COMP4047 Internet and World Wide Web

Tutorial

**Reading Web Pages in Java**

There are many ways to write a program to download web pages in Java. The simpler way is using the URL class with its input stream. The following example function is used to download a web page and return a string.

String loadWebPage(String urlString) {

byte[] buffer = **new** byte[1024];

String content = **new** String();

**try** {

URL url = **new** URL(urlString);

InputStream in = url.openStream();

int len;

**while**((len = in.read(buffer)) != -1)

content += **new** String(buffer);

} **catch** (IOException e) {

content = "<h1>Unable to download the page</h1>" + urlString;

}

**return** content;

}

Without any error, the string contains the full content of the target web page including HTML tags and values. If you want to remove all HTML tags. So, you need to use Regular Expression to match the HTML tags and delete them.

# Retrieving Text from Web Page

The *javax.swing.text.html.HTMLEditorKit* package provides a parser for parsing web pages. With the parser, we can download a web page that the HTML tags and the text of the target web page are separated.

## ParserCallback Class

The **ParserCallback** class provides the methods for handling the web content. It has the following methods:

|  |  |
| --- | --- |
| handleStartTag() | Invoked when the parser reaches a start tag, such as <p>, <div>, etc. |
| handleEndTag() | Invoked when the parser reaches an end tag, such as </p>, </div>, etc. |
| handleSimpleTag() | Invoked when the parser reaches a tag with no body, such as <hr/>,  <img />, etc. |
| handleText() | Invoked when the parser reaches the body text of a tag. |
| handleComment() | Invoked when the parser reaches a comment string. |
| handleError() | Invoked when the parser has an error parsing the content. |
| handleEndofLineString() | Invoked when the parser finishes parsing the content. |

Before we can use the parser, we must define our own **ParserCallback** class to specify how to handle the web content.

## Defining a New ParserCallback Class

The **handleText()** method is useful for scraping the web content in the plain text format (not tags). So, we now define a new **ParserCallback** class and override the **handleText()** method for outputting a plain text content of a target web page.

**import** **javax.swing.text.html.\***;

**import** **javax.swing.text.html.HTML.\***;

**import** **javax.swing.text.html.HTMLEditorKit.\***;

**import** **javax.swing.text.html.parser.\***;

**import** **javax.swing.text.\***;

**class** **MyParserCallback** **extends** HTMLEditorKit.ParserCallback {

**public** String content = **new** String();

**@Override**

**public** void handleText(char[] data, int pos) {

content += " " + **new** String(data);

}

}

The **handleText()** method accepts two parameters – **data** and **pos**. The parameter **data** contains the body text of a tag, and **pos** indicates the starting position of the body text. Inside the method, we append the body text of a tag to the string **content**.

## Using the Parser

Next, we can make use of parser with the newly defined **MyParserCallback** class to scrape the web content.

String loadPlainText(String urlString) **throws** IOException {

MyParserCallback callback = **new** MyParserCallback();

ParserDelegator parser = **new** ParserDelegator();

URL url = **new** URL(urlString);

InputStreamReader reader = **new** InputStreamReader(url.openStream());

parser.parse(reader, callback, **true**);

**return** callback.content;

}

The **parse()** method of the **ParserDelegator** instance drives the callback methods to parse the web content retrieved from the input stream reader. The last parameter (**ignoreCharSet**) of the **parse()** method indicates whether it ignores the charset of not. If **ignoreCharSet** is equal to false, the parser may throw **ChangedCharSetException** when it parses a redirected page or the page without specified **CharSet**.

# Getting Unique Words from Text Content

For different purposes, we want a list of unique words from a text content. For example, we have the following paragraph (*java.txt*) that includes English words, numbers, and symbols.

Java is a general-purpose computer-programming language that is concurrent, class-based, object-oriented,[15] and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA),[16] meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.[17] Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. As of 2016, Java is one of the most popular programming languages in use,[18][19][20][21] particularly for client-server web applications, with a reported 9 million developers.[22] Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Now, we are going to retrieve the unique words (no duplicated words, numbers, and symbols) from the above paragraph. First, we need to split the paragraph into words. The easy way is using the **split()** method of the *String* object with a regular expression. As we want to keep words only, we can use the following to match all non-alphabet characters and use them as delimiters to split the paragraph.

*[\\d\\W]+*

Then, we filter out the duplicated words using an *ArrayList* with its **contains()** method.

**public** **static** List<String> getUniqueWords(String text) {

String[] words = text.split("[0-9\\W]+");

ArrayList<String> uniqueWords = **new** ArrayList<String>();

**for** (String w : words) {

w = w.toLowerCase();

**if** (!uniqueWords.contains(w))

uniqueWords.add(w);

}

**return** uniqueWords;

}

Optionally, you may sort the sort the words by using the **sort()** method of the List.

uniqueWords.sort(**new** Comparator<String>() {

**@Override**

**public** int compare(String a, String b) {

**return** a.compareTo(b);

}

});

# Getting Hyperlinks

<a href="*url*">*link text*</a>

We know that the *<a>* tag is used for defining a hyperlink in the HTML document. The **href** attribute specifies the destination address of the link.

Next, we override the **handleStartTag()** method of the MYParserCallback class for getting the hyperlinks in the target web page. The **handleStartTag()** method performs the following actions:

1. Check whether the current tag is an *<a>* tag or not.
2. If the current tag is an *<a>* tag, it obtains the attribute names from the Mutable Attribute Set **attrSet** using the **getAttributeNames()** method. The **getAttributeNames()** method returns an Enumeration object that stores a set of attribute names.
3. The while-loop walks through all elements of the Enumeration object by using **e.hasMoreElements()** and **e.nextElement()**.
4. If the value of the element (toString()) equals “href”, use the element to retrieve the attribute value (destination address of the *<a>* tag) through the **attrSet.getAttribute()** method.  
   *\*\*Note that we cannot use the “href” string as a key to get the attribute through the* ***attrSet.getAttribute()*** *method.*
5. If the list does not contain the URL, the URL will be added to the list.

...

class MyParserCallback extends ParserCallback {

**public** List<String> urls = **new** ArrayList<String>();

...

**@Override**

**public** void handleStartTag(Tag tag, MutableAttributeSet attrSet, int pos)

{

**if** (tag.toString().equals("a")) {

Enumeration e = attrSet.getAttributeNames();

**while** (e.hasMoreElements()) {

Object aname = e.nextElement();

**if** (aname.toString().equals("href")) {

String u = (String) attrSet.getAttribute(aname);

**if** (urls.size() < max && !urls.contains(u))

urls.add(u);

}

}

}

}

}

In the code above, you can see that we add the list (*List<String>*) to the **MyParserCallback** class. The list will be used by the **handleStartTag()** method for storing the URLs retrieved from the *<a>* tags.

## Using the parser

To obtain the resulted URLs, we simply access the list of the **MyParserCallback** object as the following function.

List<String> getURLs(String srcPage) **throws** IOException {

URL url = **new** URL(srcPage);

InputStreamReader reader = **new** InputStreamReader(url.openStream());

ParserDelegator parser = **new** ParserDelegator();

MyParserCallback callback = **new** MyParserCallback();

parser.parse(reader, callback, **true**);

**return** callback.urls;

}

For example, we have a web page at *http://localhost:8080/links.html* and its source code is as follows:

<**html**><**h1**>Some Links</**h1**><**ul**>

<**li**><**a** href="http://www.google.com">Google</**a**></**li**>

<**li**><**a** href="http://www.yahoo.com">Yahoo</**a**></**li**>

<**li**><**a** href="http://www.youtube.com">Youtube</**a**></**li**>

</**ul**></**html**>

The returned list of ***getURLs(“http://localhost:8080/links.html”)*** contains:

http://www.google.com

http://www.yahoo.com

http://www.youtube.com

## Handling Local Links

How about we change the source code of the web page changed as the follows:

<**html**><**h1**>Some Links</**h1**><**ul**>

<**li**><**a** href="http://www.google.com">Google</**a**></**li**>

<**li**><**a** href="http://www.yahoo.com">Yahoo</**a**></**li**>

<**li**><**a** href="http://www.youtube.com">Youtube</**a**></**li**>

<**li**><**a** href="/index.html">Index</**a**></**li**>

<**li**><**a** href="images/cat.jpg">Cat</**a**></**li**>

</**ul**></**html**>

We get a list that contains the following elements:

http://www.google.com

http://www.yahoo.com

http://www.youtube.com

*/index.html*

*images/cat.jpg*

The last two elements do not have the protocol and host name (with port). They are relative URLs retrieved from local links defined in the web page, but they do not work individually. We need to add the protocols and host names back so that they become absolute URLs.

***/index.html >>> http://localhost:8080/index.html***

***images/cat.jpg >>> http://localhost:8080/images/cat.jpg***

We modify the **getURLs()** function as follows:

List<String> getURLs(String srcPage) **throws** IOException {

URL url = **new** URL(srcPage);

InputStreamReader reader = **new** InputStreamReader(url.openStream());

ParserDelegator parser = **new** ParserDelegator();

MyParserCallback callback = **new** MyParserCallback();

parser.parse(reader, callback, **true**);

**for** (int i=0; i<callback.urls.size(); i++) {

String str = callback.urls.get(i);

**if** (!isAbsURL(str))

callback.urls.set(i, toAbsURL(str, url).toString());

}

**return** callback.urls;

}

Before returning the list, we scan all elements. If an element is a relative URL, the **toAbsURL()** function is invoked to reconstruct an absolute URL. The element then is replaced with the new URL.

The **isAbsURL()** function is used to check whether the inputted string is an absolute URL or not. A regular expression is used for the validation. You may use another more accurate regular expression to validate the URL instead.

boolean isAbsURL(String str) {

**return** str.matches("^[a-z0-9]+://.+");

}

The **toAbsURL()** function is used to reconstruct absolute URLs. The protocol, hostname, port number, and path for the reconstruction will be retrieved from the inputted URL **ref**.

URL toAbsURL(String str, URL ref) **throws** MalformedURLException {

URL url = **null**;

String prefix = ref.getProtocol() + "://" + ref.getHost();

**if** (ref.getPort() > -1)

prefix += ":" + ref.getPort();

**if** (!str.startsWith("/")) {

int len = ref.getPath().length() - ref.getFile().length();

String tmp = "/" + ref.getPath().substring(0, len) + "/";

prefix += tmp.replace("//", "/");

}

url = **new** URL(prefix + str);

**return** url;

}

Currently, the **toAbsURL()** function can reconstruct absolute URLs from relative URLs with or without forward slashes., such as the followings:

***/index.html >>> http://localhost:8080/index.html***

***images/cat.jpg >>> http://localhost:8080/images/cat.jpg***

But, if you want to handle the following cases, you need to modify the **toAbsURL()** function.

***Query:***

*?l=sc >>> http://localhost:8080/links.html?l=sc*

***fragment:***

*#label >>> http://localhost:8080/links.html#label*

**Building Java Web Server using Spring**

# Spring Boot

Spring Boot is a framework for creating stand-alone, production-grade Spring based Applications. You can find the detail information about Spring Boot in its official website at <https://spring.io/projects/spring-boot>. In this tutorial, we will use it to create a web application.

# Spring Project Creation and Configuration

The following is the procedure for creating a new Spring project:

1. Go to the Eclipse menu “File” > “New” > “Project…”.
2. Select “Spring Starter Project” under the “Spring Boot” folder. Then, click “Next”.
3. Fill the following information for the new Spring Start Project. Then, click “Next”.

|  |  |
| --- | --- |
| Field | Value |
| Name | SearchEngine |
| Group | hk.edu.hkbu.comp |
| Description | COMP4047 Project |
| Package | hk.edu.hkbu.comp |

1. On the “New Spring Starter Project Dependencies” page, click “Finish”.
2. The Eclipse takes couple ten seconds to create a new project. You may find the progress in the bottom right corner.

The following is the procedure for configuring the project for the web service.

1. Open the **pom.xml** file, switch to the “pom.xml” tab, and add the following lines to the dependencies:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

</dependency>

1. Change the web server port to a specific port number by adding the following line to the **src/main/resources/application.properties** file.

server.port=8080

1. Create a new class named **MyController** and add the following lines to the **MyController.java** file:

**package** **hk.edu.hkbu.comp**;

**import** **javax.servlet.http.HttpServletRequest**;

**import** **org.springframework.stereotype.Controller**;

**import** **org.springframework.web.bind.annotation.\***;

**import** **org.springframework.web.servlet.HandlerMapping**;

**@Controller**

**public** **class** **MyController** {

**@GetMapping**("\*\*")

**@GetMapping**("search")

**@ResponseBody**

String load(HttpServletRequest request) {

**return** String.format("You are browsing %s with %s!",

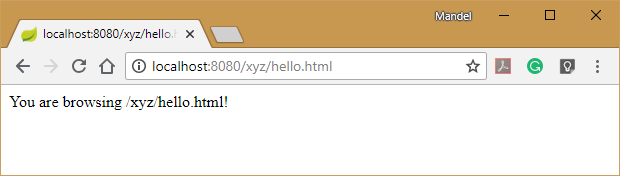
request.getRequestURI(), request.getQueryString());

}

}

1. Right-click your project folder and select “Run as” > “Spring Boot App”.
2. Then, go to the following URL:

http://localhost:8080/xyz/hello.html



1. Congratulation! Your first web service now is up and running.

The **load()** method of the **MyParserCallback** class performs the following tasks:

1. Create a file object using the request URI.
2. If the file object refers to a directory, create a new file object refers to the *index.html* file under the path.
3. Read the file content and store in the variable **response**.
4. If the file does not exist or there is any I/O exception, assign an error message to the variable **response** instead.
5. Return the variable **response**.

# Responding Specific Requests

We put *@GetMapping* annotation with a wildcard string (double asterisks, “\*\*”) at the beginning of the definition of the **load()** method. When the user types anything in the URI, the program will invoke the **load()** method then.

We can add more specific responds by using different values.

**@GetMapping**("greeting")

**@ResponseBody**

String sayHello(

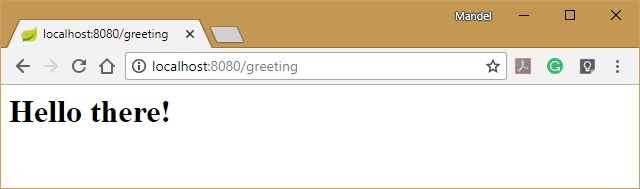
**@RequestParam**(name = "name", required = **false**, defaultValue = "there")

String name) {

**return** "<h1>Hello " + name + "!</h1>";

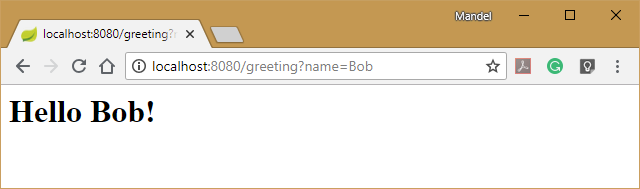
}

The example code above defines a new method **sayHello()** that returns a greeting. The *@GetMapping* annotation with the value “greeting” indicates that this method will be invoked if the user enters *“http://localhost:8080/greeting”* in the address bar of the web browser.



Default value

The *@RequestParam* annotation is used for representing a parameter defined in the query string. The value of the parameter in the query string will be stored in the variable following the annotation.



# Static Resources

The methods we define previously make our Spring Boot application responds with dynamic contents. For the static contents, we only need to put the files in the **src/main/resources/static/** directory under our project directory.

1. Let’s copy the following content to **src/main/resources/static/index.html**.

<html>

<head>

<title>My Search Engine</title>

</head>

<body>

<h1>My Search Engine</h1>

<form action="load" method="GET">

<p>Search: <input type="text" name="query" /> <input type="submit"/></p>

</form>

</form>

</body>

</html>

1. Next, you need to change the value of the *@GetMapping* of the **load()** method other than “\*\*”, say “load”. Otherwise, all static resource requests will be dropped into the **load()** method.

**@GetMapping**("load")

**@ResponseBody**

String load(HttpServletRequest request) { ... }

1. Then, right click your project folder in the Package Explorer of Eclipse and click “Refresh” to force refreshing the file linkages. After that you will see the HTML at <http://localhost:8080/>.



# References

1. ArrayList (Java Platform SE 8 ). (2018, July 7). Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
2. HTML Links. (n.d.). Retrieved from https://www.w3schools.com/html/html\_links.asp
3. InputStreamReader (Java Platform SE 8). (n.d.). Retrieved from https://docs.oracle.com/javase/8/docs/api/?java/io/InputStreamReader.html
4. Java Servlet Technology. (n.d.). Retrieved from https://www.oracle.com/technetwork/java/index-jsp-135475.html
5. Lesson: Working with URLs (The Java™ Tutorials > Custom Networking). (n.d.). Retrieved from https://docs.oracle.com/javase/tutorial/networking/urls/index.html
6. Regular Expressions and the Java Programming Language. (n.d.). Retrieved from https://www.oracle.com/technetwork/articles/java/regex-1564923.html
7. Spring Projects. (n.d.). Retrieved from https://spring.io/projects/spring-boot
8. Serving Web Content with Spring MVC. (n.d.). Retrieved from https://spring.io/guides/gs/serving-web-content/
9. String.split() (Java Platform SE 8 ). (2018, July 7). Retrieved from https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#split-java.lang.String-