENLACE-------------------------")
12                 print(paste("    contenido: ",imp %>%
                    ↪  html_text()))
13                 print(paste("    enlace: ", imp %>%
                    ↪  html_attr("href")))
14         }else if(tag == "strong"){
15                 print("STRONG-------------------------")
16                 print(paste("    contenido: ",imp %>%
                    ↪  html_text()))
17         }else{
18                 print(paste("No es enlace ni strong. Es un
                    ↪  elemento", tag))
19         }
20 }
```

Código fuente 41: Ejercicio de obtención de diversas etiquetas/elementos de una clase y su diferenciación con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
>
> #Se obtienen los elementos de importancia
> importantes <- read_html(url) %>% html_nodes(".importante")
>
> for(imp in importantes){
+ tag <- imp %>% html_name()
+ if(tag == "a"){
+ print("ENLACE-------------------------")
+ print(paste("    contenido: ",imp %>% html_text()))
+ print(paste("    enlace: ", imp %>% html_attr("href")))
+ }else if(tag == "strong"){
+ print("STRONG-------------------------")
+ print(paste("    contenido: ",imp %>% html_text()))
+ }else{
+ print(paste("No es enlace ni strong. Es un elemento", tag))
+ }
+ }
[1] "STRONG-------------------------"
[1] "    contenido:  esto es un texto importante"
[1] "ENLACE-------------------------"
[1] "    contenido:  esto es un enlace importante relativo al performance"
[1] "    enlace:  ../performance"
>
```

Ilustración 4.14: Resultado de ejecución de ejercicio de obtención de diversas etiquetas/elementos de una clase y su diferenciación con *rvest*

4.5.7. Obtener para cada enlace del cuerpo, a qué párrafo pertenece

Para realizar este ejemplo, se podrían adoptar las dos siguientes vías:

1. Recorrer todos los párrafos e iterarlos manualmente en busca de enlaces en ellos.

2. Recorrer todos los enlaces y acceder al elemento que lo contiene (párrafos).

Se ha optado por esta segunda estrategia. No obstante tiene un problema que no es menor. Dicho problema estriba en que los selectores CSS que se vienen utilizando no tienen en su conjunto selectores en los que dado un elemento concreto se acceda a su antecesor. PAra resolver esta eventualidad, se hará uso de *XPath*, que aunque es más engorroso, permite salvar este obstáculo mediante su selector de padre (*parent* o *..*).

El proceso que se realiza consiste en obtener en primer lugar los enlaces con la función `html_nodes()` y con un selector CSS que obtiene todos los enlaces que hay en el cuerpo (denotado con la etiqueta HTML5 `main`. A continuación se opta por acceder a través de los nodos y utilizando la misma función `html_nodes()` pero asignando el selector mediante el argumento denominado *xpath* [73].

Además, con los enlaces se utiliza una función adicional `url_absolute()` que está contenida en el paquete *xml2*. Esta función transforma las direcciones URL relativas a absolutas. Esto se hace pasando como primer parámetro la dirección a transformar y como segundo parámetro la dirección base [88].

Todo lo mencionado anteriormente finalmente se introduce en una tabla y se muestra tal y como se puede observar en el código fuente 42. El resultado de la ejecución se puede observar en la ilustración 4.15 y la tabla que finalmente se genera se puede observar en la ilustración 4.16.

```r
library(rvest)

url <- "http://tfg.daniel-fl.com/general/"
pagina <- read_html(url)

#Se obtienen los enlaces del cuerpo

        enlaces <- pagina %>%
                     html_nodes("main a")

        urls <- enlaces %>%
                  html_attr("href")

        urlsAbsolutas <- urls %>% url_absolute(url)

#Se obtienen los padres (parrafos) con XPath
#ya que con CSS no se puede ascender en el DOM

        parrafos <- enlaces %>%
                        html_nodes(xpath="..") %>%
                        html_text

#Se genera la tabla con enlaces y sus parrafos

tabla <- data.frame(urls, urlsAbsolutas, parrafos)

View(tabla)
```

Código fuente 42: Ejercicio de obtención de enlaces y párrafos con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
> pagina <- read_html(url)
>
> #Se obtienen los enlaces del cuerpo
>
> enlaces <- pagina %>%
+      html_nodes("main a")
>
> urls <- enlaces %>%
+   html_attr("href")
>
> urlsAbsolutas <- urls %>% url_absolute(url)
>
> #Se obtienen los padres (parrafos) con XPath
> #ya que con CSS no se puede ascender en el DOM
>
> parrafos <- enlaces %>%
+ html_nodes(xpath="..") %>%
+ html_text
>
> #Se genera la tabla con enlaces y sus parrafos
>
> tabla <- data.frame(urls, urlsAbsolutas, parrafos)
>
> View(tabla)
>
```

Ilustración 4.15: Resultado de ejecución del ejercicio de obtención de enlaces y párrafos con *rvest*

	urls	urlsAbsolutas	parrafos
1	../performance	http://tfg.daniel-fl.com/performance	Mientras que esto es un texto importante que tien>
2	../forms	http://tfg.daniel-fl.com/forms	Tras la finalización de este supuesto, se debe em>

Ilustración 4.16: Tabla con resultado de ejecución del ejercicio de obtención de enlaces y párrafos con *rvest*

4.5.8. Ver los enlaces hermanos de los textos importantes

Este es el último ejercicio donde se practica el uso de la función de selección `html_nodes()` [73]. Ya que CSS no provee ningún operador para elegir dos hermanos de distinto tipo no adyacentes dentro del mismo padre (se puede utilizar el selector strong + a en caso de que sean adyacentes), el selector se deberá construir irremediablemente con XPath.

Como se puede observar en el código fuente 43, para empezar se hace la petición de la página tal y como se venía haciendo en ejercicios anteriores con la función `read_html()`[72]. Acto seguido se utiliza la función `html_nodes()` para obtener todos los nodos que contengan al menos un nodo strong y un nodo a, de tal manera que ambos serán hermanos [73]. Para ello se hace uso de un selector XPath, ya que CSS no provee este tipo de selector.

Una vez realizada la selección, se itera la lista de nodos generada para indicar por cada nodo la siguiente información:

- El texto del nodo padre común a los enlaces y a los textos importantes. Para ello se hace uso de la función `html_text()`[74].
- El texto de cada uno de los textos importantes. Para ello se hace uso nuevamente de la función `html_nodes()` para seleccionarlos (podría haber más de uno, aunque en el ejemplo solamente hay uno) y a continuación se extrae su texto con `html_text()`.
- El texto y la dirección URL de cada uno de los enlaces que aparecen en el padre (podría haber de 1 a n, por lo que es conveniente utilizar la función plural `html_nodes()`. Luego este problema se divide en dos: obtener por un lado para cada enlace su texto (con `html_text()` y por el otro obtener su enlace accediendo al atributo que lo contiene (href) con la función `html_attr()` [74].

El resultado de la ejecución del código fuente 43 se puede observar en la ilustración 4.17.

```r
library(rvest)

#Se hace la peticion de la pagina
url <- "http://tfg.daniel-fl.com/general/"
pagina <- read_html(url)

#Se extraen aquellos padres que tengan como hijos a la vez nodos importantes y nodos enlaces
padres <- pagina %>% html_nodes(xpath="//*[strong][a]")

#Para cada nodo de los seleccionados,
#se extrae su contenido y luego las pertes de enlaces y textos importantes
for(padre in padres){

        print(paste("+Padre: ", padre %>% html_text))

        print("-Textos importantes:")
        print(padre %>% html_nodes("strong") %>% html_text)

        print("-Enlaces:")
        for(enlace in padre %>% html_nodes("a")){
                print(paste(enlace %>% html_text, " - ", enlace %>% html_attr("href")))
        }
}
```

Código fuente 43: Ejercicio de obtención de de enlaces hermanos de textos importantes con *rvest*

```
> library(rvest)
>
> #Se hace la peticion de la pagina
> url <- "http://tfg.daniel-fl.com/general/"
> pagina <- read_html(url)
>
> #Se extraen aquellos padres que tengan como hijos a la vez nodos importantes $
> padres <- pagina %>% html_nodes(xpath="//*[strong][a]")
>
> #Para cada nodo de los seleccionados,
> #se extrae su contenido y luego las pertes de enlaces y textos importantes
> for(padre in padres){
+
+ print(paste("+Padre: ", padre %>% html_text))
+
+ print("-Textos importantes:")
+ print(padre %>% html_nodes("strong") %>% html_text)
+
+ print("-Enlaces:")
+ for(enlace in padre %>% html_nodes("a")){
+ print(paste(enlace %>% html_text, " - ", enlace %>% html_attr("href")))
+ }
+ }
[1] "+Padre:  Mientras que esto es un texto importante que tiene una clase impo$
[1] "-Textos importantes:"
[1] "esto es un texto importante"
[1] "-Enlaces:"
[1] "esto es un enlace importante relativo al performance  -  ../performance"
```

Ilustración 4.17: Resultado de ejecución de ejercicio de obtención de de enlaces hermanos de textos importantes con *rvest*

4.5.9. Obtener la tabla que hay en el cuerpo

A continuación se va a extraer de la web diseñada para el test de propósito general una tabla (la única que hay). Para ello, se va a hacer uso de la función `html_table()` [75] que provee *rvest* aplicada sobre un elemento tabla, que se debe obtener con la función `html_node()` [73].

Como solo hay una tabla, dicho elemento se puede extraer directamente indicando el nombre de la etiqueta/elemento (`<table>`). En caso de ser varias tablas, en caso de desear extraer una, habría que utilizar un selector más concreto con `html_node()` o bien traer todas las tablas con `html_nodes()` y extraer la tabla con un índice, o bien aplicar directamente la función de extracción de tabla (tras lo que se generarían varios *data frames*.

Como se ha indicado anteriormente, se extrae una tabla, indicando los títulos de las columnas en sus dimensiones (dichos títulos, se han definido en la tabla con celdas `<th>`, por lo que R los reconocerá automáticamente como encabezados [75].

El código fuente 44 muestra cómo se obtiene dicha tabla, generando un *data frame* que es posteriomente visualizado. El contenido del *data frame* se puede observar en las ilustraciones 4.19 y 4.20, mientras que el resultado de la ejecución (no devuelve nada más allá que la impresión de los frames de datos) en la ilustración 4.18.

```
1 library(rvest)
2
3 url <- "http://tfg.daniel-fl.com/general/"
4
5 #Se obtiene la tabla
6 tabla <- read_html(url) %>% html_node("table") %>% html_table()
7
8 View(tabla)
```

Código fuente 44: Ejercicio de obtención de tabla con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/general/"
>
> #Se obtiene la tabla
> tabla <- read_html(url) %>% html_node("table") %>% html_table()
>
> View(tabla)
>
```

Ilustración 4.18: Resultado de ejecución de ejercicio de obtención de tabla con *rvest*

	Lunes	Martes	Miércoles
1	Ensalada	Paella	Sopa de arroz
2	Pasta con tomate	Guisantes	Albóndigas
3	Lentejas con chorizo	Plato combinado	Croquetas
4	Lomo con patatas	Redondo de ternera	Zarzuela de pescado

Ilustración 4.19: Tabla de ejecución del ejercicio de obtención de tabla con *rvest* (I)

	Miércoles	Jueves	Viernes
1	Sopa de arroz	Espaguetis a la carbonara	Judías con jamón
2	Albóndigas	Arroz con conejo	Cocido completo
3	Croquetas	Fabada	Pescado
4	Zarzuela de pescado	Guisado de costillas	Croquetas

Ilustración 4.20: Tabla de ejecución del ejercicio de obtención de tabla con *rvest* (II)

4.6. Realización del test de envío de formularios con rvest

El objetivo de este ejercicio es verificar cómo actúa *rvest* cuando se desea obtener información procesada por el servidor en base a datos enviados previamente al servidor, esto es, verificar si *rvest* permite enviar formularios y obtener la respuesta generada en base al envío realizado y *parsearla*.

Este objetivo se consigue utilizando las funciones que *rvest* provee para la gestión de formularios [76][77][78][26][25].

Para empezar, hay que crear un objeto de sesión con `html_session()` [79], que posteriormente será utilizado para enviar el formulario. A continuación se extrae el nodo que contiene el formulario con la función `html_node()` [73] y por último procesarlo con la función `html_form()` [76]. Como solamente existe un formulario, se puede utilizar el selector CSS de elemento `form`. Si hubiera más, simplemente habría que seleccionarlo por el identificador, clase o selector CSS o XPath o bien traerlos todos con `html_nodes()` y elegir mediante su ordinal de aparición en la página.

A continuación se establece el valor de todos los elementos del formulario que se desean rellenar con la función `set_values()` [77]. Como el único componente que contiene el formulario es una caja de texto para el nombre, se pasa simplemente una pareja de nombre-valor, indicando como nombre el valor del atributo *name* de la caja de texto.

Por último se envía con la función `submit_form()` [78]. Como resultado de la ejecución de esta función se obtiene el código HTML de la respuesta. Este código puede ser tratado nuevamente con `html_node()` para obtener el nodo correspondiente al nombre que se mostraría en la pantalla si la respuesta se obtuvo correctamente. Por último se extrae el texto (el nombre en este caso) con la función `html_text()` [74].

Se puede observar la resolución realizada en código para resolver este problema en el código fuente 45. Además se puede ver el resultado de ejecución donde se puede observar que la respuesta es satisfactoria en la ilustración 4.21.

Como consecuencia de este ejercicio, se puede concluir que se pueden realizar peticiones a través de formularios pues estas son respondidas y gestionadas correctamente por *rvest* (hay que tener en cuenta que es una prueba sencilla donde no se incluyen otras tecnologías del lado del cliente como *JavaScript*).

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/forms/"
4
5  #En primer lugar se crea la sesion
6  sesion <- html_session(url)
7
8  #Se extrae el formulario
9  formulario <- sesion %>% html_node("form") %>% html_form()
10
11 #Se establecen los valores
12 formRespondido <- formulario %>% set_values(nombre = "Daniel")
13
14 #Se envia
15 respuesta <- sesion %>% submit_form(formRespondido)
16
17 #Se intenta extraer el nombre en la respuesta
18 nombre <- respuesta %>% html_node("#nombre") %>% html_text()
19
20 if(exists("nombre")){
21         print(paste("Hola, te llamas ", nombre))
22 }else{
23         print("No se ha podido obtener respuesta")
24 }
```

Código fuente 45: Ejercicio de envío de formularios con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/forms/"
>
> #En primer lugar se crea la sesión
> sesion <- html_session(url)
>
> #Se extrae el formulario
> formulario <- sesion %>% html_node("form") %>% html_form()
>
> #Se establecen los valores
> formRespondido <- formulario %>% set_values(nombre = "Daniel")
>
> #Se envía
> respuesta <- sesion %>% submit_form(formRespondido)
Submitting with 'NULL'
>
> #Se intenta extraer el nombre en la respuesta
> nombre <- respuesta %>% html_node("#nombre") %>% html_text()
>
> if(exists("nombre")){
+ print(paste("Hola, te llamas ", nombre))
+ }else{
+ print("No se ha podido obtener respuesta")
+ }
[1] "Hola, te llamas  Daniel"
>
```

Ilustración 4.21: Resultado de ejecución del ejercicio de envío de formularios con *rvest*

4.7. Realización del test de gestión de cookies con rvest

El objetivo de este ejercicio es verificar cómo actúa *rvest* en aquellos casos en los que la información mostrada en la web visitada no es siempre la misma para todo el mundo, sino que varía en función del usuario y de las peticiones previas que haya hecho navegando por distintas páginas o autenticándose, entre otros. Lo que permite el almacenamiento de estos estados para un cliente dado se conoce como sesiones.

En este supuesto, se van a probar las funciones de navegación que provee *rvest*.

En primer lugar, simplemente se va a solicitar la página. Esta acción se va a realizar varias veces (en bucle) y se va a observar si la respuesta varía en cada petición, o si por el contrario no lo hace.

La página web creada para este supuesto, permite ver en la sesión actual cuántas veces se ha visitado. El número de veces que se ha visitado la página está aislado con un identificador, para poder *scrapearlo* fácilmente. Se va a verificar si de forma transparente *rvest* puede gestionar las cookies que almacenan los identificadores de sesión haciendo 5 peticiones normales (esto es, realizándolas como se han venido haciendo anteriormente), y viendo si por defecto *rvest* mantiene de forma transparente dichas *cookies de sesión* o no.

El algoritmo utilizado para realizar esta prueba se puede observar en el código fuente 46. La salida producida se puede observar en la ilustración 4.22. Además se ha generado una tabla (ilustración 4.23) que permite observar que todo el rato se muestra que previamente se habían realizado 0 visitas. Esto quiere decir, que realizando las peticiones como se venía haciendo, *rvest* no guarda la *cookie de sesión*, por lo que el servidor no la puede obtener en cada petición y asume que no había una sesión previa generada, por lo que opta por crear una nueva en cada petición.

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/cookies/"
> tabla <- data.frame(visitas=integer())
>
> #Se crea la tabla que almacenara las visitas en cada peticion
> for(i in 1:5){
+ pagina <- read_html(url)
+ visitas <- pagina %>% html_node("#visitas") %>% html_text %>% as.numeric
+ tabla[i,1] <- visitas
+ }
> #Se visualizara la tabla para ver si la sesion se mantiene
>
> View(tabla)
>
```

Ilustración 4.22: Resultado de ejecución de la verificación (fallida) de mantenimiento transparente de sesiones con *rvest*

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/cookies/"
4  tabla <- data.frame(visitas=integer())
5
6  #Se crea la tabla que almacenara las visitas en cada peticion
7  for(i in 1:5){
8          pagina <- read_html(url)
9          visitas <- pagina %>% html_node("#visitas") %>%
           ↪  html_text %>% as.numeric
10         tabla[i,1] <- visitas
11 }
12 #Se visualizara la tabla para ver si la sesion se mantiene
13
14 View(tabla)
```

Código fuente 46: Verificación (fallida) de mantenimiento transparente de sesiones con *rvest*

	visitas
1	0
2	0
3	0
4	0
5	0

Ilustración 4.23: Tabla generada en el resultado de ejecución de la verificación (fallida) de mantenimiento transparente de sesiones con *rvest*

Debido a esta razón, los desarrolladores de *rvest* están desarrollando funcionalidades (el autor aconseja darle *feedback* si se encuentra algún error o se quiere hacer alguna sugerencia [25][26]) permiten mantener una sesión de navegación. Para ello, se van a hacer algunas modificaciones sobre el código fuente 46:

- En primer lugar, la primera vez que se visite esta página, en vez de obtenerla, lo que se hará directamente es inicializar una nueva sesión que se utilizará en las subsecuentes peticiones. Esta primera sesión se hará con la función `html_session()` [79]. Dicha función además de crear la sesión, obtiene la página como si fuera una llamada corriente a `read_html()`.
- La segunda y subsiguientes llamadas se harán con la función `jump_to()` [80] que recibirá la sesión actual así como la nueva dirección URL de la página que se desea obtener (que en el caso que concierne, es la misma).

El algoritmo utilizado para realizar esta prueba se puede observar en el código fuente 47. La salida producida se puede observar en la ilustración 4.24. Además se ha generado una tabla (ilustración 4.25) que permite esta vez observar, cómo *rvest* mantiene la *cookie de sesión* y la gestiona correctamente, de forma que el servidor obtiene en cada petición a su dominio dicha *cookie* y gracias al identificador que almacena, permite identificar en el servidor la sesión que ya estaba en curso y continuar su uso.

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/cookies/"
>
> #Se crea la tabla que almacenara las visitas en cada peticion
> tabla <- data.frame(visitas=integer())
>
> #Previamente se crea la sesion
> sesion <- html_session(url)
>
> for(i in 1:5){
+ if(i==1){
+ pagina <- sesion
+ }else{
+ pagina <- jump_to(sesion, url)
+ }
+
+ visitas <- pagina %>% html_node("#visitas") %>% html_text %>% as.numeric
+ tabla[i,1] <- visitas
+ }
> #Se visualizara la tabla para ver si la sesion se mantiene
> View(tabla)
```

Ilustración 4.24: Resultado de ejecución del ejercicio de mantenimiento de sesiones con *rvest*

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/cookies/"
4
5  #Se crea la tabla que almacenara las visitas en cada peticion
6  tabla <- data.frame(visitas=integer())
7
8  #Previamente se crea la sesion
9  sesion <- html_session(url)
10
11 for(i in 1:5){
12         if(i==1){
13                 pagina <- sesion
14         }else{
15                 pagina <- jump_to(sesion, url)
16         }
17
18         visitas <- pagina %>% html_node("#visitas") %>%
           ↪  html_text %>% as.numeric
19         tabla[i,1] <- visitas
20 }
21 #Se visualizara la tabla para ver si la sesion se mantiene
22 View(tabla)
```

Código fuente 47: Ejercicio de mantenimiento de sesiones con *rvest*

	visitas
1	0
2	1
3	2
4	3
5	4

Ilustración 4.25: Tabla generada en el resultado de ejecución del ejercicio de mantenimiento de sesiones con *rvest*

4.8. Realización del test de renderizado de JavaScript con rvest

El objetivo de este ejercicio es determinar si *rvest* renderiza JavaScript tras descargar las páginas web, o si por el contrario no lo hace.

Para ello, se ha optado por utilizar el test de renderizado que se explicó en páginas anteriores. Por tanto, se va a proceder a descargar una página y a verificar si la caja cuyo identificador es *cargando* existe. En caso de existir, ello implica por como se ha realizado el test que la página no ha sido renderizada. Por el contrario, si no existiera, ello implicaría que *rvest* ha descargado la página y la ha renderizado (o al menos lo ha intentado).

El algoritmo que se ha ejecutado se puede observar en el código fuente 48. Como se puede observar en la salida mostrada en la ilustración 4.26, el *div* donde se indica que hay una carga en progreso persiste, lo que quiere decir que no se ha producido ninguna carga de Java Script. Por lo tanto, no se va a continuar con el ejemplo intentando extraer los datos cargados asíncronamente (en el caso particular del test, la extracción de profesionales), ya que estos no existen en la página.

Si se intentaran *scrapear* dichos profesionales, se observaría que no existen elementos. En las próximas secciones de este documento se podrá observar otra herramienta denominada *SeleniumPipes* que permite el renderizado de JavaScript. Adicionalmente se puede utilizar RSelenium o cualquiera otra que se puede observar en la primera sección para realizar una descarga asistida por navegador de tal forma que el renderizado es realizado por el propio motor del navegador. Como una rama de estudio próxima, se puede contemplar realizar un *wrapper* que permita opcionalmente transferir la página web descargada a una rutina que haga el renderizado manteniendo la simplicidad que tiene *rvest*.

```
1  library(rvest)
2
3  url <- "http://tfg.daniel-fl.com/test-js/"
4  cargando <- read_html(url) %>% html_node("#cargando")
5
6  if(exists("cargando")){
7          print("Div de carga encontrado. El navegador no admite
           ↪  carga de JavaScript")
8  }else{
9          print("Caja de carga no se muestra. Eso implica que el
           ↪  JS se ha ejecutado.")
10 }
```

Código fuente 48: Test de renderizado de JavaScript con *rvest*

```
> library(rvest)
>
> url <- "http://tfg.daniel-fl.com/test-js/"
> cargando <- read_html(url) %>% html_node("#cargando")
>
> if(exists("cargando")){
+ print("Div de carga encontrado. El navegador no admite carga de JavaScript")
+ }else{
+ print("Caja de carga no se muestra. Eso implica que el JS se ha ejecutado.")
+ }
[1] "Div de carga encontrado. El navegador no admite carga de JavaScript"
>
```

Ilustración 4.26: Resultado de ejecución del test de renderizado de JavaScript con *rvest*

4.9. Conclusiones de rvest

Una vez explicada toda la funcionalidad de *rvest* y una vez utilizada gran parte de la misma para realizar los tests diseñados, se pueden obtener varias conclusiones. En la tabla que aparece a continuación se puede observar una serie de datos de interés, tales como la dificultad de los supuestos planteados así como algunos datos de carácter general.

En la tabla que aparece a continuación se puede observar una serie de datos de interés, tales como la dificultad de los supuestos planteados así como algunos datos de carácter general.

Sobre este paquete se pueden hacer las siguientes afirmaciones, en base a la experiencia adquirida con él.

- Tiene una baja pendiente de aprendizaje y es de fácil instalación. En pocos minutos está en funcionamiento y no requiere de ningún parámetro de configuración especial.
- Es útil para proyectos de *web scraping* pequeños y medianos que no requieran para la obtención de los datos renderizados y resultantes de la ejecución de JavaScript.

PARÁMETROS GENERALES	
Dificultad de instalación	Fácil
Complejidad de la sintaxis	Fácil
Pendiente de aprendizaje	Reducida
TEST DE PROPÓSITO GENERAL	
Ejemplo de petición HTTP	Fácil
Obtener metaetiqueta del título de la página, descripción y palabras clave	Fácil
Obtener todos los enlaces de la barra de navegación	Fácil
Obtener enlaces por su clase y por su identificador	Medio
Obtener los atributos de una imagen	Fácil
Obtener diversas etiquetas de una clase y diferenciarlos	Fácil
Obtener para cada enlace del cuerpo, a qué párrafo pertenece	Medio
Ver los enlaces hermanos de los textos importantes	Medio
Obtener la tabla que hay en el cuerpo	Fácil
TEST DE ENVÍO DE FORMULARIOS	
Envío de información a través de formularios	Medio
TEST DE GESTIÓN DE COOKIES DE SESIÓN	
Mantenimiento de sesiones	Medio
TEST DE RENDERIZADO DE JAVASCRIPT	
Webscraping ante situaciones de carga asíncrona del DOM	Imposible - no renderiza JS

Capítulo 5

Web scraping con seleniumPipes

5.1. Introducción a seleniumPipes

seleniumPipes es una implementación ligera del protocolo W3C webdriver [51][53]. Con esta implementación se puede controlar un navegador web mediante programación en R. Este concepto se conoce como *navegación headless* y se suele utilizar para cuestiones como hacer *web scraping* o automatizar pruebas [89].

Es un *wrapper* que ha sido implementado mediante los siguientes paquetes [53]:

- **xml2:** Para realizar el *parsing* del documento HTML tras la descarga.
- **httr:** Para realizar la comunicación con el servidor *Selenium*.
- **magrittr:** Para poder realizar el encadenamiento de funciones mediante tuberías.

En la ilustración 5.1 se puede observar la arquitectura que hay que gestionar. Este paquete simplemente es el enlace entre el servidor *SeleniumPipes* y el servidor *Selenium*.

En la próxima sección se expondrán las funcionalidades principales que este paquete provee.

5.2. Visión general de las funcionalidades de seleniumPipes

Aunque ya se expusieron en la sinopsis del paquete, se vuelven a exponer brevemente las diversas funcionalidades que posee el paquete [54]:

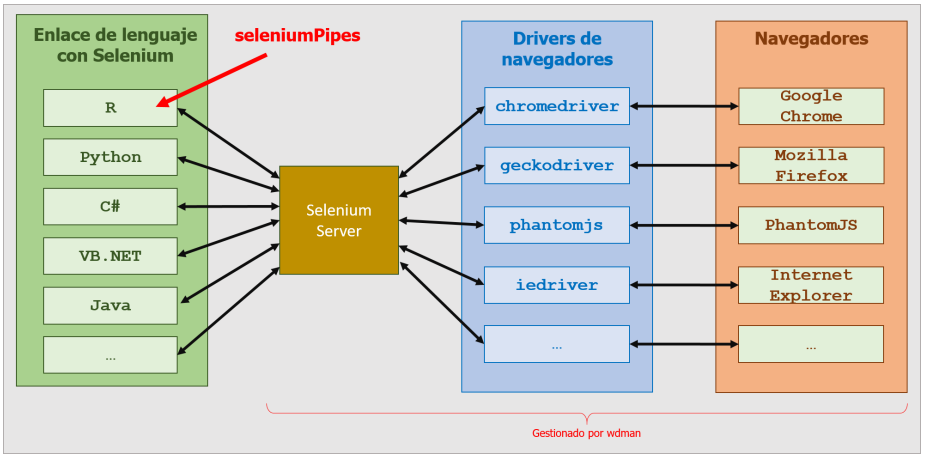


Ilustración 5.1: Esquema de la arquitectura en la comunicación de *seleniumPipes* con *Selenium*

- Permite conectarse a cierto conjunto de navegadores entre los que destaca **Firefox**, **Chrome**, **Edge** o **PhantomJS** entre otros.
- Utiliza notación de tuberías para facilitar la sintaxis encadenando funciones. Para ello hace uso subyacentemente del paquete **magrittr**.
- Permite realizar tareas de navegación con funciones tales como `go()` [90], `back()` [91], `forward()` [92] o `refresh()` [93].
- Acceder a los elementos de la página utilizando selectores de id, de clase, de nombre de etiqueta, selectores *XPath* o CSS. Esto es posible a las funciones `findElement()` [94], `findElementFromElement()` [95], `findElements()` [96] y `findElementsFromElement()` [97].
- Permite interactuar con los elementos mediante eventos de teclado o de ratón.
- Permite ejecutar *JavaScript* sobre la página.
- Permite realizar cierta gestión de errores en caso de que haya algún problema en la invocación de alguna función.
- Permite interactuar con marcos y con ventanas.

5.3. Instalación de seleniumPipes

Existen varias maneras de instalar y poner en marcha una instancia del servidor *Selenium*, que es el servidor que hace falta según la arquitectura para poder utilizar **seleniumPipes** [54]:

- Descargar Selenium Server Standalone y ejecutarlo mediante línea de comandos.
- Utilizar un script de R para instalar dicho servidor y ponerlo en marcha (dicho script está en el paquete *RSelenium* y al parecer está *deprecated*).
- Descargar un *docker* y configurarlo.
- Una última que se ha encontrado indagando en el proyecto hermano **RSelenium** consiste en instalar *wdman*, ya que este paquete agiliza la gestión de la apertura del servidor y el enlace con el navegador mediante el driver necesario [49].

  Se procederá a utilizar **seleniumPipes** utilizando este último enfoque ya que para los restantes no se ha logrado llegar a buen puerto (lo que no quiere decir que sean inválidos, simplemente no han funcionado como era de esperar y se descartaron inmediatamente para no perder demasiado tiempo). Para realizar la instalación simplemente basta con hacer la instalación desde los repositorios de CRAN 49.

```
1 install.packages("wdman")
2
3 install.packages("seleniumPipes")
```

Código fuente 49: Instalación de *seleniumPipes*

5.4. Detalle pormenorizado de las funciones de seleniumPipes

5.4.1. El operador tubería (%>%)

Debido a que el encadenamiento de funciones mediante el operador de tubería es utilizado en varios capítulos, ha sido ubicado en el anexo A. Es aconsejable leerlo antes de empezar a manejar el paquete *seleniumPipes* ya que en él aprenderá la sintaxis para encadenar funciones, muy útil porque aligera enormemente la sintaxis.

5.4.2. Cuestiones previas: crear un objeto de *driver* remoto

Para poder controlar un navegador, se debe crear un servidor remoto que lo controle, y luego se debe llamar a dicho servidor. Para esta última labor se utiliza la función `remoteDr()`, tal y como se indica en la lista de argumentos y en el código fuente 50 [98].

```
remoteDr(remoteServerAddr = "http://localhost", port = 4444L,
↪  browserName = "firefox", version = "", platform = "ANY",
↪  javascript = TRUE, nativeEvents = TRUE, extraCapabilities =
↪  list(), path = "wd/hub", newSession = TRUE)
```

Código fuente 50: Función de creación de objeto de *driver* remoto con *seleniumPipes*

Argumentos

`remoteServerAddr`	Cadena indicando la dirección URL donde se encuentra el servidor remoto. Por defecto es *localhost*.
`port`	Entero que indica el puerto del servidor remoto al que hay que conectarse.
`browserName`	Cadena de caracteres indicando el navegador que va a ser utilizado. Las cadenas posibles son fboxchrome, fboxfirefox, fboxinternet explorer, fboxiphone, fboxhtmlunit. Por defecto es firefox. Según ciertas pruebas realizadas a título personal, en principio podría utilizarse PhantomJS utilizando la cadena phantomjs.
`version`	Cadena de caracteres indicando la versión o bien cadena vacía si esta se desconoce (por defecto se suministra este último).
`platform`	Cadena de caracteres indicando la plataforma (sistema operativo) sobre la que se ejecuta el navegador web. Las cadenas posibles son WINDOWS, XP, VISTA, MAC, LINUX,UNIX. También se podría especificar ANY si se desconoce (este parámetro es opcional, por lo que por defecto el valor indicado es ANY.
`javascript`	Valor lógico donde se indica si se admite la ejecución de JavaScript por el usuario en la página actual.
`nativeEvents`	Valor booleano donde se indica si la sesión soporta eventos nativos. Un usuario avanzado realizará interacciones simulando los eventos por JavaScript (eventos sintéticos) o bien permitiendo que el navegador genere dichos eventos JavaScript (eventos nativos). Los eventos nativos simulan mejor las interacciones del usuario [98].
`extraCapabilites`	Lista conteniendo parámetros específicos relativos al sistema operativo, a la plataforma o al driver utilizado.
`path`	Ruta en el lado del servidor para realizar las llamadas al *driver* concreto. Normalmente se suele mantener el valor por defecto.
`newSession`	Valor booleano que indica si se inicia nueva del navegador. Si es verdadero, automáticamente se abre el navegador ejecutando internamente `newSession()`.

Retorno

Se devuelve como resultado de la ejecución un objeto de *driver* remoto *rDriver* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.3. Cuestiones previas: parámetro retry

Este parámetro no se utilizará en los ejemplos que se van a desarrollar, pero es posible utilizarlo en la gran mayoría de las funciones que se explicarán a continuación.

La habilidad de *retry* o de reintento está habilitada por defecto [99][100]. Esta habilidad permite reintentar la invocación de la función en curso si la ejecución de la misma ha fallado. Si el valor es cierto, se realizan las siguientes asignaciones:

- `noTry = getOption("seleniumPipes_no_try")`
- `delay = getOption("seleniumPipes_no_try_delay")`

Si el valor recibido es falso, se deshabilita la posibilidad de hacer múltiples intentos de la invocación si esta resulta fallida.

También se puede pasar como parámetro *retry* una lista asociando los valores deseados a los siguientes argumentos, lo que hará que se haga uso de estos en vez de los valores por defecto [99][100]:

- noTry: entero indicando cuántas veces se debe intentar la invocación a la función.
- delay: entero indicando el retraso entre dichas invocaciones sucesivas (en segundos).

5.4.4. Funciones de navegación: acceder a un sitio web

Para poder acceder a un sitio web en el navegador, se debe utilizar la función `go()`, tal y como se indica en la lista de argumentos y en el código fuente 51 [90].

```
1 go(remDr, url, ...)
```

Código fuente 51: Función de acceso a una web con *seleniumPipes*

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`url`	Dirección URL a la que se desea acceder.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Argumentos

Retorno

Se devuelve como resultado de la ejecución el mismo objeto de *driver* remoto *rDriver* que se ha pasado por parámetro, de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.5. Funciones de navegación: obtener la dirección URL actual

Para poder obtener la URL que se encuentra actualmente cargada en el navegador, se debe utilizar la función `getCurrentURL()`, tal y como se indica en la lista de argumentos y en el código fuente 52 [101].

```
1 getCurrentUrl(remDr, ...)
```

Código fuente 52: Función para obtener URL actual con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se devuelve como resultado de la ejecución el mismo objeto de *driver* remoto *rDriver* que se ha pasado por parámetro, de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.6. Funciones de navegación: ir hacia atrás

Para poder ir hacia atrás, esto es, a la página que se ha visitado anteriormente según el historial del navegador, se debe utilizar la función `back()`, tal y como se indica en la lista de argumentos y en el código fuente 53 [91]. Hay que destacar que esta función se ejecutará solamente si es posible, esto es, que anteriormente se haya visitado una página en el navegador actual.

```
1 back(remDr, ...)
```

Código fuente 53: Función para ir hacia atrás con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se devuelve como resultado de la ejecución el mismo objeto de *driver* remoto *rDriver* que se ha pasado por parámetro, de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.7. Funciones de navegación: ir hacia adelante

Para poder ir hacia adelante en el historial del navegador, se debe utilizar la función `forward()`, tal y como se indica en la lista de argumentos y en el código fuente 54 [92]. Hay que destacar que esta función se ejecutará solamente si es posible, esto es, que se pueda ir hacia adelante en el historial de navegación (porque se haya ido hacia atrás anteriormente).

```
1 forward(remDr, ...)
```

Código fuente 54: Función para ir hacia adelante con *seleniumPipes*

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Argumentos

Retorno

Se devuelve como resultado de la ejecución el mismo objeto de *driver* remoto *rDriver* que se ha pasado por parámetro, de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.8. Funciones de navegación: recargar la página

Para poder recargar la página, se debe utilizar la función `refresh()`, tal y como se indica en la lista de argumentos y en el código fuente 55 [93].

```
1 refresh(remDr, ...)
```

Código fuente 55: Función para recargar la página con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se devuelve como resultado de la ejecución el mismo objeto de *driver* remoto *rDriver* que se ha pasado por parámetro, de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.9. Cerrar la navegación

Para poder finalizar la sesión de navegación y cerrar el navegador se debe utilizar la función `deleteSession()`, tal y como se indica en la lista de argumentos y en el código fuente 56 [102].

```
deleteSession(remDr, ...)
```

Código fuente 56: Función para cerrar la navegación con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *rDriver*. Sin embargo, el identificador de sesión ha sido borrado y el navegador asociado debe ser cerrado por el servidor.

5.4.10. Selección de elementos: obtener el título de la página

Para poder obtener el título de la página indicado en la metaetiqueta title, se debe utilizar la función `getTitle()`, tal y como se indica en la lista de argumentos y en el código fuente 57 [103].

```
getTitle(remDr, ...)
```

Código fuente 57: Función para obtener el título de la página con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Como resultado de la ejecución se obtiene la cadena de caracteres correspondiente al título indicado en la metaetiqueta de título de la página.

5.4.11. Selección de elementos: obtener el código fuente

Para obtener el código fuente de la última página cargada se debe utilizar la función `getPageSource()`, tal y como se indica en la lista de argumentos y en el código fuente 58 [104].

```
1 getTitle(remDr, ...)
```

Código fuente 58: Función para obtener el código fuente con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se obtiene el código fuente mediante un objeto *xml_document*. Este código fuente es el resultado del *parsing* mediante *xml2* (función mintinlineRread_html).

5.4.12. Selección de elementos: obtener elementos determinados del DOM

Para obtener elementos determinados del DOM, se debe utilizar alguna de las siguientes funciones:

- `findElement()` [94]: Debe utilizarse si se espera obtener un solo elemento.
- `findElements()` [96]: Debe utilizarse si se espera obtener más de un elemento.

Dichas funciones se explican en el código fuente 59 así como los argumentos que aparecen a continuación.

Argumentos

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *wbElement* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

```
findElement(remDr, using = c("xpath", "css selector", "id",
↪  "name", "tag name", "class name", "link text", "partial link
↪  text"), value, ...)
findElements(remDr, using = c("xpath", "css selector", "id",
↪  "name", "tag name", "class name", "link text", "partial link
↪  text"), value, ...)
```

Código fuente 59: Funciones para obtener elementos de otros elementos con *seleniumPipes*

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`using`	Esquemas de localización para determinar en qué atributo ha de tenerse atención para realizar la búsqueda, siendo los esquemas admitidos los siguientes: çlass name", çss selector", ïd", "name", "link text", "partial link text", "tag name", "xpath". Por defecto se escoge 'xpath'. Se acepta la coincidencia parcial de cadenas.
`value`	Objetivo de búsqueda (según el esquema, se puede introducir un nombre de clase, un selector CSS, un identificador, un nombre, etcétera).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

5.4.13. Selección de elementos: obtener elementos a partir de otros elementos

Para obtener elementos determinados a partir de otros elementos seleccionados previamente, se debe utilizar alguna de las siguientes funciones:

- `findElementFromElement()` [95]: Debe utilizarse si se espera obtener un solo elemento.
- `findElementsFromElement()` [97]: Debe utilizarse si se espera obtener más de un elemento.

Dichas funciones se explican en el código fuente 60 así como los argumentos que aparecen a continuación.

Argumentos

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *wbElement* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

```
1 findElementFromElement(webElem, using = c("xpath", "css
  ↪ selector", "id", "name", "tag name", "class name", "link
  ↪ text", "partial link text"), value, ...)
2 findElementsFromElement(webElem, using = c("xpath", "css
  ↪ selector", "id", "name", "tag name", "class name", "link
  ↪ text", "partial link text"), value, ...)
```

Código fuente 60: Funciones para obtener elementos de otros elementos con *seleniumPipes*

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`using`	Esquemas de localización para determinar en qué atributo ha de tenerse atención para realizar la búsqueda, siendo los esquemas admitidos los siguientes: çlass name", çss selector", ïd", "name", "link text", "partial link text", "tag name", "xpath". Por defecto se escoge 'xpath'. Se acepta la coincidencia parcial de cadenas.
`value`	Objetivo de búsqueda (según el esquema, se puede introducir un nombre de clase, un selector CSS, un identificador, un nombre, etcétera).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

5.4.14. Selección de elementos: obtener el elemento activo

Para poder obtener el elemento activo (aquel que actualmente tiene el foco), se debe utilizar la función `getActiveElement()`, tal y como se indica en la lista de argumentos y en el código fuente 61 [105].

```
1 getActiveElement(remDr, ...)
```

Código fuente 61: Función para obtener el elemento activo con *seleniumPipes*

Argumentos

`remDr`	Objeto de la clase *rDriver* (objeto de *driver* remoto).
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *wbElement* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.15. Obtención de características de elementos: obtener el texto visible del elemento

Para obtener el texto visible de un elemento dado, se debe utilizar la función `getElementText()`. La cabecera de dicha función se encuentra en el código fuente 62 [106]. Los argumentos de dicha cabecera aparecen a continuación.

```
1 getElementText(webElem, ...)
```

Código fuente 62: Función para obtener el texto visible de un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se obtiene como resultado de la función el texto visible como cadena de caracteres.

5.4.16. Obtención de características de elementos: obtener el valor de un atributo del elemento

Para obtener el valor de un atributo concreto de un elemento dado, se debe utilizar la función `getElementAttribute()` [107]. La cabecera de dicha función se encuentra en el código fuente 63. Los argumentos de dicha cabecera aparecen a continuación.

```
getElementAttribute(webElem, attribute, ...)
```

Código fuente 63: Función para obtener el valor de un atributo de un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`attribute`	Cadena de caracteres que identifica el nombre del atributo del que se quiere obtener su valor
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se obtiene como resultado de la función el valor del atributo o valor nulo si dicho elemento no tiene el atributo requerido.

5.4.17. Obtención de características de elementos: obtener el nombre del elemento

Para obtener el nombre de un elemento dado, se debe utilizar la función `getElementTagName()` [108]. La cabecera de dicha función se encuentra en el código fuente 64. Los argumentos de dicha cabecera aparecen a continuación.

```
getElementTagName(webElem, ...)
```

Código fuente 64: Función para obtener el nombre de un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se obtiene como resultado de la función el nombre del elemento en minúsculas.

5.4.18. Obtención de características de elementos: obtener el valor de una propiedad CSS

Para obtener el valor de una propiedad CSS de un elemento dado, se debe utilizar la función `getElementCssValue()` [109]. La cabecera de dicha función se encuentra en el código fuente 65. Los argumentos de dicha cabecera aparecen a continuación.

```
1 getElementCssValue(webElem, propertyName, ...)
```

Código fuente 65: Función para obtener el valor de una propiedad CSS de un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`propertyName`	Cadena de caracteres que hace referencia al nombre de la propiedad a consultar.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se obtiene como resultado de la función el valor de la propiedad CSS indicada.

5.4.19. Obtención de características de elementos: obtener el valor de una propiedad JavaScript

Para obtener el valor de una propiedad JavaScript de un elemento dado, se debe utilizar la función `getElementProperty()` [109]. La cabecera de dicha función se encuentra en el código fuente 66. Los argumentos de dicha cabecera aparecen a continuación.

```
getElementProperty(webElem, property, ...)
```

Código fuente 66: Función para obtener el valor de una propiedad JavaScript de un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`property`	Cadena de caracteres que hace referencia a la propiedad a consultar.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se obtiene como resultado de la función el valor de la propiedad JavaScript indicada.

5.4.20. Obtención de características de elementos: obtener las dimensiones y coordenadas de un elemento

Para obtener las dimensiones y coordenadas del elemento dado en el momento actual, se debe utilizar la función `getElementRect()` [110]. La cabecera de dicha función se encuentra en el código fuente 67. Los argumentos de dicha cabecera aparecen a continuación.

```
getElementRect(webElem, ...)
```

Código fuente 67: Función para obtener las dimensiones y coordenadas de un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

El valor retornado por la función estriba en una lista asociativa que incluye los siguientes elementos [111]:

- `x`: Posición en el eje X de la esquina superior izquierda del elemento web relativa al contexto de navegación actual en píxeles de CSS.
- `y`: Posición en el eje Y de la esquina superior izquierda del elemento web relativa al contexto de navegación actual en píxeles de CSS.
- `height`: Altura del elemento web en píxeles de CSS.
- `width`: Anchura del elemento web en píxeles de CSS.

5.4.21. Obtención de características de elementos: verificar si un elemento está habilitado

Esta función puede ser de utilidad para ser verificada en campos de formulario que así lo permiten como botones, entradas de valores, desplegables, cajas de texto, etcétera [112]. Para verificar si un elemento está o no habilitado, se debe utilizar la función `isElementEnabled()` [113]. La cabecera de dicha función se encuentra en el código fuente 68. Los argumentos de dicha cabecera aparecen a continuación.

```
1 isElementEnabled(webElem, ...)
```

Código fuente 68: Función para verificar si un elemento está habilitado con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Como resultado de ejecución de esta función se obtendrá un valor booleano donde se indique si el elemento está habilitado indicando valor cierto, o si no lo está indicando valor falso.

5.4.22. Obtención de características de elementos: verificar si el elemento está seleccionado

Para verificar si un elemento de opción option o un elemento de entrada input de selección múltiple (checkbox) o excluyente (radio) ha sido seleccionado se debe utilizar la función `isElementSelected()` [114]. La cabecera de dicha función se encuentra en el código fuente 69. Los argumentos de dicha cabecera aparecen a continuación.

```
1 isElementSelected(webElem, ...)
```

Código fuente 69: Función para verificar si un elemento está seleccionado con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Como resultado de ejecución de esta función se obtendrá un valor booleano donde se indique si el elemento está seleccionado indicando valor cierto, o si no lo está indicando valor falso.

5.4.23. Interacción con elementos: enviar pulsaciones de teclado

En caso de que se desee escribir en algún elemento de formulario, `elementSendKeys()` [115] es la función indicada. Permite dado un elemento, enviar pulsaciones de teclado, de tal forma que se podría rellenar una caja de texto o hacer algún otro tipo de interacción.

En caso de que el elemento indicado no permita interacción de teclado, se obtendrá un error de elemento no interactuable por teclado.

La cabecera de dicha función se encuentra en el código fuente 70. Los argumentos de dicha cabecera aparecen a continuación.

```
1 elementSendKeys(webElem, ...)
```

Código fuente 70: Función para enviar pulsaciones de teclado a un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Pulsaciones de teclado que se desean enviar. Se puede enviar una o más de una cadenas combinables (opcionalmente) con símbolos Unicode accesibles a través de la lista *selKeys* mediante su identificador. En caso de requerir símbolos, se aconseja imprimir dicha lista y escoger aquel que pueda interesar previamente.

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *wbElement* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.24. Interacción con elementos: vaciar controles de formulario

Para vaciar una caja de texto o una entrada de texto normal de un formulario se debe utilizar la función `elementClear()` [116]. La cabecera de dicha función se encuentra en el código fuente 71. Los argumentos de dicha cabecera aparecen a continuación.

```
1 elementClear(webElem, ...)
```

Código fuente 71: Función para vaciar un control de formulario con *seleniumPipes*

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Argumentos

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *wbElement* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.25. Interacción con elementos: hacer clic en un elemento

Para hacer clic en un elemento de una web (por ejemplo hacer clic en el botón de envío de un formulario) se debe utilizar la función `elementClick()` [117]. La manera de proceder de esta función es hacer *scrolling* hasta que se puede observar el elemento deseado; tras ello hace clic exactamente en el punto central de dicho elemento.

Si el elemento no admite interacciones con el puntero (por ejemplo porque no sea visible), un error de elemento no interactuable sería indicado.

La cabecera de dicha función se encuentra en el código fuente 72. Los argumentos de dicha cabecera aparecen a continuación.

```
1 elementClick(webElem, ...)
```

Código fuente 72: Función para hacer clic en un elemento con *seleniumPipes*

Argumentos

`webElem`	Objeto de la clase *wbElement* obtenido de una búsqueda previa.
`...`	Argumentos adicionales. Actualmente se puede pasar el argumento *retry*.

Retorno

Se devuelve como resultado de la ejecución un objeto de clase *wbElement* de forma invisible (no se imprime en pantalla). La única utilidad de dicho retorno, es permitir la posibilidad de encadenar varias invocaciones que requieran

dicho objeto como primer parámetro. Dicho encadenamiento es posible gracias a *magrittr* [85].

5.4.26. Miscelánea: otras funciones

Este paquete tiene un montón de funcionalidades, sobre todo para poder influir en detalles pequeños de la ejecución de la web. A continuación se va a exponer una lista con el resto de funciones [118][119]. Son funciones muy interesantes si se desea hacer *web scraping* para realizar pruebas unitarias u otro tipo de cuestiones más avanzadas. También han sido eliminadas aquellas funciones que sean antiguas y de las que en principio no se haría uso al existir sus equivalentes más modernos.

- `acceptAlert()`: Aceptar mensaje de alerta.
- `addCookie()`: Añadir una *cookie* específica.
- `checkResponse()`: Verifica la respuesta del servidor remoto.
- `closeWindow()`: Cierra la ventana actual.
- `deleteAllCookies()`: Borra todas las *cookies*.
- `deleteCookie()`: Borra una *cookie* dada.
- `dismissAlert()`: Descartar una alerta.
- `errorContent()`: Devuelve el contenido de error del servidor remoto *webdriver*.
- `errorResponse()`: Devuelve la respuesta de error del servidor remoto *webdriver*.
- `executeAsyncScript()`: Ejecuta JavaScript de forma asíncrona en el navegador.
- `executeScript()`: Ejecuta JavaScript en el navegador.
- `fullscreenWindow()`: Redimensiona la ventana actual a pantalla completa.
- `getAlertText()`: Obtiene el texto de la alerta.
- `getAllCookies()`: Obtiene todas las *cookies* del dominio.
- `getNamedCookie()`: Obtiene una *cookie* por su nombre.
- `getWindowHandle()`: Obtiene el *handle* de la ventana actual.
- `getWindowHandles()`: Obtiene todos los *handles* de ventanas.
- `getWindowPosition()`: Obtiene la posición de la ventana actual.

- `getWindowSize()`: Obtiene el tamaño de la ventana actual.
- `maximizeWindow()`: Maximiza la ventana actual.
- `newSession()`: Crea una nueva sesión.
- `queryDriver()`: Envía una consulta al *driver* remoto.
- `sendAlertText()`: Enviar texto a la alerta.
- `setTimeout()`: Configurar la cantidad de tiempo que un tipo de operación debe tardar como máximo.
- `setWindowPosition()`: Cambiar la posición de la ventana actual.
- `setWindowSize()`: Cambiar el tamaño de la ventana actual.
- `switchToFrame()`: Cambiar el foco a otro marco de la página.
- `switchToParentFrame()`: Cambiar el foco al marco principal (marco padre).
- `switchToWindow()`: Cambiar el foco a otra ventana.
- `takeElementScreenshot()`: Hacer una captura de pantalla de un elemento dado.
- `takeScreenshot()`: Hacer una captura de toda la página.

5.5. Realización del test de propósito general con seleniumPipes

5.5.1. Ejemplo de petición HTTP

En primer lugar, se procede a abrir un navegador. Para ello, se utiliza el paquete *wdman* y su función `chrome()` para abrir dicho navegador en Chrome. Hay que destacar que no es una apertura cualquiera, si no que ejecuta el servidor de Selenium Webdriver y es éste el que abre el navegador Chrome utilizando para ello un driver específico para Chrome. De esta manera, se accede al navegador a través de la API unificada de Selenium Webdriver. El puerto utilizado es indiferente siempre y cuando sea un puerto distinto del rango de puertos bien conocidos (los puertos conocidos son aquellos menores que 1024 ya que están estandarizados para aplicaciones concretas) y que esté libre.

Una vez ya abierto el servidor, se procede a la conexión con él mismo para comandar el navegador. Esto se hace utilizando la función `remoteDr()` [98], accediendo al mismo puerto. El host viene dado de forma implícita (*localhost*) pero si fuera otro habría que especificarlo.

Una vez hecho esto que va a ser común para todos los ejercicios, se puede observar que se ha abierto un navegador Chrome, donde se puede observar una

indicación que destaca que un software automatizado de pruebas está controlando Chrome (ilustración 5.2). Tras abrirse, queda liberado el *prompt* en *RGui*, y es cuando se procede a acceder a la página deseada con `go()` [90] y a obtener el código fuente de la misma con `getPageSource()` [104].

Para finalizar se cierra la sesión con el servidor remoto utilizando *deleteSession()* [102]. Esto cerrará el servidor, y éste antes de cerrarse dará orden al navegador para proceder con su cierre.

El código fuente que se obtiene es un objeto de documento XML xml_document al igual que con *rvest*, pudiendo ser tratado si así se requiriese con *xml2* o algún derivado.

El código fuente completo se puede observar en el código fuente 73. El resultado de dicha ejecución donde se puede contrastar lo explicado anteriormente se puede observar en la ilustración 5.3.

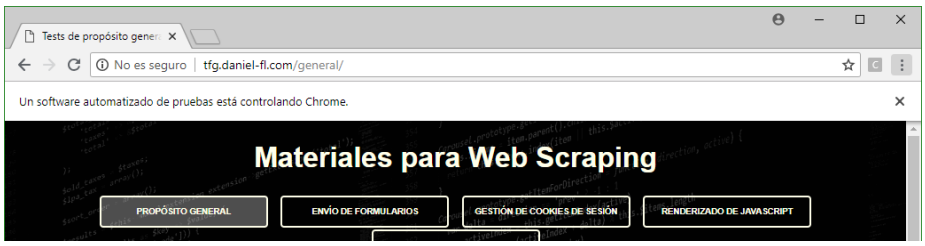


Ilustración 5.2: Navegador comandado remotamente con *seleniumPipes*

```
1  library(seleniumPipes)
2  library(wdman)
3
4  #Se arranca
5  selServ <- chrome(port = 5005L)
6  remDr <- remoteDr(port = 5005L)
7
8
9  pagina <- remDr %>%
10           go(url = "http://tfg.daniel-fl.com/general/") %>%
11           getPageSource
12
13 remDr %>% deleteSession
14
15 print(pagina)
```

Código fuente 73: Ejercicio de petición HTTP con *seleniumPipes*

```
> library(seleniumPipes)
> library(wdman)
>
> #Se arranca
> selServ <- chrome(port = 5005L)
checking chromedriver versions:
BEGIN: PREDOWNLOAD
BEGIN: DOWNLOAD
BEGIN: POSTDOWNLOAD
> remDr <- remoteDr(port = 5005L)
>
>
> pagina <- remDr %>%
+   go(url = "http://tfg.daniel-fl.com/general/") %>%
+   getPageSource
>
> remDr %>% deleteSession
>
> print(pagina)
{xml_document}
<html xmlns="http://www.w3.org/1999/xhtml" lang="es">
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 .$
[2] <body>\n        <div id="wrapper">\n            <header><h1><a href="../" .$
> |
```

Ilustración 5.3: Resultado de ejecución del ejercicio de petición HTTP con *seleniumPipes*

5.5.2. Obtener metaetiqueta del título de la página, descripción y palabras clave

Para realizar este ejercicio, se va a partir del primer ejercicio de *seleniumPipes*. El objetivo es obtener el elemento *title* de la página así como las metaetiquetas de descripción y de palabras clave.

Para el caso del título *seleniumPipes* provee la función `getTitle()` [103], que permite su extracción directamente.

Para el caso de la descripción y las metaetiquetas, se deben empezar a utilizar dos funciones: `findElement()` [94], que permite mediante el criterio de búsqueda indicado (en este caso un selector CSS) obtener el primer elemento que cumpla con la condición indicada. Con este selector se acceden a las metaetiquetas cuyo nombre sea la descripción o las palabras clave. Además se utiliza la función `getElementAttribute()` [107] para obtener el contenido de las mismas, ya que su información no se guarda en el cuerpo porque las metaetiquetas no lo tienen, sino que se debe extraer de un atributo denominado content.

El código fuente completo se puede observar en el código fuente 74. Un extracto del resultado de dicha ejecución donde se puede contrastar lo explicado anteriormente se puede observar en la ilustración 5.4.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5  remDr %>% go("http://tfg.daniel-fl.com/general/")
6
7  #Se imprime el titulo
8  titulo <- remDr %>% getTitle
9  print(titulo)
10
11 #Se imprime la descripcion:
12 descr <- remDr %>%
13           findElement('css selector', "meta[name=description]")
              ↪  %>%
14           getElementAttribute("content")
15 print(descr)
16
17 #Se imprimen las palabras clave:
18 kw <- remDr %>%
19       findElement('css selector', "meta[name=keywords]") %>%
20       getElementAttribute("content")
21
22 print(strsplit(as.character(kw), ', '))
23
24 remDr %>% deleteSession
```

Código fuente 74: Ejercicio de obtención de metaetiquetas con *seleniumPipes*

```
> #Se imprime el titulo
> titulo <- remDr %>% getTitle
> print(titulo)
[1] "Tests de propósito general | Daniel Francisco López"
>
> #Se imprime la descripcion:
> descr <- remDr %>%
+          findElement('css selector', "meta[name=description]") %>%
+          getElementAttribute("content")
> print(descr)
[1] "Tests para hacer diversas pruebas de selectores, para ver las posibilidade$
>
> #Se imprimen las palabras clave:
> kw <- remDr %>%
+       findElement('css selector', "meta[name=keywords]") %>%
+       getElementAttribute("content")
>
> print(strsplit(as.character(kw), ', '))
[[1]]
[1] "web"       "scrapping" "utilidad"  "CSS"       "XPath"     "DOM"

>
> remDr %>% deleteSession
```

Ilustración 5.4: Resultado de ejecución del ejercicio de obtención de metaetiquetas con *seleniumPipes*

5.5.3. Obtener todos los enlaces de la barra de navegación

Para continuar se obtienen todos los enlaces que hay en la barra de navegación. Para ello, se va a hacer uso de CSS como selector, utilizando la función `findElements()` que permite obtener todas las apariciones localizadas con el criterio de búsqueda establecido [96]. Luego es necesario hacer dos listas, para unirlas posteriormente en una tabla.

La primera lista contendrá el texto de los enlaces, para lo que se utilizará la función `getElementText()` [106] aplicado a cada uno de los elementos de la lista de elementos encontrados con `sapply()`.

Por otra parte se hace la misma operación pero para obtener los enlaces con la función `getElementAttribute()` [107].

Por último se crea un *data frame* y se visualiza en pantalla con la función `View()`.

El código fuente completo se puede observar en el código fuente 75. Un extracto del resultado de dicha ejecución donde se puede contrastar lo explicado anteriormente se puede observar en la ilustración 5.5 y la tabla generada en la ilustración 5.6.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5  remDr %>% go("http://tfg.daniel-fl.com/general/")
6
7  #Se obtienen los enlaces
8  links <- remDr %>% findElements('css selector', "nav a")
9
10 textos <- sapply(links, function(x){x %>% getElementText()})
11 enlaces <- sapply(links, function(x){x %>%
   ↪  getElementAttribute("href")})
12
13 tabla <- data.frame(textos, enlaces)
14 View(tabla)
15
16 remDr %>% deleteSession
```

Código fuente 75: Ejercicio de obtención de enlaces de navegación con *seleniumPipes*

```
> remDr <- remoteDr(port = 5005L)
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces
> links <- remDr %>% findElements('css selector', "nav a")
>
> textos <- sapply(links, function(x){x %>% getElementText()})
> enlaces <- sapply(links, function(x){x %>% getElementAttribute("href")})
>
> tabla <- data.frame(textos, enlaces)
> View(tabla)
>
> remDr %>% deleteSession
>
```

Ilustración 5.5: Resultado de ejecución del ejercicio de obtención de enlaces de navegación con *seleniumPipes*

	textos	enlaces
1	PROPÓSITO GENERAL	http://tfg.daniel-fl.com/general/#
2	ENVÍO DE FORMULARIOS	http://tfg.daniel-fl.com/forms
3	GESTIÓN DE COOKIES DE SESIÓN	http://tfg.daniel-fl.com/cookies
4	RENDERIZADO DE JAVASCRIPT	http://tfg.daniel-fl.com/test-js
5	PERFORMANCE	http://tfg.daniel-fl.com/performance

Ilustración 5.6: Tabla de la ejecución del ejercicio de obtención de enlaces de navegación con *seleniumPipes*

5.5.4. Obtener enlaces por su clase y por su identificador

El objetivo de este ejercicio es hacer dos tipos de obtención de enlaces para observar diversas modalidades de los criterios de búsqueda que admite *seleniumPipes*. En la parte central (cuerpo) del test de propósito general se han añadido enlaces identificados por *id* y por clase.

Ambos apartados empiezan haciendo uso de la función `findElement()` [94]. En cada caso concreto se indica un criterio de búsqueda concreto (por *id* o por clase). Además en cada código se repite la misma acción pero creando el selector equivalente en CSS.

Luego el proceso es común al ejercicio anterior. Se extrae el nombre de los enlaces con `getElementText()` [106] y los enlaces con `getElementAttribute()` [107].

En primer lugar se va a realizar la obtención de enlaces mediante su identificador, lo que se puede observar en el código fuente 76 y en la ilustración 5.7.

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces
> linkla <- remDr %>% findElement('id', "link_seccion_formularios")
> print(paste("Seccion de formularios:", linkla %>% getElementText(), " - ", li$
[1] "Seccion de formularios: ENVÍO DE FORMULARIOS  -  http://tfg.daniel-fl.com/$
>
> linklb <- remDr %>% findElement('css selector', "#link_seccion_js")
> print(paste("Seccion de JS:", linklb %>% getElementText(), " - ", linklb %>% $
[1] "Seccion de JS: RENDERIZADO DE JAVASCRIPT  -  http://tfg.daniel-fl.com/test$
>
> remDr %>% deleteSession
>
```

Ilustración 5.7: Resultado de ejecución del ejercicio de obtención de enlaces por su identificador con *seleniumPipes*

En segundo lugar se va a realizar la obtención de enlaces mediante su clase, lo que se puede observar en el código fuente 77 y en la ilustración 5.8. Además en este caso, se ha optado por ubicar los datos en forma de tabla, pudiéndose ver tales tablas en las ilustraciones 5.9 y 5.10.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5  remDr %>% go("http://tfg.daniel-fl.com/general/")
6
7  #Se obtienen los enlaces
8  link1a <- remDr %>% findElement('id',
   ↪  "link_seccion_formularios")
9  print(paste("Seccion de formularios:", link1a %>%
   ↪  getElementText(), " - ", link1a %>%
   ↪  getElementAttribute("href")))
10
11 link1b <- remDr %>% findElement('css selector',
   ↪  "#link_seccion_js")
12 print(paste("Seccion de JS:", link1b %>% getElementText(), " -
   ↪  ", link1b %>% getElementAttribute("href")))
13
14 remDr %>% deleteSession
```

Código fuente 76: Ejercicio de obtención de enlaces por su identificador con *seleniumPipes*

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
>
> #Se obtienen los enlaces
> links <- remDr %>% findElements('class', "dificultad_media")
> textos <- sapply(links, function(x){x %>% getElementText()})
> enlaces <- sapply(links, function(x){x %>% getElementAttribute("href")})
> tabla <- data.frame(textos, enlaces)
> View(tabla)
>
> links <- remDr %>% findElements('css selector', ".dificultad_alta")
> textos <- sapply(links, function(x){x %>% getElementText()})
> enlaces <-sapply(links, function(x){x %>% getElementAttribute("href")})
> tabla2 <- data.frame(textos, enlaces)
> View(tabla2)
>
> remDr %>% deleteSession
>
```

Ilustración 5.8: Resultado de ejecución del ejercicio de obtención de enlaces por su clase con *seleniumPipes*

```
library(seleniumPipes)
library(wdman)
selServ <- chrome(port = 5005L)
remDr <- remoteDr(port = 5005L)
remDr %>% go("http://tfg.daniel-fl.com/general/")


#Se obtienen los enlaces
links <- remDr %>% findElements('class', "dificultad_media")
textos <- sapply(links, function(x){x %>% getElementText()})
enlaces <- sapply(links, function(x){x %>%
 ↪  getElementAttribute("href")})
tabla <- data.frame(textos, enlaces)
View(tabla)

links <- remDr %>% findElements('css selector',
 ↪  ".dificultad_alta")
textos <- sapply(links, function(x){x %>% getElementText()})
enlaces <-sapply(links, function(x){x %>%
 ↪  getElementAttribute("href")})
tabla2 <- data.frame(textos, enlaces)
View(tabla2)

remDr %>% deleteSession
```

Código fuente 77: Ejercicio de obtención de enlaces por su clase con *seleniumPipes*

	textos	enlaces
1	ENVÍO DE FORMULARIOS	http://tfg.daniel-fl.com/forms
2	GESTIÓN DE COOKIES DE SESIÓN	http://tfg.daniel-fl.com/cookies

Ilustración 5.9: Visualización de *tabla* en el ejercicio de obtención de enlaces por su clase con *seleniumPipes*

	textos	enlaces
1	RENDERIZADO DE JAVASCRIPT	http://tfg.daniel-fl.com/test-js
2	PERFORMANCE	http://tfg.daniel-fl.com/performance

Ilustración 5.10: Visualización de *tabla2* en el ejercicio de obtención de enlaces por su clase con *seleniumPipes*

5.5.5. Obtener los atributos de una imagen

Se van a obtener los atributos básicos de una imagen. Como se comentaba en el caso de *rvest*, una imagen siempre tiene un atributo de ubicación (*src*). Además, es aconsejable que tenga un texto alternativo (atributo *alt*).

Para ello simplemente hay que localizar la imagen con su identificador y obtenerla haciendo uso de la función `findElement()` [94]. Posteriormente se extraen los atributos requeridos utilizando `getElementAttribute()` [107].

El código fuente completo se puede observar en el código fuente 78 y el resultado de su ejecución puede ser observado en la ilustración 5.11.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5  remDr %>% go("http://tfg.daniel-fl.com/general/")
6
7  #Se obtiene la imagen
8  img <- remDr %>% findElement('id', "imagen")
9
10 #Se extraen los atributos
11 print(paste("URL de la imagen: ", img %>%
   ↪  getElementAttribute("src")))
12 print(paste("ALT de la imagen: ", img %>%
   ↪  getElementAttribute("alt")))
13
14 remDr %>% deleteSession
```

Código fuente 78: Ejercicio de obtención de los atributos de una imagen con *seleniumPipes*

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se obtiene la imagen
> img <- remDr %>% findElement('id', "imagen")
>
> #Se extraen los atributos
> print(paste("URL de la imagen: ", img %>% getElementAttribute("src")))
[1] "URL de la imagen:  http://tfg.daniel-fl.com/general/pics/original-compress$
> print(paste("ALT de la imagen: ", img %>% getElementAttribute("alt")))
[1] "ALT de la imagen:  Fotografía de carretera convencional"
>
> remDr %>% deleteSession
>
```

Ilustración 5.11: Resultado de ejecución del ejercicio de obtención de los atributos de una imagen con *seleniumPipes*

5.5.6. Obtener diversas etiquetas de una clase y diferenciarlas

Como se comentaba en el caso de *rvest*, se han creado en el test de propósito general algunos elementos HTML en los párrafos con una coloración roja (que da lugar a la interpretación de que son elementos importantes). Dicha coloración roja se establece mediante una clase denominada ïmportante".

El objetivo es extraer todos los elementos que tengan esta clase, indicando a cual de los siguientes tres apartados corresponde: uno primero, con los elementos en línea *strong* que es un indicador de que es un texto al que se le quiere dar relevancia según los estándares del W3C [68], esto es, un texto importante; un segundo apartado en donde la clase es aplicada a un enlace (a) y un tercer apartado a modo de cajón de sastre donde se incluyen los elementos que no están bajo el paraguas de los dos apartados anteriores.

Para proceder con este ejercicio, en primer lugar se buscan todos los elementos que tengan como clase ïmportante" haciendo uso de la función `findElements()` [96].

Posteriormente se recorren en bucle los elementos encontrados y son clasificados verificando para cada elemento localizado qué tipo de etiqueta es con la función `getElementTagName()` [108].

En función de qué tipo de etiqueta sea, se extrae solo el texto con `getElementText()` [106] o también se extrae en el caso de un enlace su dirección con `getElementAttribute()` [107]. en el caso de que no se pudiera catalogar el elemento en ninguno de los dos casos mencionados anteriormente, se indicaría en un mensaje por pantalla.

Se puede observar de forma detallada este algoritmo en el código fuente 79. Además se puede observar la salida de ejecución en la ilustración 5.12 donde (al igual que en *rvest*) se puede observar que se han localizado dos elementos con la clase ïmportante", de los cuales uno es un enlace a una página del sitio web referida a *performance*, y la otra simplemente es un texto que se ha deseado destacar.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5  remDr %>% go("http://tfg.daniel-fl.com/general/")
6
7  #Se obtienen los elementos de importancia
8  importantes <- remDr %>% findElements('class', "importante")
9  for(imp in importantes){
10         tag <- imp %>% getElementTagName()
11         if(tag == "a"){
12                 print("ENLACE-------------------------")
13                 print(paste("    contenido: ",imp %>%
                    ↪  getElementText()))
14                 print(paste("    enlace: ",imp %>%
                    ↪  getElementAttribute("href")))
15         }else if(tag == "strong"){
16                 print("STRONG-------------------------")
17                 print(paste("    contenido: ",imp %>%
                    ↪  getElementText()))
18         }else{
19                 print("No es enlace ni strong")
20         }
21 }
22
23 remDr %>% deleteSession
```

Código fuente 79: Ejercicio de obtención de diversas etiquetas/elementos de una clase y su diferenciación con *seleniumPipes*

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los elementos de importancia
> importantes <- remDr %>% findElements('class', "importante")
> for(imp in importantes){
+ tag <- imp %>% getElementTagName()
+ if(tag == "a"){
+ print("ENLACE-------------------------")
+ print(paste("    contenido: ",imp %>% getElementText()))
+ print(paste("    enlace: ",imp %>% getElementAttribute("href")))
+ }else if(tag == "strong"){
+ print("STRONG-------------------------")
+ print(paste("    contenido: ",imp %>% getElementText()))
+ }else{
+ print("No es enlace ni strong")
+ }
+ }
[1] "STRONG-------------------------"
[1] "    contenido:  esto es un texto importante"
[1] "ENLACE-------------------------"
[1] "    contenido:  esto es un enlace importante relativo al performance"
[1] "    enlace:  http://tfg.daniel-fl.com/performance"
>
> remDr %>% deleteSession
>
```

Ilustración 5.12: Resultado de ejercicio de obtención de diversas etiquetas/elementos de una clase y su diferenciación con *seleniumPipes*

5.5.7. Obtener para cada enlace del cuerpo, a qué párrafo pertenece

Repasando las estrategias indicadas para resolver este ejercicio con *rvest* ya que son planteables para todos los paquetes que permiten realizar *web scraping*, dichas estrategias son las siguientes:

1. Recorrer todos los párrafos e iterarlos manualmente en busca de enlaces en ellos.

2. Recorrer todos los enlaces y acceder al elemento que lo contiene (párrafos).

Se opta por la segunda, ya que permite realizar un algoritmo menos complejo, y además permite probar el uso de *XPath* con *seleniumPipes* (ya que los selectores CSS no tienen en su especificación selectores en los que dado un elemento concreto se acceda a su antecesor).

En primer lugar, se accede a cada uno de enlaces que contiene el cuerpo con la función `findElements()` [96].

A continuación, se crean dos listas. Una primera con las URL's de los enlaces. Para ello se obtiene la lista de enlaces y mediante `sapply()` se obtiene otra lista con los enlaces extraídos de cada elemento mediante la función `getElementAttribute()` [107]. Por otro lado, se obtienen los párrafos, utilizando la misma estrategia de `sapply()` pero obteniendo los padres

con `findElementFromElement()` [95] y obteniendo los textos completos de los párrafos con `getElementText()` [106].

Finalmente se fusionan ambas listas para formar un *data frame*.

El código fuente 80 destaca cómo se realiza el algoritmo explicado de forma más detallada. Así mismo, se puede observar el resultado de la ejecución en la ilustración 5.13. Por último se puede observar la tabla generada con los enlaces y los párrafos padre en la ilustración 5.14.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5
6  remDr %>% go("http://tfg.daniel-fl.com/general/")
7
8  #Se obtienen los enlaces del cuerpo
9  enlaces <- remDr %>%
10             findElements('css selector',"main a")
11 urls <- sapply(enlaces, function(x){x %>%
   ↪  getElementAttribute("href")})
12
13 #Se obtienen los padres (parrafos) con XPath ya que con CSS no
   ↪  se puede ascender en el DOM
14
15 parrafos <- sapply(enlaces, function(x){
16                  x %>% findElementFromElement('xpath',"..") %>%
17                  getElementText()})
18
19 #Se genera la tabla con enlaces y sus parrafos
20 tabla <- data.frame(urls, parrafos)
21 View(tabla)
22
23 remDr %>% deleteSession
```

Código fuente 80: Ejercicio de obtención de enlaces y párrafos con *seleniumPipes*

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen los enlaces del cuerpo
> enlaces <- remDr %>%
+             findElements('css selector',"main a")
> urls <- sapply(enlaces, function(x){x %>% getElementAttribute("href")})
>
> #Se obtienen los padres (parrafos) con XPath ya que con CSS no se puede ascen
>
> parrafos <- sapply(enlaces, function(x){
+ x %>% findElementFromElement('xpath',"..") %>%
+ getElementText()})
>
> #Se genera la tabla con enlaces y sus parrafos
> tabla <- data.frame(urls, parrafos)
> View(tabla)
```

Ilustración 5.13: Resultado de ejecución del ejercicio de obtención de enlaces y párrafos con *seleniumPipes*

	urls	parrafos
1	http://tfg.daniel-fl.com/performance	Mientras que esto es un texto importante que tien>
2	http://tfg.daniel-fl.com/forms	Tras la finalización de este supuesto, se debe em>

Ilustración 5.14: Tabla con resultado de ejecución del ejercicio de obtención de enlaces y párrafos con *seleniumPipes*

5.5.8. Ver los enlaces hermanos de los textos importantes

Para resolver este ejercicio, es necesario ineludiblemente el uso de *XPath* ya que CSS no provee ningún operador para elegir dos hermanos de distinto tipo no adyacentes dentro del mismo padre.

En primer lugar, se extrae con el selector *XPath* conveniente todos los nodos que tengan simultáneamente al menos un hijo enlace `a` y un hijo importante `strong`, contando con la ayuda de la función `findElements()` [96].

Posteriormente para cada nodo padre se imprime el texto que lo compone (texto de los párrafos) con `getElementText()` [106]. Luego se obtienen los elementos `strong` y `a` y se imprime su contenido como se ha comentado anteriormente. Además en el caso de los enlaces también se imprime su dirección URL gracias a extraerla mediante `getElementAttribute()` [107].

Se puede observar el algoritmo desarrollado en R en el código fuente 81 y una muestra del resultado de ejecución en la ilustración 5.15.

```
library(seleniumPipes)
library(wdman)
selServ <- chrome(port = 5005L)
remDr <- remoteDr(port = 5005L)

remDr %>% go("http://tfg.daniel-fl.com/general/")

#Se extraen aquellos padres que tengan como hijos a la vez nodos importantes y nodos enlaces
padres <- remDr %>% findElements('xpath',"//*[strong][a]")

#Para cada nodo de los seleccionados,
#se extrae su contenido y luego las partes de enlaces y textos importantes
for(padre in padres){

        print(paste("+Padre: ", padre %>% getElementText))

        print("-Textos importantes:")
        sapply(padre %>% findElementsFromElement('tag name',"strong"), function(x){
                x %>% getElementText %>% print
        })

        print("-Enlaces:")
        sapply(padre %>% findElementsFromElement('tag name',"a"), function(x){
                print(paste(x %>% getElementText, " - ", x %>% getElementAttribute("href")))
        })
}
remDr %>% deleteSession
```

Código fuente 81: Ejercicio de obtención de de enlaces hermanos de textos importantes con *seleniumPipes*

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se extraen aquellos padres que tengan como hijos a la vez nodos importantes $
> padres <- remDr %>% findElements('xpath',"//*[strong][a]")
>
> #Para cada nodo de los seleccionados,
> #se extrae su contenido y luego las partes de enlaces y textos importantes
> for(padre in padres){
+
+ print(paste("+Padre: ", padre %>% getElementText))
+
+ print("-Textos importantes:")
+ sapply(padre %>% findElementsFromElement('tag name',"strong"), function(x){
+ x %>% getElementText %>% print
+ })
+
+ print("-Enlaces:")
+ sapply(padre %>% findElementsFromElement('tag name',"a"), function(x){
+ print(paste(x %>% getElementText, " - ", x %>% getElementAttribute("href")))
+ })
+ }
[1] "+Padre:  Mientras que esto es un texto importante que tiene una clase impo$
[1] "-Textos importantes:"
[1] "esto es un texto importante"
[1] "-Enlaces:"
[1] "esto es un enlace importante relativo al performance  -  http://tfg.daniel$
> remDr %>% deleteSession
>
```

Ilustración 5.15: Resultado de ejecución de ejercicio de obtención de de enlaces hermanos de textos importantes con *seleniumPipes*

5.5.9. Obtener la tabla que hay en el cuerpo

Desafortunadamente *seleniumPipes* no admite de forma nativa la extracción de una tabla. Se podría proceder de dos formas:

1. Realizar un algoritmo que extraiga la tabla, accediendo en primer lugar a las celdas de título para generar un *data frame* que se iría rellenando con el contenido de todas las celdas.

2. Utilizar un paquete externo que permita la resolución de este aspecto sin necesidad diseñar este algoritmo

Se opta finalmente por esta segunda alternativa, siendo la primera posible objetivo a realizar en un futuro como ampliación de este documento. La razón de optar por esta segunda opción es que el propio autor (John D. Harrison) ya respondió a la comunidad cuando esta pregunta surgió respecto a la herramienta hermana *RSelenium* [120]. Él indica a la persona que tiene la duda que utilice la función `readHTMLTable()` de XML para extraer la tabla.

En el caso que se ha realizado derivado de ese se procede en primer lugar a importar el paquete *XML*. A continuación se obtiene la página como se ha hecho siempre con `go()` [90]. A continuación se extrae el elemento de tabla deseado con `findElement()` [94]. Por último, se procede a extraer toda la tabla en HTML. Para ello se hará uso de un atributo del DOM denominado outerHTML. Este atributo se extraerá con la función `getElementAttribute()` [107].

Tras ello empieza a trabajar *XML parseando* en primer lugar la tabla con `htmlParse()` y luego transformándolo en una tabla con la función `readHTMLTable()` [67].

Tras ello, se genera una tabla y se imprime posteriormente. Todo este proceso se puede observar en el código fuente 82 y en la ilustración 5.16. Además, se puede observar la tabla extraída en la ilustración 5.17. Hay que destacar que bien el paquete *XML* o su interacción con *seleniumPipes* da algunos problemas, respecto a algunos problemas en la codificación y en la presentación de la tabla que habrá que mirarlos.

```r
library(seleniumPipes)
library(wdman)
library(XML)

selServ <- chrome(port = 5005L)
remDr <- remoteDr(port = 5005L)

remDr %>% go("http://tfg.daniel-fl.com/general/")

#Se obtienen la tabla que hay (solo hay una)
tabla <- remDr %>% findElement('tag name', "table")
codigoTabla <- tabla %>% getElementAttribute("outerHTML")
docTabla <- htmlParse(codigoTabla[[1]])
tabla <- readHTMLTable(docTabla, header = TRUE)

View(tabla)

remDr %>% deleteSession
```

Código fuente 82: Ejercicio de obtención de tabla con *seleniumPipes*

```
> remDr %>% go("http://tfg.daniel-fl.com/general/")
>
> #Se obtienen la tabla que hay (solo hay una)
> tabla <- remDr %>% findElement('tag name', "table")
> codigoTabla <- tabla %>% getElementAttribute("outerHTML")
> docTabla <- htmlParse(codigoTabla[[1]])
> tabla <- readHTMLTable(docTabla, header = TRUE)
>
> View(tabla)
>
> remDr %>% deleteSession
>
```

Ilustración 5.16: Resultado de ejecución de ejercicio de obtención de tabla con *seleniumPipes*

	NULL.Lunes	NULL.Martes	NULL.MiÃƒÂ.rcoles	NULL.Jueves	NULL.Viernes
1	Ensalada	Paella	Sopa de arroz	Espaguetis a la carbonara	JudÃƒÂ-as con jamÃƒÂ³n
2	Pasta con tomate	Guisantes	AlbÃƒÂ³ndigas	Arroz con conejo	Cocido completo
3	Lentejas con chorizo	Plato combinado	Croquetas	Fabada	Pescado
4	Lomo con patatas	Redondo de ternera	Zarzuela de pescado	Guisado de costillas	Croquetas

Ilustración 5.17: Tabla de ejecución del ejercicio de obtención de tabla con *seleniumPipes*

5.6. Realización del test de envío de formularios con seleniumPipes

El objetivo de este ejercicio es observar cómo se puede interactuar con el servidor mediante *seleniumPipes* haciendo un envío de un formulario y observando si la respuesta obtenida es la esperada.

Para empezar, tras hacer la petición a la página de formularios con `go()` [90], se debe obtener la caja en la que se desea escribir. Esta caja está identificada con un identificador y se puede acceder a ella mediante la función `findElement()` [94]. Tras ello, se envía la cadena deseada (en este caso mi nombre). Para hacer ese envío de cadena, se debe hacer uso de una nueva función: `elementSendKeys()` [115].

Tras ello, se debe enviar el formulario. Para ello hay que hacer clic en el botón de envío. Este botón puede ser localizado con un selector CSS ya que es el único botón de envío presente en el documento web. Nuevamente se puede realizar con `findElement()` [94]. Para hacer clic se utiliza la nueva función `elementClick()` [117].

Por último queda extraer el nombre con las funciones `findElement` [94] y `getElementText()` [106]. Si ese nombre existe se imprime en pantalla (lo que implicará que la interacción cliente-servidor ha sido correcta). Si no, se observará un mensaje de error.

Se puede observar este algoritmo de forma detallada en el código fuente 83. Se puede observar el funcionamiento correcto de este algoritmo, ya que se ha impreso *Daniel* en la pantalla, tal y como se observa en la ilustración 5.18.

```
> remDr %>% go("http://tfg.daniel-fl.com/forms/")
>
> #Se busca la caja y se escribe en ella
> remDr %>% findElement('id', "caja_nombre") %>% elementSendKeys("Daniel")
>
> #Se clica en enviar
> remDr %>% findElement('css selector', "input[type='submit']") %>% elementClick
>
> #Se intenta extraer el nombre
> nombre <- remDr %>% findElement('id', "nombre") %>% getElementText()
> if (exists("nombre")){
+ print(nombre)
+ }else{
+ print("La peticion no se ha procesado correctamente. No existe nombre")
+ }
[1] "Daniel"
> remDr %>% deleteSession
>
```

Ilustración 5.18: Resultado de ejecución del ejercicio de envío de formularios con *seleniumPipes*

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5  remDr %>% go("http://tfg.daniel-fl.com/forms/")
6
7  #Se busca la caja y se escribe en ella
8  remDr %>% findElement('id', "caja_nombre") %>%
   ↪  elementSendKeys("Daniel")
9
10 #Se clica en enviar
11 remDr %>% findElement('css selector', "input[type='submit']")
   ↪  %>% elementClick
12
13 #Se intenta extraer el nombre
14 nombre <- remDr %>% findElement('id', "nombre") %>%
   ↪  getElementText()
15 if (exists("nombre")){
16         print(nombre)
17 }else{
18         print("La peticion no se ha procesado correctamente. No
           ↪  existe nombre")
19 }
20 remDr %>% deleteSession
```

Código fuente 83: Test de envío de formularios con *seleniumPipes*

5.7. Realización del test de gestión de cookies con seleniumPipes

El objetivo de este ejercicio es observar cómo *seleniumPipes*, al basar su modo de funcionamiento en el control de un navegador, es capaz de gestionar de gestionar la problemática de las sesiones de forma natural, no perdiéndose la *cookie* de sesión y gestionándose adecuadamente sin necesidad de hacer ninguna variación en el algoritmo (al contrario de lo que sucede en *seleniumPipes*).

Para hacer este supuesto en primer lugar se crea un *data frame* en el que se va a ir almacenando el número de veces que se hace una visita a la página.

Posteriormente en un bucle de 5 iteraciones, para cada iteración se accede a la página concreta del test con la función `go()` [90]. A continuación se localiza el número indicado (marcado con un identificador particular a la hora de hacer el test) con la función `findElement()` [94] y se extrae con la función `getElementText()` [106]. Por último antes de realizar la próxima iteración se inserta el número obtenido en el *data frame*.

El algoritmo utilizado para hacer este ejercicio se encuentra en el código fuente 84 y el resultado de su ejecución en la ilustración 5.19.

En la ilustración 5.20 se puede observar que el número de visitas crece correlativamente, por lo que en efecto, el navegador envía (como era de esperar) correctamente el identificador de sesión almacenado en la *cookie* por lo que el servidor localiza la sesión correspondiente y accede sin problema al contador en el estado en el que permanecía en la anterior petición.

```
1  library(seleniumPipes)
2  library(wdman)
3  selServ <- chrome(port = 5005L)
4  remDr <- remoteDr(port = 5005L)
5
6  df <-data.frame(Visitas=integer())
7
8  for(i in 1:5){
9          remDr %>% go("http://tfg.daniel-fl.com/cookies/")
10         elemVis <- remDr %>% findElement("id","visitas")
11         visitas <- elemVis %>% getElementText()
12         df[nrow(df) + 1,] = c(visitas)
13 }
14
15 View(df)
16 remDr %>% deleteSession
```

Código fuente 84: Verificación de mantenimiento transparente de sesiones con *seleniumPipes*

```
> remDr <- remoteDr(port = 5005L)
>
> df <-data.frame(Visitas=integer())
>
> for(i in 1:5){
+ remDr %>% go("http://tfg.daniel-fl.com/cookies/")
+ elemVis <- remDr %>% findElement("id","visitas")
+ visitas <- elemVis %>% getElementText()
+ df[nrow(df) + 1,] = c(visitas)
+ }
>
> View(df)
> remDr %>% deleteSession
>
```

Ilustración 5.19: Resultado de ejecución de la verificación de mantenimiento transparente de sesiones con *seleniumPipes*

	Visitas
1	0
2	1
3	2
4	3
5	4

Ilustración 5.20: Tabla generada en el resultado de ejecución de la verificación de mantenimiento transparente de sesiones con *seleniumPipes*

5.8. Realización del test de renderizado de JavaScript con seleniumPipes

El objetivo de este ejercicio es determinar si *seleniumPipes* es capaz de detectar los elementos insertados en el DOM como consecuencia de los datos obtenidos por una invocación mediante JavaScript al servidor y su tratamiento posterior.

Debido a que con *seleniumPipes* lo que se hace realmente es controlar de forma automática un navegador, y el navegador que se está utilizando (Chrome) sí que renderiza estos componentes, se va a intentar directamente *parsear* el contenido dinámico (los profesionales) ya que (como se podrá ver a continuación) sí que se concluye la carga de la página, tanto la llamada síncrona inicial como la asíncrona posterior, y por lo tanto para *seleniumPipes* esos profesionales cargados asíncronamente existen.

El supuesto se inicia buscando las cajas de profesionales y verificando su existencia con la función `findElements()` [96]. Si existe al menos una caja se continúa con su análisis. En caso contrario se lanza un mensaje de error.

Se crea un *data frame* para almacenar todos los datos de los profesionales: nombre, edad, descripción y ubicación de su fotografía.

Para cada profesional se obtienen los datos anteriomente mencionados. Dichos datos se extraen gracias a las funciones de *seleniumPipes* `findElementFromElement()` [95], `getElementText()` [106] y `getElementAttribute()` [107] y posteriormente se añaden al *data frame*. Posteriormente este *data frame* se mostrará en pantalla.

El algoritmo para la extracción de profesionales se puede observar en el código fuente 85. Una imagen mostrando su ejecución puede ser observada en la ilustración 5.21. Por último, se pueden observar los datos de los profesionales extraídos en la ilustración 5.22.

```
> remDr %>% go("http://tfg.daniel-fl.com/test-js/")
>
> profesionales <- remDr %>% findElements('css selector', "#profesionales .caja{
>
> if(exists("profesionales") && (length(profesionales) > 0)){
+ df <-data.frame(Nombre=character(), Edad=integer(), Descripcion=character(), {
+ for(p in profesionales){
+ nombre <- p %>% findElementFromElement('tag name',"h3") %>% getElementText()
+ edad <- p %>% findElementFromElement('class',"edad") %>% getElementText() %>%{
+ descr <- p %>% findElementFromElement('css selector',"p:last-child") %>% getE{
+ url_pic <- p %>% findElementFromElement('tag name',"img") %>% getElementAttri{
+
+ nuevo<-c(nombre, edad, descr, url_pic)
+ df[nrow(df) + 1,] = nuevo
+ }
+ View(df)
+ }else{
+ print("Elemento no encontrado. Se esta cargando o bien el navegador no admite{
+ }
> remDr %>% deleteSession
>
```

Ilustración 5.21: Resultado de ejecución del test de renderizado de JavaScript con *seleniumPipes*

	Nombre	Edad	Descripcion	Foto
1	Max Power	22	Persona con una gran formación pero poca experien>	http://tfg.daniel-fl.com/test-js/pics/person1.jpeg
2	Emma Mc'Calister	26	A pesar de su juventud, es una de las cientificas>	http://tfg.daniel-fl.com/test-js/pics/person2.jpeg
3	John Doe	36	Persona curtida en entornos altamente competitivo>	http://tfg.daniel-fl.com/test-js/pics/person3.jpeg

Ilustración 5.22: Tabla del resultado de ejecución del test de renderizado de JavaScript con *seleniumPipes*

```
library(seleniumPipes)
library(wdman)
library(readr)

selServ <- chrome(port = 5005L)
remDr <- remoteDr(port = 5005L)
remDr %>% go("http://tfg.daniel-fl.com/test-js/")

profesionales <- remDr %>% findElements('css selector',
 ↪  "#profesionales .caja")

if(exists("profesionales") && (length(profesionales) > 0)){
        df <-data.frame(Nombre=character(), Edad=integer(),
         ↪  Descripcion=character(), Foto=character(),
         ↪  stringsAsFactors=FALSE)
        for(p in profesionales){
                nombre <- p %>% findElementFromElement('tag
                 ↪  name',"h3") %>% getElementText()
                edad <- p %>%
                 ↪  findElementFromElement('class',"edad") %>%
                 ↪  getElementText() %>% parse_number
                descr <- p %>% findElementFromElement('css
                 ↪  selector',"p:last-child") %>%
                 ↪  getElementText()
                url_pic <- p %>% findElementFromElement('tag
                 ↪  name',"img") %>% getElementAttribute("src")

                nuevo<-c(nombre, edad, descr, url_pic)
                df[nrow(df) + 1,] = nuevo
        }
        View(df)
}else{
        print("Elemento no encontrado. Se esta cargando o bien
         ↪  el navegador no admite carga de JavaScript")
}
remDr %>% deleteSession
```

Código fuente 85: Test de renderizado de JavaScript con *seleniumPipes*

5.9. Conclusiones de seleniumPipes

Para finalizar con **seleniumPipes** se procede a indicar algunas conclusiones individuales de este paquete. En la tabla que aparece posteriormente, se puede observar una serie de datos de interés, tales como la dificultad de los supuestos planteados así como algunos datos de carácter general. Las conclusiones individuales extraídas son las siguientes:

- La instalación es algo complicada al principio, algunas de las maneras de instalarlo no funcionaron correctamente (desconozco si es que hay algún matiz que no se realizó correctamente o si realmente dan problemas a la comunidad de usuarios).
- La curva de aprendizaje es baja. Aunque hay un número en cierta medida elevado de funciones, tienen nombres muy sencillos que representan bastante bien la funcionalidad que realizan.
- El uso de notación en tubería, facilita mucho la labor de generar los scripts.
- Dispone de una gran variedad de selectores, por lo que no se perciben límites a la hora de realizar consultas.
- Debido a que se controla un navegador, toda la funcionalidad interactiva (sesiones, formularios, ejecución asíncrona de *JavaScript*) que se ha probado en los tests ha funcionado perfectamente.

PARÁMETROS GENERALES	
Dificultad de instalación	Dificil
Complejidad de la sintaxis	Baja-media
Pendiente de aprendizaje	Baja
TEST DE PROPÓSITO GENERAL	
Ejemplo de petición HTTP	Fácil
Obtener metaetiqueta del título de la página, descripción y palabras clave	Fácil
Obtener todos los enlaces de la barra de navegación	Fácil
Obtener enlaces por su clase y por su identificador	Medio
Obtener los atributos de una imagen	Fácil
Obtener diversas etiquetas de una clase y diferenciarlos	Fácil
Obtener para cada enlace del cuerpo, a qué párrafo pertenece	Medio
Ver los enlaces hermanos de los textos importantes	Medio
Obtener la tabla que hay en el cuerpo	Medio (no nativo, se usa XML)
TEST DE ENVÍO DE FORMULARIOS	
Envío de información a través de formularios	Fácil
TEST DE GESTIÓN DE COOKIES DE SESIÓN	
Mantenimiento de sesiones	Fácil
TEST DE RENDERIZADO DE JAVASCRIPT	
Webscraping ante situaciones de carga asíncrona del DOM	Fácil

Capítulo 6

Benchmark básico de los paquetes de web scraping. Conclusiones finales

6.1. Introducción

Una vez que ya se han realizado los tests de supuestos prácticos y se ha visto qué características proveen los distintos paquetes, se procede a realizar un último test, cuyo objetivo es medir de forma aproximada el rendimiento o *performance* a la hora de descargar, *parsear* y extraer la información deseada.

Para diseñar el test se han tenido en cuenta los siguientes detalles:

1. El test de *performance* **debe poder ser ejecutados en todos los paquetes elegidos** Por esta razón no se pueden utilizar páginas donde la información a *scrapear* se inserte en el DOM de forma asíncrona como resultado de la ejecución de una función JavaScript. Si no fuera así no se podrían probar tanto *XML* como *rvest*.

2. Las páginas que se vayan a descargar deben ser parecidas entre sí y deben requerir todas el mismo selector para realizar la extracción. Esto es importante porque simplifica el diseño de las funciones y además permite hacer comparaciones (al tener las páginas un tamaño parecido).

3. Todas las funciones deben ser ejecutadas en el mismo equipo, y se debe procurar que dicho equipo no tenga una conexión a Internet que pueda suponer un cuello de botella a la hora de hacer el *web scraping*. De igual manera, la fuente de datos debe tener un ancho de banda suficiente para realizar una petición sin problemas (no se creará un algoritmo que haga descargas paralelas ya que esto complica el algoritmo).

4. Las direcciones URL deben poder ser predecibles, utilizando números correlativos o fechas para acceder a la información deseada recorriendo todas

las páginas. La idea de esto es simplificar el problema al máximo, ya que si se requiriera diseñar toda una lista de enlaces o una lógica para obtener enlaces de las web recorridas y realimentar la entrada (*web crawling*) complicaría en exceso el algoritmo.

La determinación que se ha seguido es utilizar los reportes en HTML del balance eléctrico de Red Eléctrica de España (REE). En concreto se va a extraer el balance diario de producción y consumo diarios producid por fuentes de energía renovables (en GWh) en la península de forma diaria. En la siguiente sección se explica el test de forma concreta y la lógica común a utilizar.

6.1.1. Advertencia

Se ha utilizado este servicio de Red Eléctrica de España para hacer estas pruebas. Sin embargo, no se aconseja que se utilice de forma regular, ya que estas pruebas pueden afectar al rendimiento del servidor. Además, como se indicó en la introducción a este trabajo, es aconsejable utilizar *web scraping* cuando no exista otro medio mejor a través del que obtener la información. En el caso de REE, existe una API totalmente gratuita denominada *Esios* a través de la que se pueden obtener los datos del balance eléctrico de forma mucho más sencilla y afectando menos al rendimiento.

6.2. Descripción del test de performance

6.2.1. Elección de la web. Cumplimiento de condiciones requeridas

Como se indicaba anteriormente se van a obtener datos del balance eléctrico de Red Eléctrica de España. En particular se va a obtener la generación diaria de energía eléctrica que proviene de fuentes renovables. Se extrae este dato ya es un dato de fácil acceso dentro de una tabla.

Se ha elegido esta web para realizar la extracción ya que tiene los criterios que se han comentado de forma anterior:

1. El dato que se desea extraer aparece en la página en el DOM inicial que se obtiene nada más hacer la descarga, y no proviene de una interacción posterior mediante *JavaScript*. Por lo tanto aquellos paquetes que hacen renderizado como aquellos que no pueden leer dicho dato. Este hecho puede ser fácilmente contrastado si se desactiva *JavaScript* en el navegador. Esto puede ser realizado mediante las herramientas de desarrollo de Google Chrome, o mediante otro tipo de complementos o de opciones que existen en el resto de navegadores. Si se hace la petición de la página con *JavaScript* desactivado se puede observar que la información está en el documento descargado (y parcialmente renderizado, por lo que podrá haber características de la página que no se puedan observar o se observen de forma diferente, generalmente peor).

2. Los reportes que se van a descargar son todos iguales desde el año 2013, ya que la información que se genera respecto de la generación y el consumo de energía eléctrica es la misma y es inusual que varíe. Los datos de interés son los mismos también, y dado que es un reporte que genera Red Eléctrica de España de forma automática, esto garantiza que el orden de los nodos del árbol DOM siempre va a ser el mismo en todos y cada uno de los reportes (salvo que modifiquen el algoritmo de visualización). En la ilustración 6.1 se puede observar que hay varias tablas y unos gráficos (que posiblemente se ejecuten en *JavaScript* o *Flash* y que realmente no afectan a la tarea de *web scraping* ya que no se van a extraer). Hay una tabla en particular rotulada ."Energía renovable (en GWh)". De esta tabla se extraerá la generación realizada en el día. En todos los días que se observan desde el 1 de enero 2013, la información se estructura de igual manera.

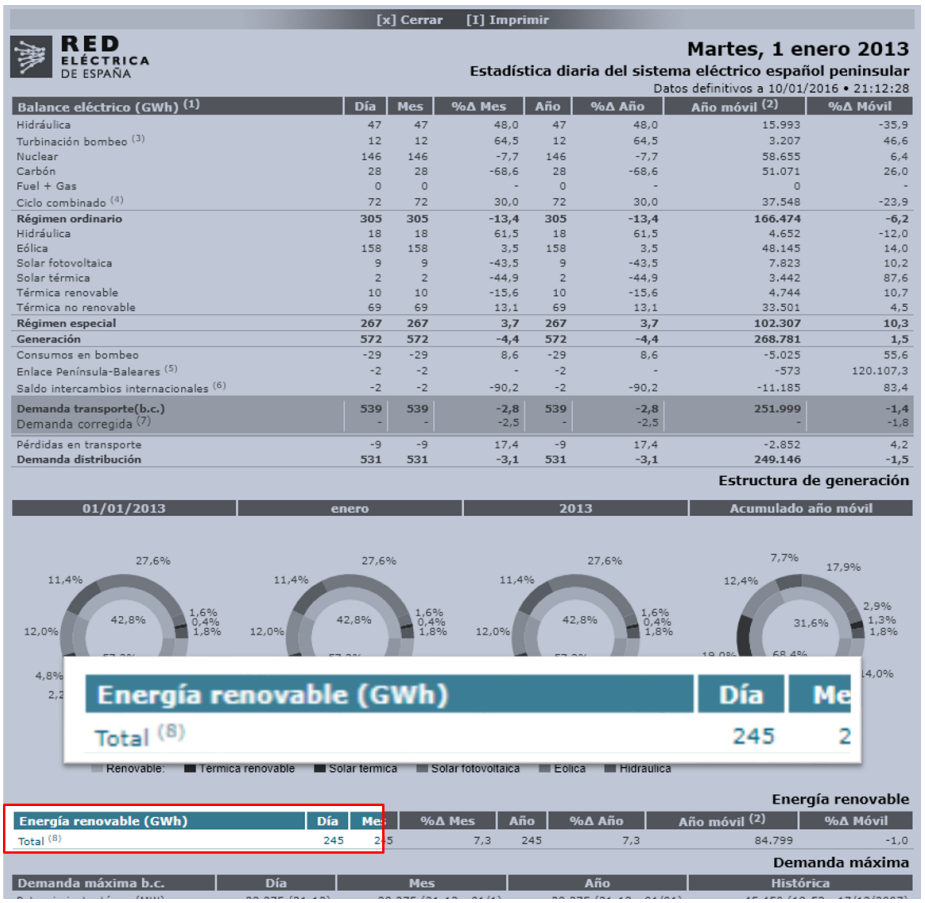
3. El punto 3 no afecta realmente al ejemplo utilizado, será de aplicación al realizar la ejecución cumpliendo los supuestos indicados.

4. Las URL's son predecibles, ya que hay tres parámetros por la URL que permiten indicar la fecha del balance correspondiente al día indicado. En la ilustración 6.2 se observa un ejemplo de URL, para el 1 de enero de 2013, donde se indica cual es cada parámetro de fecha.

Los parámetros relativos al día y al mes han de indicarse en la URL con dos cifras numéricas mientras que el año debe ser representado con cuatro cifras.

6.2.2. Búsqueda del selector indicado para la extracción del dato requerido

Una vez que ya se ha visto que los balances diarios de estadística diaria del sistema eléctrico español peninsular cumplen con los criterios deseados, ahora simplemente se debe saber qué selector utilizar para extraer la información deseada. Si se dispone de *Google Chrome*, se puede obtener dicho selector de forma sencilla. Para ello se debe acceder una vez que se ha accedido a través del navegador a un reporte de un día concreto al que se desee hacer *web scraping* o estudiarlo a las herramientas de desarrollo web, pero a través del elemento directamente. Para ello se hace clic con el botón derecho en el elemento que se desea extraer y se hace clic en el menú contextual en "Inspeccionar...". Una vez realizado se puede observar que aparece en la parte derecha del navegador el código fuente de la página, apareciendo el foco sobre el elemento elegido. Simplemente hay que posicionarse sobre él, hacer clic en el botón derecho, acceder a "Copy" y dentro hacer clic en "Copy XPath". Con este simple hecho se pasa a tener en el portapapeles el selector concreto para acceder a ese dato. Se ha optado por elegir el selector en *XPath* ya que es el único que es compatible con los tres paquetes que se quieren probar. Si se deseara obtener el selector CSS simplemente habría que hacer clic en "Copy selector".



[x] Cerrar [I] Imprimir

RED ELÉCTRICA DE ESPAÑA

Martes, 1 enero 2013

Estadística diaria del sistema eléctrico español peninsular

Datos definitivos a 10/01/2016 • 21:12:28

Balance eléctrico (GWh) (1)	Día	Mes	%Δ Mes	Año	%Δ Año	Año móvil (2)	%Δ Móvil
Hidráulica	47	47	48,0	47	48,0	15.993	-35,9
Turbinación bombeo (3)	12	12	64,5	12	64,5	3.207	46,6
Nuclear	146	146	-7,7	146	-7,7	58.655	6,4
Carbón	28	28	-68,6	28	-68,6	51.071	26,0
Fuel + Gas	0	0	-	0	-	0	-
Ciclo combinado (4)	72	72	30,0	72	30,0	37.548	-23,9
Régimen ordinario	**305**	**305**	**-13,4**	**305**	**-13,4**	**166.474**	**-6,2**
Hidráulica	18	18	61,5	18	61,5	4.652	-12,0
Eólica	158	158	3,5	158	3,5	48.145	14,0
Solar fotovoltaica	9	9	-43,5	9	-43,5	7.823	10,2
Solar térmica	2	2	-44,9	2	-44,9	3.442	87,6
Térmica renovable	10	10	-15,6	10	-15,6	4.744	10,7
Térmica no renovable	69	69	13,1	69	13,1	33.501	4,5
Régimen especial	**267**	**267**	**3,7**	**267**	**3,7**	**102.307**	**10,3**
Generación	**572**	**572**	**-4,4**	**572**	**-4,4**	**268.781**	**1,5**
Consumos en bombeo	-29	-29	8,6	-29	8,6	-5.025	55,6
Enlace Península-Baleares (5)	-2	-2	-	-2	-	-573	120.107,3
Saldo intercambios internacionales (6)	-2	-2	-90,2	-2	-90,2	-11.185	83,4
Demanda transporte(b.c.)	**539**	**539**	**-2,8**	**539**	**-2,8**	**251.999**	**-1,4**
Demanda corregida (7)	-	-	-2,5	-	-2,5	-	-1,8
Pérdidas en transporte	-9	-9	17,4	-9	17,4	-2.852	4,2
Demanda distribución	**531**	**531**	**-3,1**	**531**	**-3,1**	**249.146**	**-1,5**

Estructura de generación

01/01/2013	enero	2013	Acumulado año móvil

Energía renovable (GWh)	Día	Me
Total (8)	245	2

Renovable: Térmica renovable Solar térmica Solar fotovoltaica Eólica Hidráulica

Energía renovable

Energía renovable (GWh)	Día	Mes	%Δ Mes	Año	%Δ Año	Año móvil (2)	%Δ Móvil
Total (8)	245	245	7,3	245	7,3	84.799	-1,0

Demanda máxima

Demanda máxima b.c.	Día	Mes	Año	Histórica

Ilustración 6.1: Ejemplo de reporte deL balance diario de *Red Eléctrica de España*

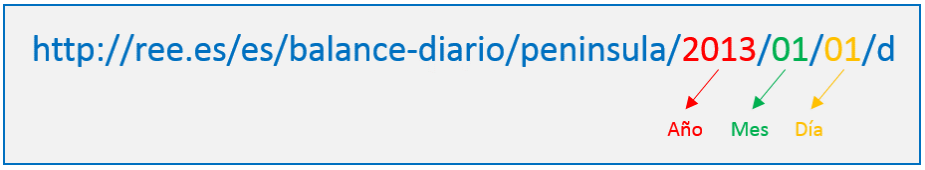


Ilustración 6.2: Ejemplo de URL deL balance diario de *Red Eléctrica de España*

Este proceso para la obtención del selector se puede observar de forma gráfica en la ilustración 6.3.

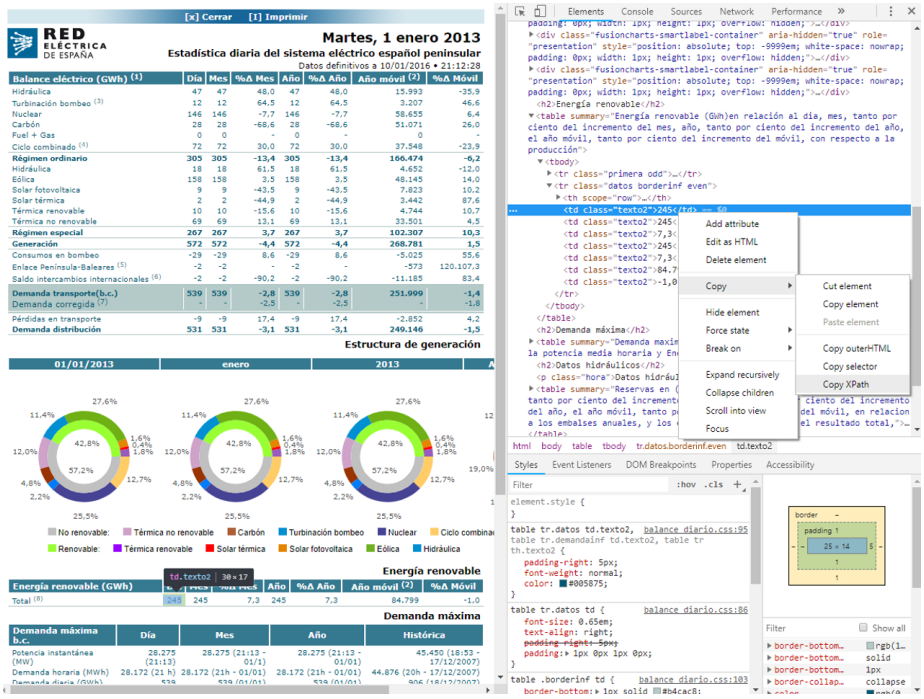


Ilustración 6.3: Ejemplo de obtención de selector con las herramientas para desarrolladores de *Google Chrome*

El selector que permite obtener la generación se puede observar en la ilustración 6.4.

```
/html/body/table[3]/tbody/tr[2]/td[1]
```

Ilustración 6.4: Selector para la obtención de la generación diaria de energía de fuentes renovables

6.2.3. Algoritmo a ejecutar para la extracción

El algoritmo que se va a realizar va a ser el mismo en todos los programas. Consistirá en una función cuyo objetivo será en base a un número de días re-

cibido, descargar desde el 1 de enero de 2013 tantas páginas como se hayan indicado y almacenar en un *data frame* los GWh renovables generados. Esta función habrá que realizarla para cada paquete y se realizarán las siguientes operaciones:

1. Se recibirá como parámetro de entrada un número entero que indica el número de días que se desea *scrapear*.
2. Se genera un *data frame* que permitirá almacenar para cada día la generación correspondiente.
3. Se genera una primera fecha, haciendo uso del paquete *lubridate*.
4. Mientras que los días transcurridos sean menores que el número de días establecido:
   a) Se generan el mes y el día en dos variables diferentes, procurando respetar el formato requerido de la URL (dos números para cada uno).
   b) Se crea la URL nueva, concatenando de forma correcta el día, el mes y el año (para ello se utiliza `paste0()`, que no introduce ningún carácter entre las cadenas a concatenar).
   c) Se descarga la página y se realiza la consulta deseada dentro del árbol con ayuda del selector *XPath*.
   d) Se suma 1 a los días ya recorridos y se suma un día a la fecha actual.
5. Por último se devuelve el *data frame*.

6.3. Implementación de las funciones del test de performance

Una vez explicado el test de *performance*, se procede a mostrar las funciones concretas realizadas.

6.3.1. Implementación del test de performance en XML

El código fuente 86 contiene la función que se invocará indicando el número de días que se desean obtener, haciendo *web scraping* con *XML*.

```
renovablesXML <- function(dias){

  library(XML)
  library(lubridate)

  df <-data.frame(gwh=integer())

  fecha <- ymd("2013-01-01")
  xp <- "/html/body/table[3]/tbody/tr[2]/td[1]"
  i <- 1

  while(i <= dias){
    mes <- format.Date(fecha, "%m")
    dia <- format.Date(fecha, "%d")
    url <-
     ↪  paste0("http://www.ree.es/es/balance-diario/peninsula/",
     ↪  year(fecha), "/", mes, "/", dia, "/d")

    pagina <- htmlParse(url)
    nodosNumero <- getNodeSet(pagina, xp)
    numero <- xmlValue(nodosNumero[[1]])

    df[nrow(df) + 1,] = numero

    fecha <- fecha + 1
    i <- i + 1
  }

  return(df)

}
```

Código fuente 86: Función para la ejecución del test de *performance* con *XML*

6.3.2. Implementación del test de performance en rvest

El código fuente 87 contiene la función que se invocará indicando el número de días que se desean obtener, haciendo *web scraping* con *rvest*.

```
renovablesRvest <- function(dias){

  library(rvest)
  library(lubridate)

  df <-data.frame(gwh=integer())

  fecha <- ymd("2013-01-01")
  xp <- "/html/body/table[3]/tbody/tr[2]/td[1]"
  i <- 1

  while(i <= dias){
    mes <- format.Date(fecha, "%m")
    dia <- format.Date(fecha, "%d")
    url <-
     ↪  paste0("http://www.ree.es/es/balance-diario/peninsula/",
     ↪  year(fecha), "/", mes, "/", dia, "/d")

    pagina <- read_html(url)
    numero <- pagina %>% html_node(xpath=xp) %>% html_text

    df[nrow(df) + 1,] = numero
    fecha <- fecha + 1
    i <- i + 1
  }

  return(df)

}
```

Código fuente 87: Función para la ejecución del test de *performance* con *rvest*

6.3.3. Implementación del test de performance en seleniumPipes

El código fuente 88 contiene la función que se invocará indicando el número de días que se desean obtener, haciendo *web scraping* con *seleniumPipes*.

Cabe destacar que este código es un poco diferente al resto ya que se abre

```
renovablesSeleniumPipes <- function(dias){

  library(seleniumPipes)
  library(wdman)
  library(lubridate)

  #La generacion del aleatorio es necesaria, ya que parece ser
   ↪  que a veces el puerto se queda ocupado tras utilizarlo
   ↪  anteriormente
  puerto <- sample.int(10000:65000, 1)[1]
  selServ <- chrome(port = puerto)
  remDr <- remoteDr(port = puerto)

  df <-data.frame(gwh=integer())

  fecha <- ymd("2013-01-01")
  xp <- "/html/body/table[3]/tbody/tr[2]/td[1]"
  i <- 1

  while(i <= dias){
    mes <- format.Date(fecha, "%m")
    dia <- format.Date(fecha, "%d")
    url <-
     ↪  paste0("http://www.ree.es/es/balance-diario/peninsula/",
     ↪  year(fecha), "/", mes, "/", dia, "/d")

    remDr %>% go(url)

    numero <-remDr %>%
    findElement('xpath',xp) %>% getElementText

    df[nrow(df) + 1,] = numero
    fecha <- fecha + 1
    i <- i + 1
  }

  remDr %>% deleteSession

  return(df)

}
```

Código fuente 88: Función para la ejecución del test de *performance* con *seleniumPipes*

el servidor *Selenium* con *wdman* y se procede a la apertura del navegador solamente una vez, y que al finalizar la navegación se cierra. Además, se detectaron problemas a la hora de reabrir el navegador en siguientes iteraciones, dado que el sistema mantenía bloqueado el puerto que se había utilizado. Dado que en principio se ha realizado el cierre del navegador a través de **seleniumPipes** tal y como se indica en la documentación mediante el uso de `deleteSession()` [102], simplemente se ha procedido a indicarle al software que escoja automáticamente un número aleatorio de puerto para abrir el servidor de *Selenium* (fuera del rango de puertos bien conocidos, esto es, entre el 1024 y el 65535, aunque siendo generosos, se ha partido desde 10000 ya que algunos puertos son utilizados por otras aplicaciones propias en el ordenador en el que se ha ejecutado este algoritmo). De esta manera, el programa no ha vuelto a dar fallo que sea relativo a que el puerto estaba ocupado.

6.4. Ejecución de las funciones del test de performance

Como se ha indicado con anterioridad, habrá que medir el rendimiento a la hora de ejecutar estas funciones. Existen bastantes formas de medir el rendimiento en R [121]. En este caso concreto se va a utilizar un paquete denominado **microbenchmark**, que realizará la ejecución de los supuestos que se le indiquen un número de veces indicado y arrojará algo de estadística descriptiva en relación a la duración de las actividades o casos. Se ha optado por repetir los tests 5 veces. Además, los tests abarcarán la obtención de 7, 15, 30, 60, 90, 180 y 360 días, de tal manera que se va incrementando progresivamente el volumen de las peticiones realizadas.

En el código fuente 89 se puede observar cómo se hace uso de **microbenchmark** dentro de la función `ejecutarBenchmarkN()` y cómo se hace uso de cada una de las implementaciones para cada paquete y en el código fuente 90 se puede observar cómo se llama a dicha función para el número de días deseado.

```
ejecutarBenchmarkN <- function(n){

  library(microbenchmark)

  df <- microbenchmark(
    "XML" = {
      renovablesXML(n)
    },
    "rvest" = {
      renovablesRvest(n)
    },
    "SeleniumPipes" = {
      renovablesSeleniumPipes(n)
    },
    times = 5
  )

  return(df)
}
```

Código fuente 89: Ejecución de *microbenchmark* para un número de días recibido por parámetro

```
source("xmlfunc.R")
source("rvestfunc.R")
source("spfunc.R")
source("ejecutarn.R")

for(dias in c(7, 15, 30, 60, 90, 180, 360)){
    df <- ejecutarBenchmarkN(dias)

    View(df)

    save(df, file = paste0("performance-report-", dias,
    ↪  ".rdata"))

    Sys.sleep(60)
  }
}
```

Código fuente 90: Invocación de la función *ejecutarBenchmarkN()*

6.5. Resultados de la ejecución de las funciones del test de performance

A continuación se podrán observar los datos estadísticos obtenidos como consecuencia de ejecutar 5 veces cada test. Los datos que se podrán observar son el valor mínimo de duración, el primer cuartil, la media, la mediana, el tercer cuartil y la duración máxima.

6.5.1. Resultado de la ejecución para 7 días

En este caso, analizando un poco la tabla mostrada en la ilustración 6.5, se puede observar que en los casos de **XML** y de **rvest**, las duraciones son más o menos iguales (en el caso de **rvest** se puede observar que de las 5 repeticiones parece ser que ha habido una que ha tardado un 20 por ciento más que el tercer cuartil, por lo que puede deberse a una fluctuación en la conexión a Internet o en la respuesta del servidor y se podría tratar como si de un *outlier* a descartar se tratara). No obstante, en el caso de **seleniumPipes** se puede observar que la duración es bastante más significativa, aproximadamente unas 10 veces superior. Esto es debido a que este paquete tiene la fase intermedia de renderizado y ejecución de *JavaScript*, que no tiene el resto. Además, habría que contar el tiempo de la apertura y del cierre de *Selenium* y el navegador correspondiente (en este caso *Google Chrome*).

```
> load(file="performance-report-7.rdata")
> df
Unit: milliseconds
          expr       min        lq       mean    median        uq       max neval
           XML  868.2438  880.9913   951.3141  947.5529  1005.371  1054.412     5
         rvest  865.4294  900.5772  1000.4490  994.8627  1014.801  1226.575     5
 SeleniumPipes 9409.3679 9496.5174 10081.8983 9733.9915 10438.405 11331.209     5
> |
```

Ilustración 6.5: Ejecución del test de *performance* para 7 días

6.5.2. Resultado de la ejecución para 15 días

En el caso de esta ejecución las cuestiones relativas son significativamente parecidas. Tanto **XML** como **rvest** son bastante más rápidos que **seleniumPipes**. No obstante, las diferencias se han estrechado, siendo **seleniumPipes** esta vez unas 8,5 veces más lento que las otras dos opciones. Esto es debido a que al recorrer más dias con el mismo navegador y la misma instancia de *Selenium*, el tiempo de apertura y de cierre se va diluyendo conforme se va incrementando el número de páginas visitadas. Si se obtiene una media de duración por página, para **seleniumPipes** en el caso de 7 días, la media sería de 1,44 segundos, mientras que con 15 días es de 1,11 segundos. Respecto a los líderes, se observa que **rvest** es aproximadamente un 7 por ciento más rápido.

```
> load(file="performance-report-15.rdata")
> df
Unit: seconds
          expr       min        lq      mean    median        uq       max neval
           XML  1.904435  2.114103  2.185832  2.181732  2.312040  2.416849     5
         rvest  1.756781  1.962246  2.030805  2.063047  2.130398  2.241551     5
 SeleniumPipes 16.000250 16.047907 16.673244 17.058846 17.079707 17.179510     5
> |
```

Ilustración 6.6: Ejecución del test de *performance* para 15 días

6.5.3. Resultado de la ejecución para 30 días

Se repìte el mismo escenario que antes para **seleniumPipes**, donde se observa que nuevamente es el paquete más lento, aunque en este caso la media por página no supera el segundo. Se observa nuevamente que hay un *outlier* en este caso en **seleniumPipes**, donde en una de las ejecuciones tardó unos 10 segundos más que la media. Esto puede ser debido a alguna ralentización en el servidor o bien debido a problemas con la conexión a Internet.

En el caso de **rvest** y **XML**, en esta ocasión ha sido más rápido de media **XML**, aunque si se observa el rango de los datos, ha ganado debido a que en general las duraciones han sido bastante dispersas (duración máxima menos la mínima). Sin embargo aunque **rvest** ha sido un poco más lento, ha sido más regular en las duraciones. Sin embargo a estas escalas es poco relevante, ya que estas fluctuaciones pueden ser debidas a la conexión de red, ya que el protocolo IP es un protocolo *best effort* y no garantiza que todos los paquetes de datos recorran la misma ruta.

```
> load(file="performance-report-30.rdata")
> df
Unit: seconds
          expr       min        lq      mean    median        uq       max neval
           XML  3.749687  3.868825  4.132023  4.101253  4.212150  4.728198     5
         rvest  4.031841  4.143254  4.259060  4.203942  4.393975  4.522286     5
 SeleniumPipes 28.004264 28.008737 30.566422 28.485673 28.494652 39.838782     5
> |
```

Ilustración 6.7: Ejecución del test de *performance* para 30 días

6.5.4. Resultado de la ejecución para 60 días

La situación se repite, **seleniumPipes** vuelve a ser el más lento, vuelve a haber un *outlier* en este caso nuevamente en **seleniumPipes** y **rvest** vuelve a ser ligeramente más rápido.

```
> load(file="performance-report-60.rdata")
> df
Unit: seconds
          expr       min        lq      mean    median        uq       max neval
           XML  8.036271  8.305469  8.798338  8.459657  9.439839  9.750455     5
         rvest  7.977821  8.093486  8.610986  8.863917  8.938684  9.181021     5
 SeleniumPipes 48.488934 50.767503 56.297601 51.704254 52.677943 77.849372     5
>
```

Ilustración 6.8: Ejecución del test de *performance* para 60 días

6.5.5. Resultado de la ejecución para 90, 180 y 360 días

Estos resultados se fusionan ya que las circunstancias vuelven a ser parecidas. Sin embargo ya se puede observar que conforme se aumentan el número de páginas, se incrementa el número de *outliers* que se observan. Se puede observar que en el desarrollo de 90, 180 y 360 días hay un *outlier* en el caso del paquete **seleniumPipes**, mientras que en el 360 días también lo hay en el caso de **rvest**.

Si se calcula la media de duración por página en el caso de **seleniumPipes** tomando como valor total la mediana (la media no vale, ya que se ve desvirtuada por el *outlier*), se observa que la media ha descendido a 0,75 segundos por página en el caso de 360 días, mientras que para 90 días era de 0,85. En el caso de 180 días era de 0.6 segundos, por lo que se puede observar una cierta anomalía, en la que en el caso de 180 días cae este valor estrepitosamente. Habría que investigar más concretamente este caso, ya que puede deberse a que los parámetros del test (velocidad, respuestas del servidor) cambiaran inadvertidamente.

Respecto a los otros dos paquetes, **rvest** y **XML**, se observa que el rendimiento ha caído un poco, aunque los dos son bastante rápidos. Se observa como se mencionaba antes que en el caso de **rvest** en el valor máximo se puede observar un *outlier* porque es un valor muy alto. Sin embargo, si se observa detenidamente en **XML**, se observa que todos los valores por encima del tercer cuartil son anormalmente altos. Esto puede ser debido a que el servidor, al detectar una gran cantidad de peticiones de la misma IP haya determinado resolver con más lentitud las peticiones.

```
> load(file="performance-report-90.rdata")
> df
Unit: seconds
          expr      min       lq     mean   median       uq      max neval
           XML 11.68912 12.82027 13.29083 13.11012 14.10968  14.72495     5
         rvest 12.42142 12.76123 13.08095 12.93590 13.50704  13.77918     5
 SeleniumPipes 36.03379 73.29088 74.41233 76.33698 77.19081 109.20920     5
>
```

Ilustración 6.9: Ejecución del test de *performance* para 90 días

```
> load(file="performance-report-180.rdata")
> df
Unit: seconds
          expr       min        lq      mean    median        uq       max neval
           XML  26.68041  27.46261  32.26099  31.04004  33.82030  42.30158     5
         rvest  27.41115  29.44906  53.72894  31.05152  40.77162 139.96133     5
 SeleniumPipes 101.76802 104.11677 106.45650 108.46901 108.46903 109.45968     5
> |
```

Ilustración 6.10: Ejecución del test de *performance* para 180 días

```
> load(file="performance-report-360.rdata")
> df
Unit: seconds
          expr       min        lq      mean    median        uq       max neval
           XML  49.99196  55.88139  68.48419  66.08274  83.29870  87.16616     5
         rvest  51.69129  52.62608 134.67904  53.14352  54.19436 461.73996     5
 SeleniumPipes 237.34607 252.54881 264.19894 271.69867 279.49729 279.90385     5
> |
```

Ilustración 6.11: Ejecución del test de *performance* para 360 días

6.6. Conclusiones finales sobre este trabajo

Para finalizar se extraen una serie de conclusiones. En el caso de que la web requiera de *JavaScript* para cargar los datos que se requieren, será necesario el uso de **seleniumPipes** ya que aunque es más lento, garantiza que este tipo de webs se cargarán y se *parsearán* sin ningún tipo de problema.

En caso contrario, si se tiene la certeza de que nada más realizar la descarga de la web, los datos deseados ya están allí sin necesidad de realizar ningún esfuerzo adicional de renderizado, **XML** y **rvest** son más que aconsejables. En relación a su rendimiento y en base a los datos obtenidos, no se puede concluir que ninguno de los dos es más rápido que el otro, ya que en ciertas ocasiones ganaba **rvest** pero en el caso de **XML** se observaba que había muestras que eran mejores que **rvest** y otras que eran peores. Por lo tanto la elección de uno de estos dos paquetes no está motivada respecto al rendimiento, si no que lo está respecto a las funcionalidades que proporcionan.

La elección final en caso de no requerir el renderizado comentado anteriormente, sería **rvest**, ya que provee más funcionalidades para hacer *web scraping* (en detrimento de la creación de ficheros XML, que en este caso no es relevante). **rvest** provee una sintaxis mucho más sencilla y mantenible gracias al uso de tuberías, permite utilizar selectores CSS (que en el ámbito web se utilizan mucho más y en general suelen ser más sencillos). Además, trabaja con envíos de formularios y trabaja bien en el caso de requerir sesiones. El único test que falla es el de renderizado de *JavaScript*. En caso de que se requiriera a posteriori esta funcionalidad, hay paquetes como **rdom** o **decapitated**, las cuales podrían facilitar esa fase de renderizado de *JavaScript* si así se necesitara.

Como última conclusión, los datos obtenidos de rendimiento, especialmente

para un número elevado de días demuestran que para realizar una prueba de *performance* y obtener resultados más precisos, es aconsejable crear un test propio con una gran batería de datos a *scrapear* y cargarlo en un servidor web local. Además sería interesante ejecutar en local (o en una intranet) los tests de *performance*. Incluso podría ser más óptimo cargar las páginas como ficheros. Esto se haría por las siguientes dos razones:

- Dado que el servidor sería propio, se anularía la política que impide o reduce el rendimiento en el caso de grandes peticiones. Otra opción que permiten las herramientas sería cargar ficheros en vez de direcciones URL. Esto permitiría evitar el uso de un servidor web y posiblemente acercaría los resultados a la duración mínima teórica.
- Se evitarían los posibles cuellos de botella y al minimizar la ruta es más probable que dicha ruta fuera la óptima. Dado que el protocolo IP es un protocolo *best effort* donde no se garantiza que los paquetes van por la ruta más óptima, ni se garantiza el orden ni la completitud ni corrección de los paquetes (esto se hace en la capa de transporte TCP y tiene un coste). Utilizar archivos en este caso también podría ser lo más óptimo ya que se evita este problema.

6.7. Futuras líneas de progreso

Tras realizar este trabajo, merece la pena detenerse a pensar en las cuestiones que se considera que podrían realizarse para mejorar o progresar en el área de *web scraping* y en lo que es relativo a este trabajo:

- Avanzar en conceptos más concretos del *web scraping* tales como las pruebas de software web automatizadas o el *web crawling* (un posible candidato a probar para esto último es **rCrwaler**) [35]. También sería bastante interesante ver **htmltab** [36] que está especializado en extracción de información de tablas.
- Continuar trabajando con más paquetes que permitan renderizado de *JavaScript*. Esto es muy importante ya que la web está evolucionando hacia un modelo donde se incrementa la interactividad y donde se transfiere al lado del cliente la funcionalidad. Esto quiere decir que naturalmente el número de webs que pueden ser *scrapeadas* sin renderizado y ejecución de *JavaScript* está disminuyendo.
- Mejorar los tests de *performance* para reducir la interferencia de la red o del servidor, montando dichos tests en un ordenador propio.
- Avanzar en la generación de valor mediante la información, viendo el potencial de obtenerla, ordenarla, analizarla y representarla, viendo el *web scraping* como una fase de una cadena de generación de valor denominada *data science*. El conjunto de paquetes **tidyverse** [122] puede ser útil en esta tarea.

Anexos

Anexo A

Encadenamiento de funciones con magrittr

A.1. Introducción a magrittr

Magrittr es un paquete de *tidyverse* que proporciona un conjunto de operadores entre los que se destacar el operador de tubería principal (%>%) que permite facilitar el paso de parámetros entre funciones aportando las siguientes cualidades [85][123][122][123]:

- Permite estructurar secuencias de operaciones de datos de izquierda a derecha con facilidad.
- Evita el uso de invocaciones anidadas a funciones.
- Minimiza el uso de variables locales y definiciones de funciones.
- Facilita introducir pasos adicionales en cualquier parte de la secuencia de operaciones.

Además del operador principal existen más operadores, pero no se van a exponer ya que no son útiles en el contexto en que se van a utilizar.

A.2. El operador de tubería principal (%>%)

El operador tubería es un operador infijo (esto es, que se utiliza entre dos operandos), que permite facilitar la transferencia del resultado de una función (que está representada a la izquierda) a la siguiente. A lo largo de los supuestos planteados, se utilizará mucho el operador infijo %>% (también conocido como operador tubería). El operador tubería sirve para pasar el resultado retornado por una función a otra cuya invocación es subsiguiente y cuyo primer parámetro depende del resultado de la función anterior [85][123]. Exactamente, se realiza

el paso del resultado de la invocación a la función *n* al primer parámetro de la invocación a la función *n+1*. En caso de que el resultado de la función previa deba situarse en la siguiente función como parámetro en posición diferente de la primera, se invocará dicha función subsiguiente indicando con un punto el lugar al que debe ir dicho parámetro.

A continuación, en las ilustraciones A.1 y A.2 se podrá observar un esquema simplificado en el que varias funciones son invocadas tomando cada función sucesiva el valor de la función precedente, observándose la sintaxis cuando se utiliza el operador de tubería y cuando se decide no utilizar.

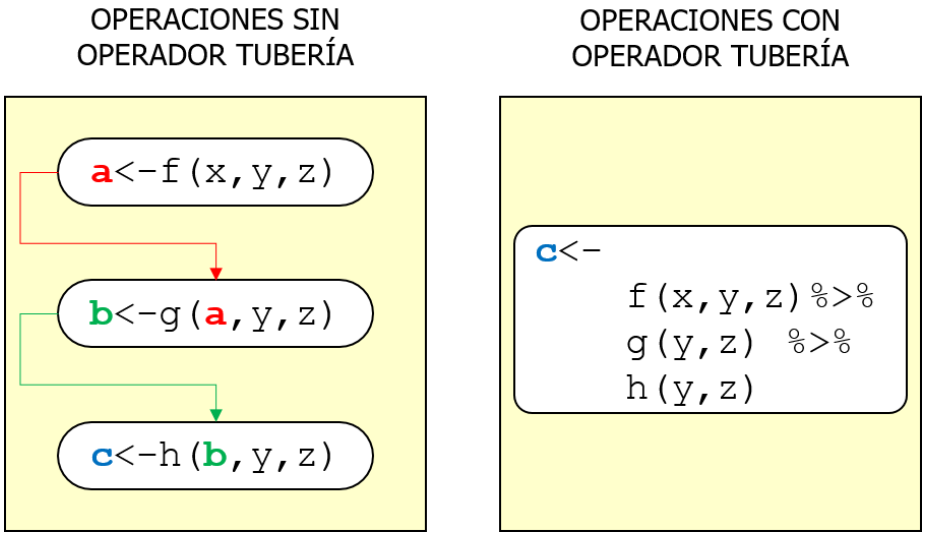


Ilustración A.1: Supuestos de invocación a funciones subsecuentes utilizando el operador tubería requiriendo las funciones sucesivas del resultado de las que les precede a través del primer parámetro

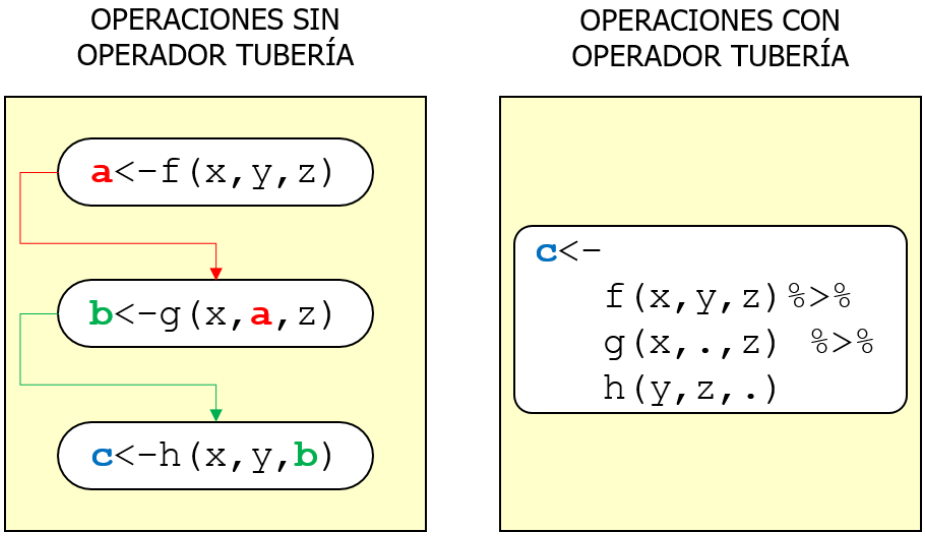


Ilustración A.2: Supuestos de invocación a funciones subsecuentes utilizando el operador tubería requiriendo las funciones sucesivas del resultado de las que les precede a través de cualquier orden de parámetros determinado

Índice de códigos fuente

Índice de ilustraciones

Bibliografía

[1] "Headless browser - wikipedia." [Online]. Available: https://es.wikipedia.org/wiki/Web_scraping

[2] "Web scraping - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Web_scraping

[3] S. vanden Broucke; Bart Baesens, *Practical Web Scraping for Data Science: Best Practices and Examples with Python.* Apress, 2018.

[4] "Web crawler - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Web_crawler

[5] "Grabación de pruebas con selenium y junit — marco de desarrollo de la junta de andalucía." [Online]. Available: http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/385

[6] "Selenium y la automatización de las pruebas — marco de desarrollo de la junta de andalucía." [Online]. Available: http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/381

[7] "¿es legal el web scraping? : De webscraping y legalidad." [Online]. Available: http://www.spri.eus/euskadinnova/es/enpresa-digitala/agenda/scraping-usos-herramientas-ejemplos-legalidad-vigente/13913.aspx

[8] "Web scraping: usos, herramientas, ejemplos y legalidad vigente - euskadi+innova." [Online]. Available: https://diariodeuneletrado.wordpress.com/2017/03/22/es-legal-el-web-scrapping-de-webscrapping-y-legalidad/

[9] "Web scraping - juan riaza." [Online]. Available: http://www.spri.eus/euskadinnova/documentos/3198.aspx

[10] "The comprehensive r archive network." [Online]. Available: https://cran.r-project.org/

[11] "Rdocumentation." [Online]. Available: https://www.rdocumentation.org/

[12] "R-forge project." [Online]. Available: https://r-forge.r-project.org/

[13] "The omega project for statistical computing." [Online]. Available: http://www.omegahat.net/

[14] "Bioconductor - home." [Online]. Available: https://www.bioconductor.org/

[15] "Github." [Online]. Available: https://github.com/

[16] T. Leeper, S. Chamberlain, P. Mair, K. Ram, and C. Gandrud, "Cran task view: Web technologies and services." [Online]. Available: https://cran.r-project.org/web/views/WebTechnologies.html

[17] D. Temple and C. Team, "Cran - package xml." [Online]. Available: https://cran.r-project.org/web/packages/XML/index.html

[18] L. Duncan Temple, "An xml package for the s language." [Online]. Available: http://www.omegahat.net/RSXML/

[19] ——, "Memory management in the the xml package." [Online]. Available: http://www.omegahat.net/RSXML/MemoryManagement.html

[20] "Xml: Tools for parsing and generating xml within r and s-plus." [Online]. Available: https://cran.r-project.org/web/packages/XML/index.html

[21] "Xml package, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML

[22] H. Wickham, H. James, O. Jeroen, RStudio, and more, "Cran - package xml2." [Online]. Available: https://cran.r-project.org/web/packages/xml2/index.html

[23] ——, "Parse xml • xml2." [Online]. Available: http://xml2.r-lib.org/

[24] ——, "xml2 package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/xml2

[25] H. Wickham, "Easily harvest (scrape) web pages package, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/

[26] ——, "Github - hadley/rvest: Simple web scraping for r." [Online]. Available: https://github.com/hadley/rvest

[27] ——, "Readme - rvest." [Online]. Available: https://cran.r-project.org/web/packages/rvest/README.html

[28] S. Potter, S. Simon, and B. Ian, "Cran - package selectr." [Online]. Available: https://cran.r-project.org/web/packages/selectr/index.html

[29] ——, "selectr package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/selectr

[30] R. M. Acton, "Cran - package scraper." [Online]. Available: https://cran.r-project.org/web/packages/scrapeR/index.html

[31] ——, "scraper package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/scrapeR/

[32] ——, "scrape function — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/scrapeR/topics/scrape

[33] S. Khalil, "Cran - package rcrawler." [Online]. Available: https://cran.r-project.org/web/packages/Rcrawler/index.html

[34] ——, "Rcrawler package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/Rcrawler

[35] ——, "Github - salimk/rcrawler: An r web crawler and scraper." [Online]. Available: https://github.com/salimk/Rcrawler

[36] C. Rubba, "htmltab package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/htmltab

[37] ——, "Cran - package htmltab." [Online]. Available: https://cran.r-project.org/web/packages/htmltab/index.html

[38] ——, "Hassle-free html tables with htmltab." [Online]. Available: https://cran.r-project.org/web/packages/htmltab/vignettes/htmltab.html

[39] J. D. Harrison, "Cran - package xml2r." [Online]. Available: https://cran.r-project.org/web/packages/XML2R/index.html

[40] M. Annau, "Package 'boilerpiper'." [Online]. Available: https://cran.r-project.org/web/packages/boilerpipeR/boilerpipeR.pdf

[41] ——, "Short introduction to boilerpiper." [Online]. Available: https://cran.r-project.org/web/packages/boilerpipeR/vignettes/ShortIntro.pdf

[42] C. Sievert and W. Chang, "Github - cpsievert/rdom: Render and parse dynamic web pages from r." [Online]. Available: https://github.com/cpsievert/rdom

[43] "Problem with vignette example · issue #9 · cpsievert/rdom · github." [Online]. Available: https://github.com/cpsievert/rdom/issues/9

[44] C. Sievert, "Multiple urls · issue #6 · cpsievert/rdom · github." [Online]. Available: https://github.com/cpsievert/rdom/issues/6

[45] L. Ramon, "relenium package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/relenium

[46] ——, "Github - lluisramon/relenium: Easy and powerful web scraping with selenium." [Online]. Available: https://github.com/LluisRamon/relenium

[47] B. Rudis, "splashr package — r documentation." [Online]. Available: https://www.rdocumentation.org/packages/splashr

[48] ——, "Cran - package splashr." [Online]. Available: https://cran.r-project.org/web/packages/splashr/index.html

[49] J. D. Harrison, "Rpubs - rselenium: Basics." [Online]. Available: https://rpubs.com/johndharrison/RSelenium-Basics

[50] ——, "Basics - rselenium." [Online]. Available: http://ropensci.github.io/RSelenium/articles/basics.html

[51] "Webdriver - w3c candidate recommendation." [Online]. Available: https://w3c.github.io/webdriver/webdriver-spec.html

[52] J. Harrison, "Cran - package seleniumpipes." [Online]. Available: https://cran.r-project.org/web/packages/seleniumPipes/index.html

[53] ——, "seleniumpipes - readme." [Online]. Available: https://cran.r-project.org/web/packages/seleniumPipes/README.html

[54] ——, "seleniumpipes: Basic operation." [Online]. Available: https://cran.r-project.org/web/packages/seleniumPipes/vignettes/basicOperation.html

[55] G. Csárdi, "Cran - package webdriver." [Online]. Available: https://cran.r-project.org/web/packages/webdriver/index.html

[56] ——, "webdriver package - r documentation." [Online]. Available: https://www.rdocumentation.org/packages/webdriver/

[57] ropenscilabs, "Github - ropenscilabs/decapitated: Chrome headless but rly websockets." [Online]. Available: https://github.com/ropenscilabs/decapitated

[58] G. Sanchez, "Parsing xml, stat 133, university of california-berkeley." [Online]. Available: http://www.gastonsanchez.com/stat133/slides/33-parsing-xml/33-parsing-xml.pdf

[59] "The comprehensive r archive network." [Online]. Available: http://cran.rediris.es/

[60] "xmltreeparse function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/xmlTreeParse

[61] "xmlstructuredstop function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/xmlStructuredStop

[62] "getnodeset function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/getNodeSet

[63] "xmlvalue function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/xmlValue

[64] "xmlgetattr function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/xmlGetAttr

[65] "xmlattrs function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/xmlAttrs

[66] "xmlname function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/xmlName

[67] "readhtmltable function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/XML/topics/readHTMLTable

[68] "Html strong tag - w3schools.com." [Online]. Available: https://www.w3schools.com/tags/tag_strong.asp

[69] D. Lang, "Rcurl package, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/RCurl/versions/1.95-4.11

[70] H. Wickham, "httr package, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/httr/

[71] J. Hester, "xml2 package, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/xml2/

[72] H. Wickham, "html/xml_read function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html

[73] ——, "html_nodes/html_node functions, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html_nodes

[74] ——, "html_text/html_name/html_children/html_attrs/html_attr functions, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html_text

[75] ——, "html_table function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html_table

[76] ——, "html_form function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html_form

[77] ——, "set_values function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/set_values

[78] ——, "submit_form function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/submit_form

[79] ——, "html_session function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html_session

[80] ——, "html_jump_to function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/html_jump_to

[81] ——, "jump_to: Navigate to a new url. in rvest: Easily harvest (scrape) web pages." [Online]. Available: https://rdrr.io/cran/rvest/man/jump_to.html

[82] ——, "session_history function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/session_history

[83] ——, "encoding functions, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/encoding

[84] ——, "pluck function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/rvest/topics/pluck

[85] S. M. Bache, "magrittr package - r documentation." [Online]. Available: https://www.rdocumentation.org/packages/magrittr

[86] C. Yau, "Data frame — r tutorial." [Online]. Available: http://www.r-tutor.com/r-introduction/data-frame

[87] J. Eguiluz, "2.1. selectores básicos." [Online]. Available: http://librosweb.es/libro/css/capitulo_2/selectores_basicos.html

[88] J. Hester, "url_absolute function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/xml2/topics/url_absolute

[89] "Spider trap - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Headless_browser

[90] J. Harrison, "go function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/go

[91] ——, "back function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/back

[92] ——, "forward function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/forward

[93] ——, "refresh function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/refresh

[94] ——, "findelement function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/findElement

[95] ——, "getpagesource function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/findElementFromElement

[96] ——, "findelements function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/findElements

[97] ——, "findelementsfromelement function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/findElementsFromElement

[98] ——, "remotedr function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/remoteDr

[99] ——, "retry function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/retry

[100] ——, "retry: Documetation of retry argument." [Online]. Available: https://rdrr.io/cran/seleniumPipes/man/retry.html

[101] ——, "getcurrenturl function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getCurrentUrl

[102] ——, "deletesession function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/deleteSession

[103] ——, "gettitle function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getTitle

[104] ——, "getpagesource function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getPageSource

[105] ——, "getactiveelement function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getActiveElement

[106] ——, "getelementtext function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getElementText

[107] ——, "getelementattribute function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getElementAttribute

[108] ——, "getelementtagname function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getElementTagName

[109] ——, "getelementcssvalue function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getElementCssValue

[110] ——, "getelementrect function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/getElementRect

[111] ——, "getelementrect: Return the dimensions and coordinates of an element." [Online]. Available: https://rdrr.io/cran/seleniumPipes/man/getElementRect.html

[112] w3schools, "Html disabled attribute." [Online]. Available: https://www.w3schools.com/tags/att_disabled.asp

[113] J. Harrison, "iselementenabled function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/isElementEnabled

[114] ——, "iselementselected function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/isElementSelected

[115] ——, "elementsendkeys function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/elementSendKeys

[116] ——, "elementclear function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/elementClear

[117] ——, "elementclick function, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes/topics/elementClick

[118] ——, "seleniumpipes package, r documentation." [Online]. Available: https://www.rdocumentation.org/packages/seleniumPipes

[119] ——, "Man pages for seleniumpipes, rdrr.io." [Online]. Available: https://rdrr.io/cran/seleniumPipes/man/

[120] "r - how to read an html table using rselenium? - stack overflow." [Online]. Available: https://stackoverflow.com/questions/29932542/how-to-read-an-html-table-using-rselenium

[121] A. Gossmann, "5 ways to measure running time of r code – alexej gossmann." [Online]. Available: https://www.alexejgossmann.com/benchmarking_r/

[122] "Tidyverse packages - tidyverse." [Online]. Available: https://www.tidyverse.org/packages/

[123] "A forward-pipe operator for r - magrittr." [Online]. Available: https://magrittr.tidyverse.org/

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá