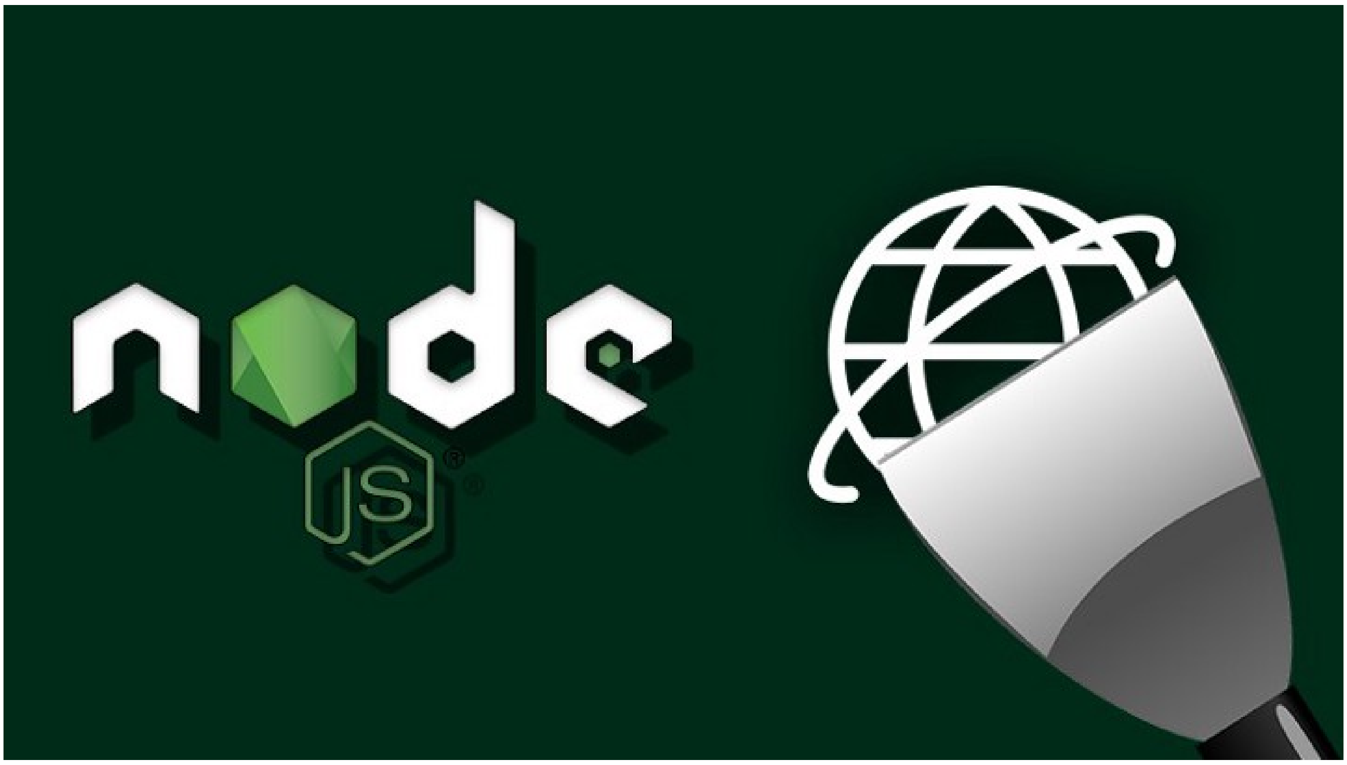
Stefan Hyltoft Follow
Jan 27, 2020 · 11 min read · Listen

Scraping HTML Tables with Nodejs Request and Cheerio

Learn how to build web scraper to scrape HTML tables with Nodejs, Request-promise and Cheerio!

Are you more of a video learning person? Check out my course on Udemy with a section showing how to do this!

Click here to check out my 10 hour Nodejs Web Scraping course on Udemy!

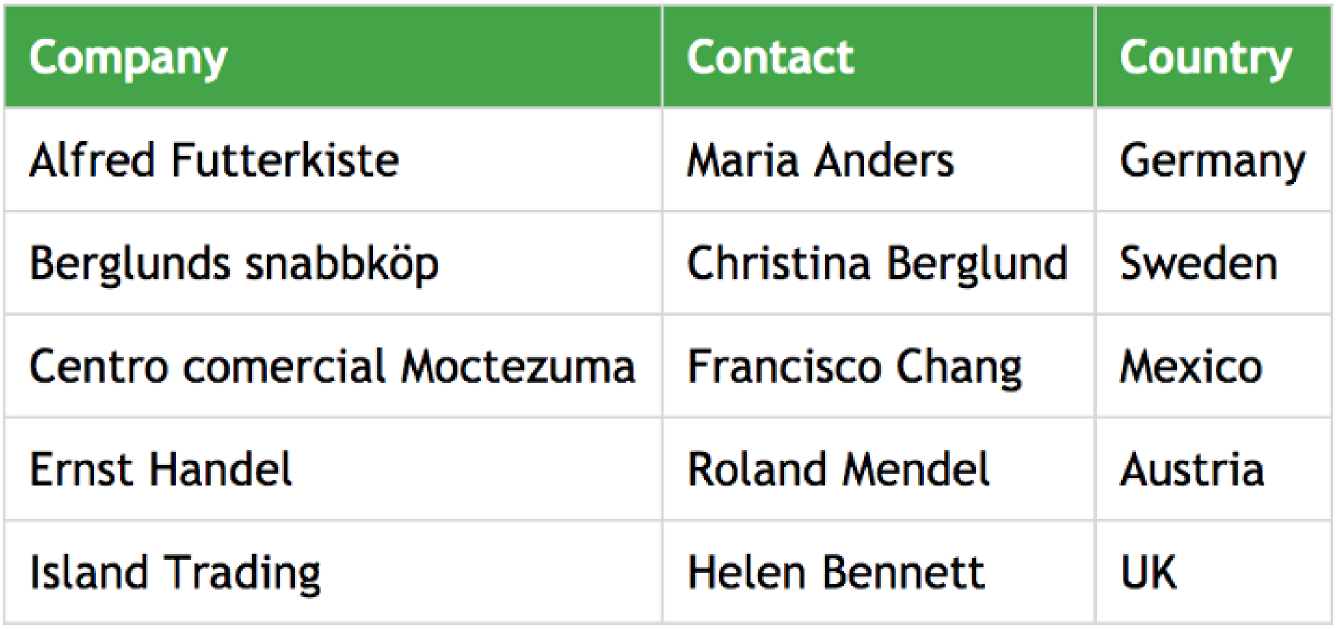


Check out my 10 hour Nodejs Web Scraping course on Udemy!

We are going to see an example of how to scrape data from a simple HTML table.

We are going to work from this simple example table I have here:

https://www.codingwithstefan.com/table-example/



Company	Contact	Country
Alfred Futterkiste	Maria Anders	Germany
Berglunds snabbköp	Christina Berglund	Sweden
Centro comercial Moctezuma	Francisco Chang	Mexico
Ernst Handel	Roland Mendel	Austria
Island Trading	Helen Bennett	UK

Understand HTML Table Structure

In the table above we can see *table headers*, saying Company, Contact and Country.

Then in the following rows we have the data of the table itself.

Now let’s take a look at the HTML code of the table.

```
<table>
    <tr>
      <th>Company</th>
      <th>Contact</th>
      <th>Country</th>
    </tr>
      <tr>
        <td>Alfred Futterkiste</td>
        <td>Maria Anders</td>
```

```
            <td>Sweden</td>
        </tr>
        <tr>
            <td>Centro comercial Moctezuma</td>
            <td>Francisco Chang</td>
            <td>Mexico</td>
        </tr>
        <tr>
            <td>Ernst Handel</td>
            <td>Roland Mendel</td>
            <td>Austria</td>
        </tr>
        <tr>
            <td>Island Trading</td>
            <td>Helen Bennett</td>
            <td>UK</td>
        </tr>
    </table>
```

We have the HTML *<table>* element, then we have *table row (<tr>)* elements inside it. In the first table row element, we have 3 <th> elements, known as *table headers*. Company, Contact, Country. These are the green/white row we see at the top of the table, the table header, or <th> element's.

Now on to the remaining <tr> or table rows, we can see some <td> elements, or table data elements. These contain the table's data itself.

So for each table row we have a <tr> element with a corresponding closing element, </tr>.

Structure of our scraped data

There's different ways we could arrange our scraped data from the table. I think the simplest and most readable one, would simply be an array of objects. Each object would have a property with a name that matches the table header, and a value inside from the table data row. So it would look like this in JavaScript:

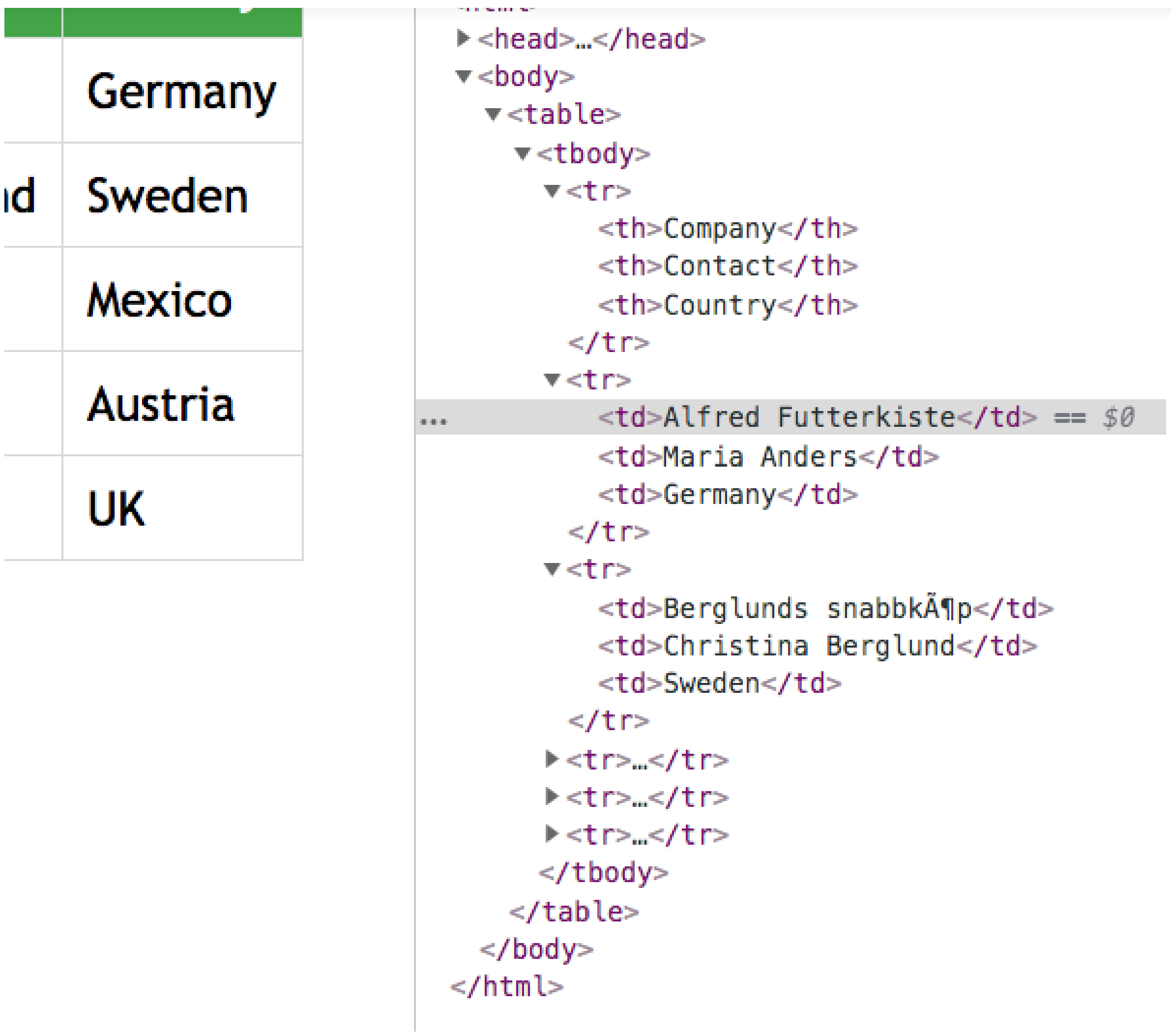
```
    Country: "Germany"
  },
  {
    Company: "Berglunds snabbköp",
    Contact: "Christina Berglund",
    Country: "Sweden"
  },
  {
    Company: "Centro comercial Moctezuma",
    Contact: "Francisco Chang",
    Country: "Mexico"
  },
  {
    Company: "Ernst Handel",
    Contact: "Roland Mendel",
    Country: "Austria"
  },
  {
    Company: "Island Trading",
    Contact: "Helen Bennett",
    Country: "UK"
  }
];
```

So here we have an array shown by the square brackets ([ ]), containing the objects, each object inside the curly brackets ({ }). Once we've scraped the table data from the HTML and converted it into a data structure like this, we can easily manipulate it, save it in a database, do statistics and so on.

In the next section, let's move on to creating a selector that we can use to select the data we need.

A look at selectors and jQuery injection

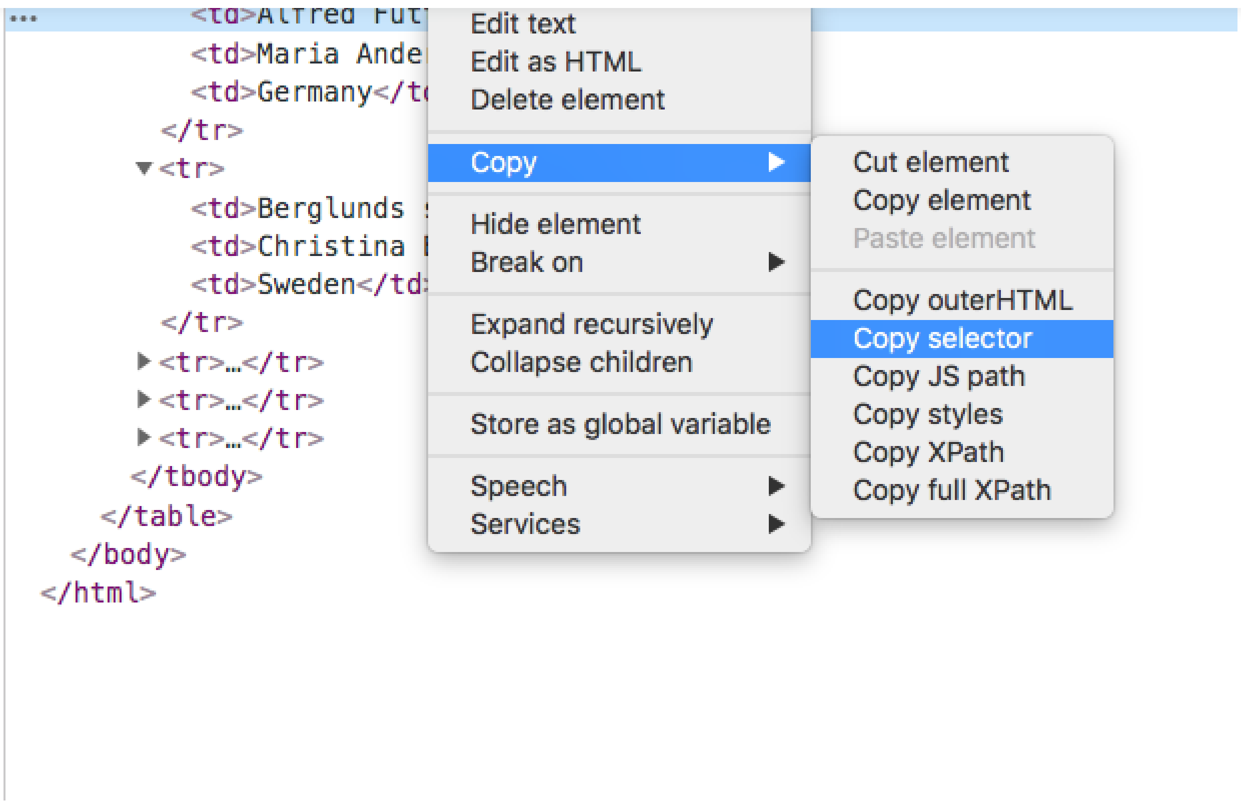
To select and get exactly the data we need for our project from the HTML table, we have to use selectors. Let's open up Chrome Tools with the table page. Press F12 to open Chrome Tools.



```
    ▶<head>…</head>
    ▼<body>
      ▼<table>
        ▼<tbody>
          ▼<tr>
              <th>Company</th>
              <th>Contact</th>
              <th>Country</th>
            </tr>
          ▼<tr>
...           <td>Alfred Futterkiste</td> == $0
              <td>Maria Anders</td>
              <td>Germany</td>
            </tr>
          ▼<tr>
              <td>Berglunds snabbkÃ¶p</td>
              <td>Christina Berglund</td>
              <td>Sweden</td>
            </tr>
          ▶<tr>…</tr>
          ▶<tr>…</tr>
          ▶<tr>…</tr>
          </tbody>
        </table>
      </body>
    </html>
```

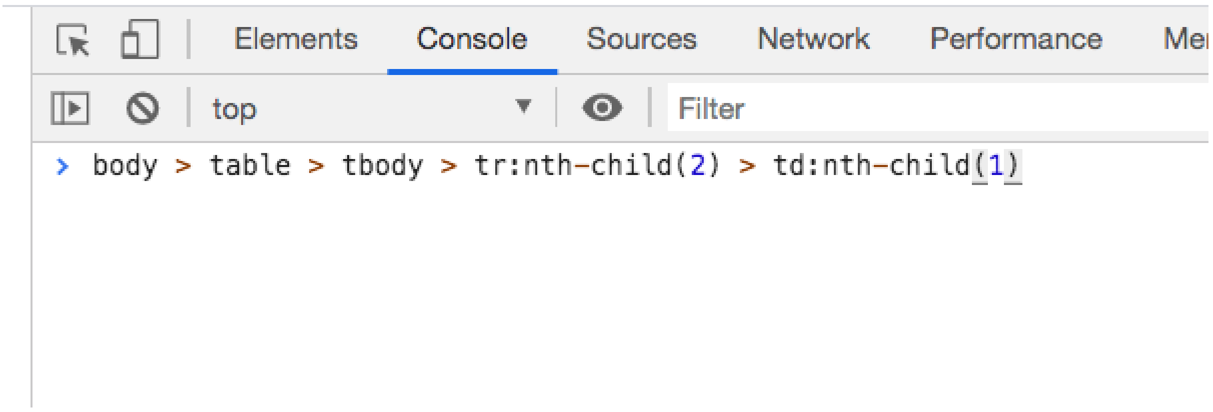
Use the Inspect Tool to select an element and see it's HTML code

Right click on "Alfred Futterkiste", and click on "Copy selector".


```
<td>Alfred Fut
<td>Maria Ande
<td>Germany</t
</tr>
▼<tr>
<td>Berglunds
<td>Christina
<td>Sweden</td
</tr>
▶<tr>…</tr>
▶<tr>…</tr>
▶<tr>…</tr>
</tbody>
</table>
</body>
</html>
```

Right click the element, and select “Copy selector”.

Then head on over to the Console tab, and paste your selector in using Ctrl+V or Cmd+V. Mine looks like this, yours should too:

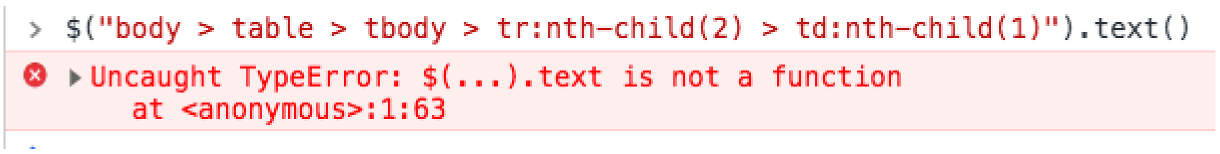


```
> body > table > tbody > tr:nth-child(2) > td:nth-child(1)
```

The selector generated by Chrome Tools for us

Let’s go over what this selector does. The arrows here is a *child selector*, meaning it will select the children of a certain element. So first we will select the table child element of the HTML body element. Then we will select the <tbody> child, then the <tr> child, then the td child.

$("body > table > tbody > tr:nth-child(2) > td:nth-child(1)").text();



```
> $("body > table > tbody > tr:nth-child(2) > td:nth-child(1)").text()
✖ ▸Uncaught TypeError: $(...).text is not a function
      at <anonymous>:1:63
```

Missing jQuery on a website can give this type of error

We can see it actually prints out that $(…).text is not a function. This is because the dollar sign is a shorthand for the jQuery library, and the site we are scraping doesn't contain jQuery.

We use jQuery to more easily select elements on the page we are scraping, although we could just use pure JavaScript. Another benefit is that our Cheerio package, that we use inside Nodejs has a very similar syntax to jQuery.

So let's inject jQuery into the page! I have a little Chrome Extension here that let's me easily inject jQuery into a page, if it doesn't have it.

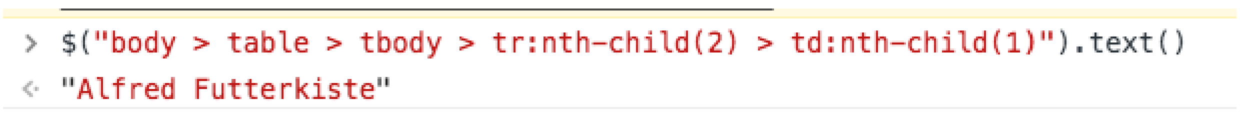
jQuery Inject

Injects jQuery in the page.

chrome.google.com

This way I can easily press the extension in Chrome, and we suddenly have jQuery injected.

Now let's try and run the .text() call again and see our output inside the console tab:



```
> $("body > table > tbody > tr:nth-child(2) > td:nth-child(1)").text()
< "Alfred Futterkiste"
```

Displaying the text content of the first cell in the first column of our table!

Hooray! We can see the data of one of our table cells!

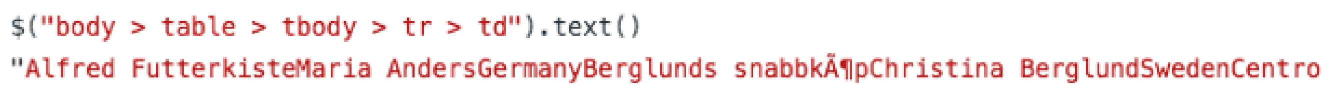
the first table data row, since the first row is the table header.

The td:nth-child(1) means we'll only select the first table data cell in this row. Hence, we get Alfred Futterkiste!

Now in the next section, let's see how we could change our selector to select all the data of our table.

Getting all the names in Chrome Tools

Now let's change our selector a bit, so we don't only get Alfred Futterkiste. It's pretty cool, but we need all the data!



```
$("body > table > tbody > tr > td").text()
"Alfred FutterkisteMaria AndersGermanyBerglunds snabbkÃ¶pChristina BerglundSwedenCentro
```

So now we remove the :nth-child selectors, so we get all the table rows, and all the data cells!

So far so good, but notice how there's no space between the names. It's hard to see which name is what. We need to have a list of names, not all the names in one long string. How can we do this?

We could use a *.each* loop, to go through all the rows, and print out the name, line by line. Let's see how that would look, try typing this out inside the console tab in Chrome Tools:

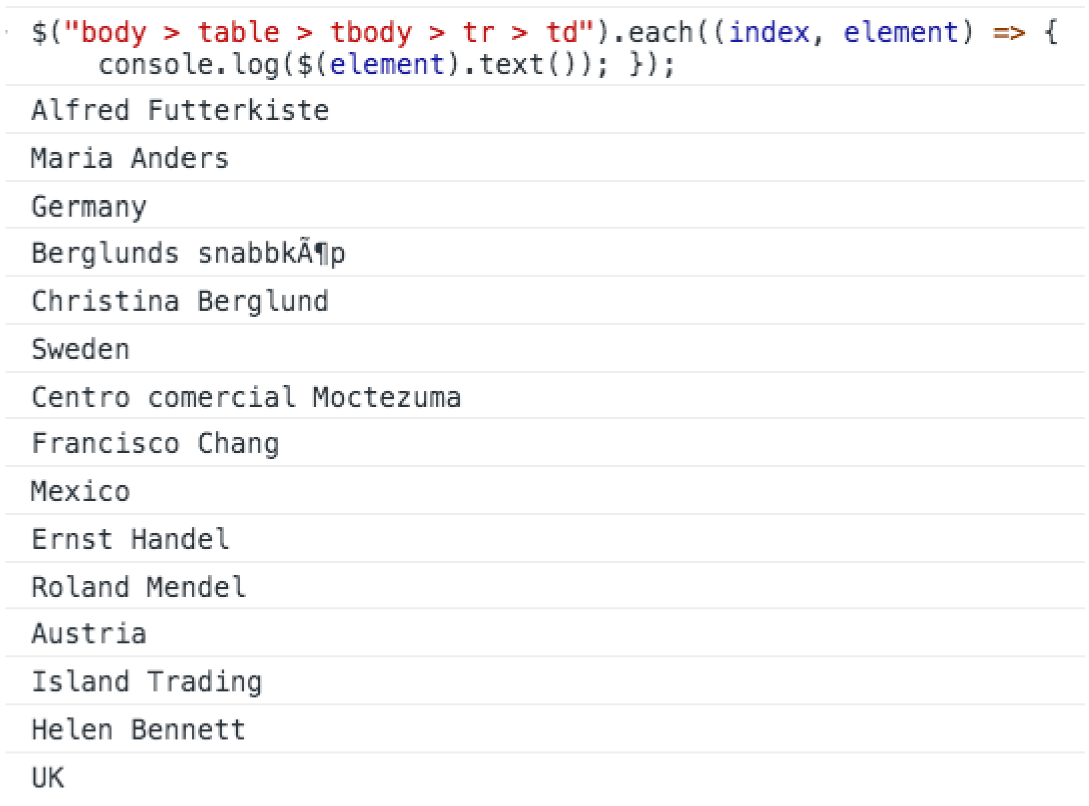
> *$("body > table > tbody > tr > td").each((index, element) => {*
>
> *console.log($(element).text());*
>
> *});*

Remember you can use Shift+Enter to make a line break inside Chrome Tools, to make it more readable to yourself as you're typing it in.

So what's going on here?

Then inside the loop, we do a console.log of this element's text contents. We use the dollar sign ($) jQuery selector to get the text content of each DOM element. It can be a little complicated if it's the first time you see this, but I hope you can understand it on closer inspection.

With this .each() loop, we are now getting all the table cell data!



```
$("body > table > tbody > tr > td").each((index, element) => {
    console.log($(element).text()); });
Alfred Futterkiste
Maria Anders
Germany
Berglunds snabbkÃ¶p
Christina Berglund
Sweden
Centro comercial Moctezuma
Francisco Chang
Mexico
Ernst Handel
Roland Mendel
Austria
Island Trading
Helen Bennett
UK
```

Printing out every cell in our table!

In the next section, let's take a look at how we could do this process inside Nodejs as well.

Getting all data inside Nodejs

So now we've created a selector for getting all the table data, and we've tested that it works well in Chrome Tools. Now it's time to use it inside Nodejs. This way we can scrape pages from a server running Nodejs, instead of only our Chrome browser.

Now open the directory in Visual Studio Code, and let's start importing some packages.

Inside the terminal write

```
npm init --yes
```

to initialize npm, so we can import the packages.

Then let's add the necessary packages.

```
yarn add cheerio request request-promise
```

If you don't have yarn, you can also use npm, but I like using yarn since it's faster.

```
npm i nstall request request-promise
```

Then make a new file called *index.js* inside the project folder.

Now let's start by importing the request-promise and cheerio package in the top of our index.js file:

```
const request = require("request-promise");

const cheerio = require("cheerio");
```

Next, we're going to make an async function that contains our code. The async keyword is so we can use the newest async/await features in JavaScript.

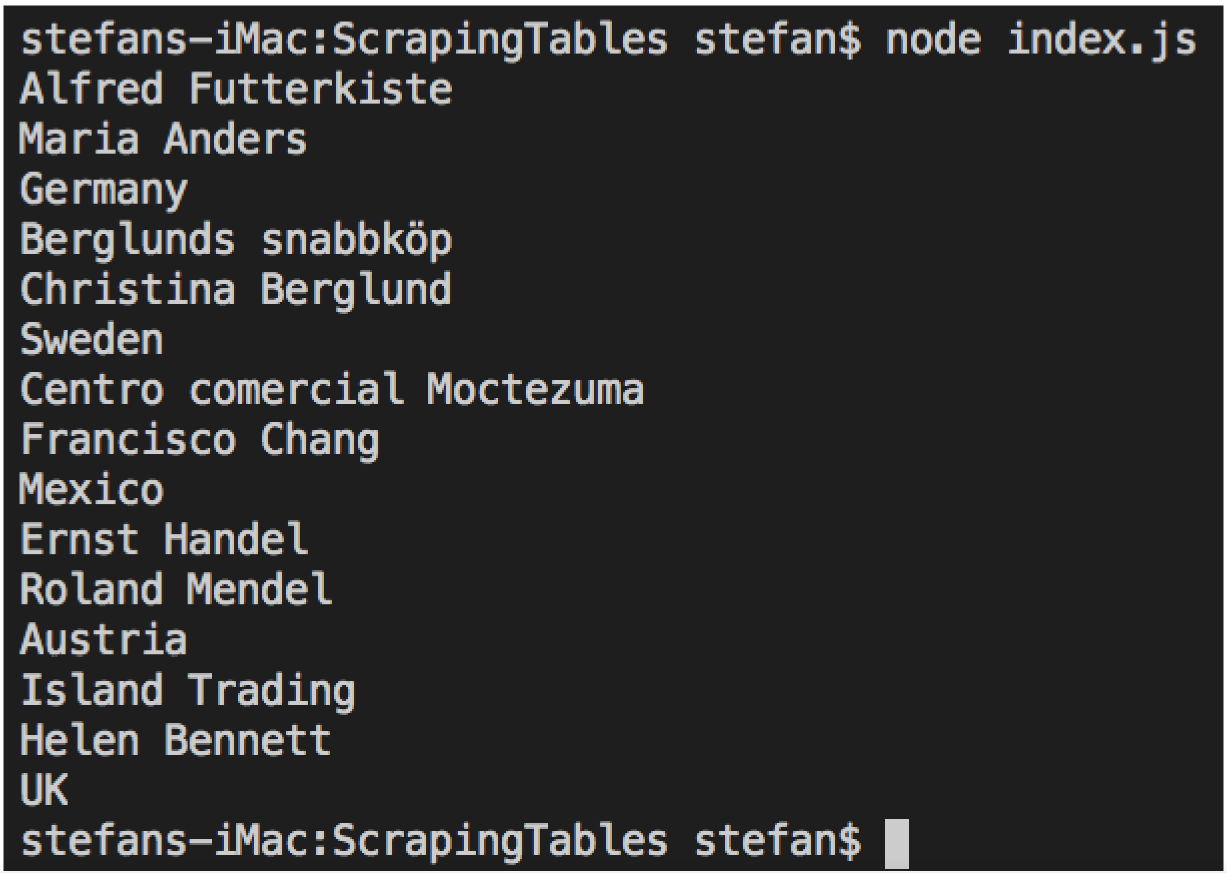
```
  const $ = cheerio.load(result);
  $("body > table > tbody > tr > td").each((index, element) => {
    console.log($(element).text());
  });
}

main();
```

test.js hosted with ❤ by GitHub view raw

We store the requested html page in the *result* variable. This html page is then passed onto cheerio.load, which returns a jQuery like object, which we call $ (dollar sign). This means we can write our code pretty much like when we were fiddling around in Chrome tools with jQuery.

We can simply paste the selector and loop we found to be working in Chrome tools with jQuery, into our Nodejs code.

Let's try to run the code in nodejs and see what happens in our console. Type *node index.js* inside your Visual Studio Code terminal and see the result!



```
stefans-iMac:ScrapingTables stefan$ node index.js
Alfred Futterkiste
Maria Anders
Germany
Berglunds snabbköp
Christina Berglund
Sweden
Centro comercial Moctezuma
Francisco Chang
Mexico
Ernst Handel
Roland Mendel
Austria
Island Trading
Helen Bennett
UK
stefans-iMac:ScrapingTables stefan$
```

Let's recap on what's going on in the node js code here:

```
const request = require("request-promise");
const cheerio = require("cheerio");

async function main() {
 const result = await request.get("http://codingwithstefan.com/table-example");
 const $ = cheerio.load(result);
 $("body > table > tbody > tr > td").each((index, element) => {
   console.log($(element).text());
 });
}

main();
```

test.js hosted with ❤ by GitHub view raw

Using the request-promise package we make an HTTP request to get the actual HTML page. We then feed this HTML page into cheerio. Cheerio returns an object that can be used just like the jQuery library on an actual webpage. We assign this to a dollar sign variable ($). This way, we can take the loop and the selector we created and tested in Chrome Tools, and use it inside Nodejs.

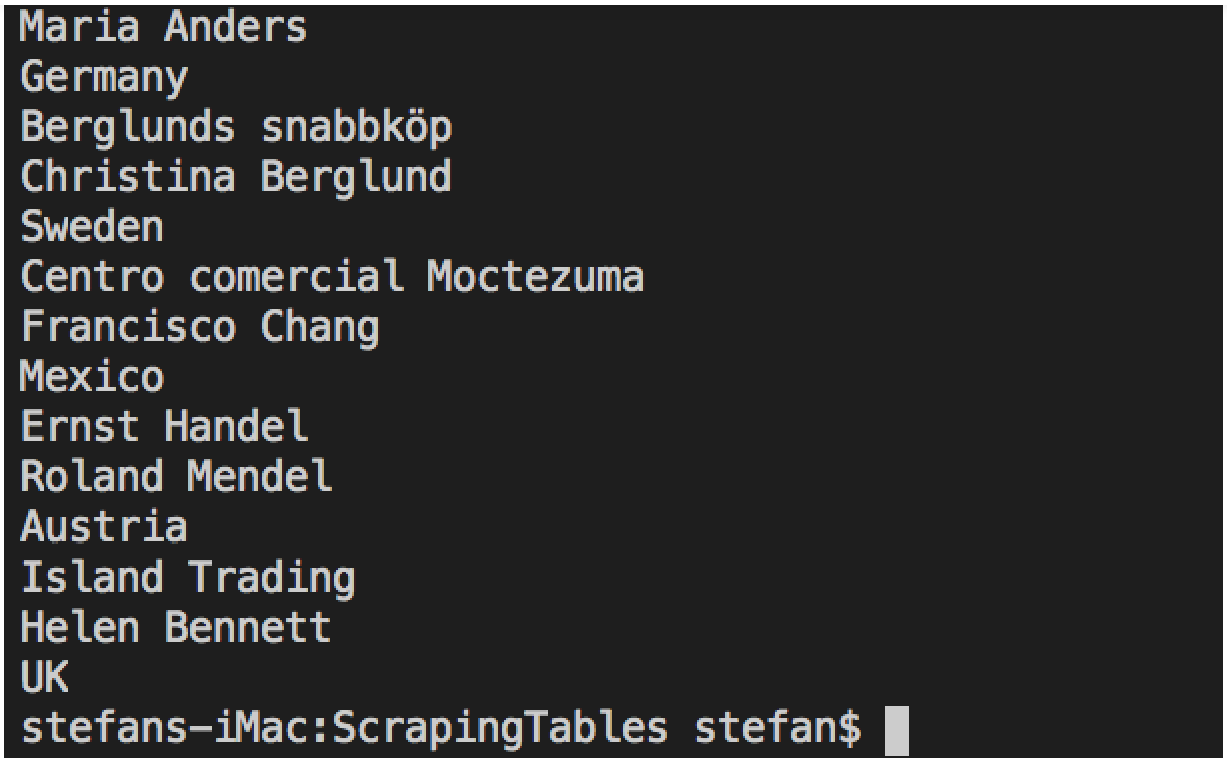
Now we can scrape the page without having a browser open, we could even build an API that returns the scraped data. We could also build an automatic periodic scraper that scrapes the table once every hour or day, or any other interval.

Okay, now let's move on to the next section, because we still don't have the scraped data into any data structure!

Getting only company names

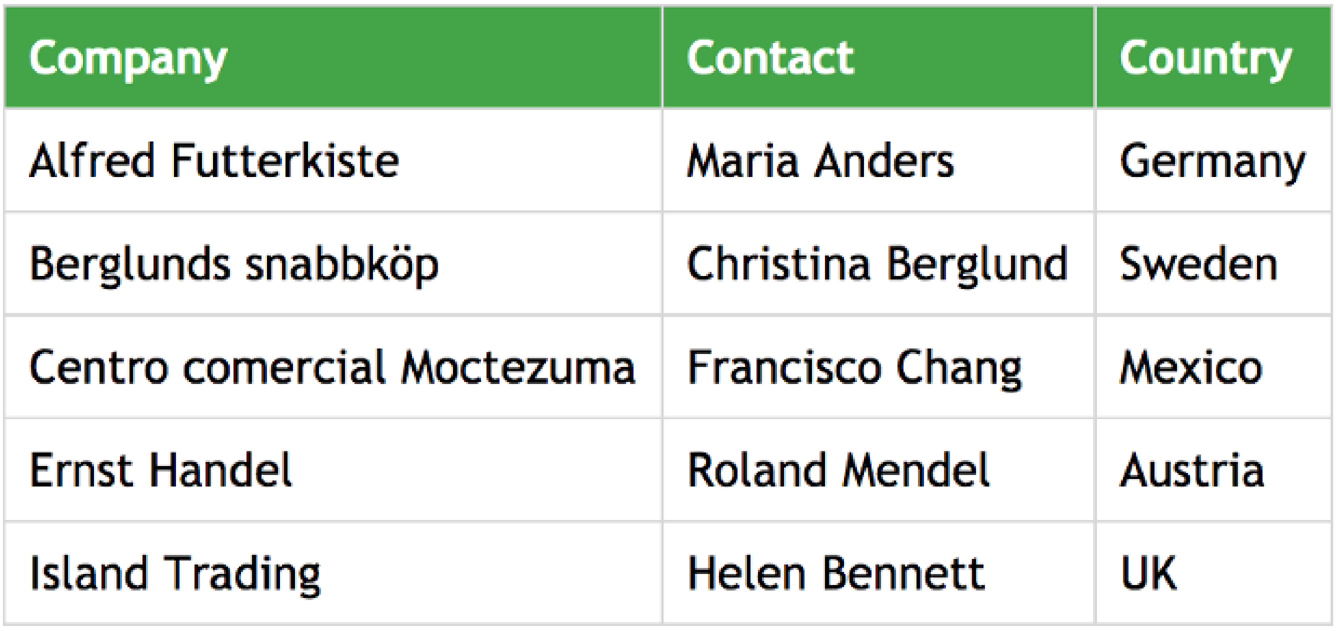
Now we got all the data out from the table, but how can we tell in the loop, which one is Company Name, which one is a Contact name?

If you look at our output in nodejs and the table, we are simply getting the table cells



```
Maria Anders
Germany
Berglunds snabbköp
Christina Berglund
Sweden
Centro comercial Moctezuma
Francisco Chang
Mexico
Ernst Handel
Roland Mendel
Austria
Island Trading
Helen Bennett
UK
stefans-iMac:ScrapingTables stefan$
```

The names are printed from top left, to lowest right according to the HTML table



Company	Contact	Country
Alfred Futterkiste	Maria Anders	Germany
Berglunds snabbköp	Christina Berglund	Sweden
Centro comercial Moctezuma	Francisco Chang	Mexico
Ernst Handel	Roland Mendel	Austria
Island Trading	Helen Bennett	UK

Names in corresponding HTML table

So how can we know in the code when we are at a company name, a contact name or a country name?

Well, we could do some funky switch statements based on the index use the first data cell is the company name, the second is contact, and the third is country.

Let’s try something like this:

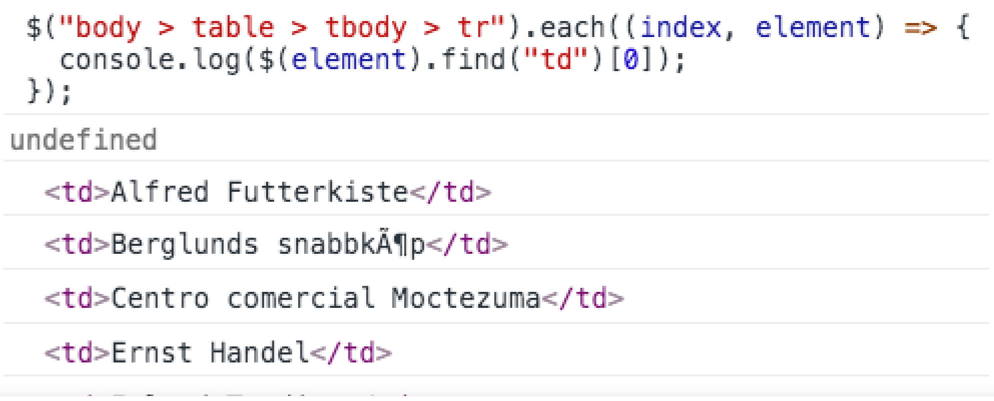
```
$(“body > table > tbody > tr”).each((index, element) => {

console.log($(element).find(“td”));

});
```

Now we removed the > td child selector, because we want to loop through the tr elements instead.

Then we use .find(“td”) to *find* all child elements of td inside this element. This is basically the same as the arrow > CSS child selector we use in our “body > table > tbody > tr” selector. This is going to return an array of three td elements for every table row. But now we can easily see that element number 0 (the first element), is the company name. The second td element is the contact name, and so on.

Let’s try a console.log on the first td element in each table row:

```
$(“body > table > tbody > tr”).each((index, element) => {

console.log($(element).find(“td”)[0]);

});
```



```
$("body > table > tbody > tr").each((index, element) => {
  console.log($(element).find("td")[0]);
});
undefined
  <td>Alfred Futterkiste</td>
  <td>Berglunds snabbkÃ¶p</td>
  <td>Centro comercial Moctezuma</td>
  <td>Ernst Handel</td>
```

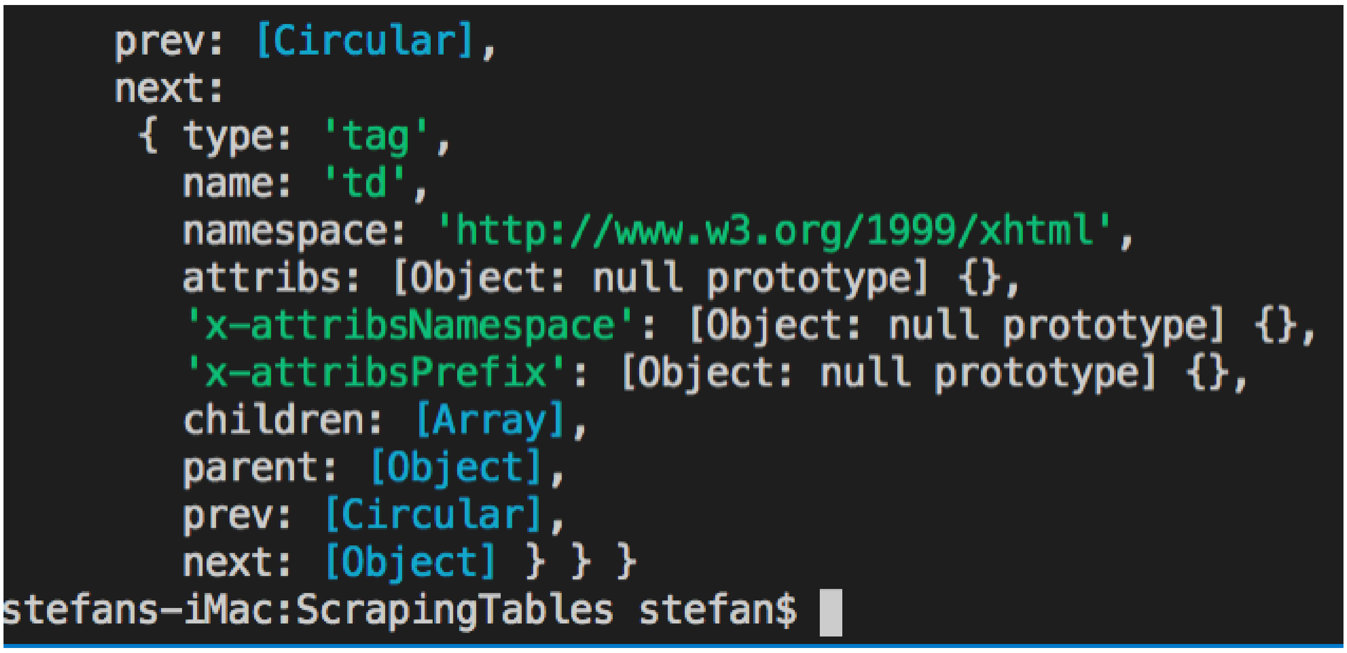
contains *<th>* elements, not *<td>* elements.

Let's try to do this inside Nodejs now:

```javascript
const request = require("request-promise");
const cheerio = require("cheerio");

async function main() {
 const result = await request.get("http://codingwithstefan.com/table-example");
 const $ = cheerio.load(result);
 $("body > table > tbody > tr").each((index, element) => {
   console.log($(element).find("td")[0]);
 });
}

main();
```

nowinsidenodejs.js hosted with ❤ by GitHub view raw

You might notice you get some pretty strange output in the nodejs terminal:

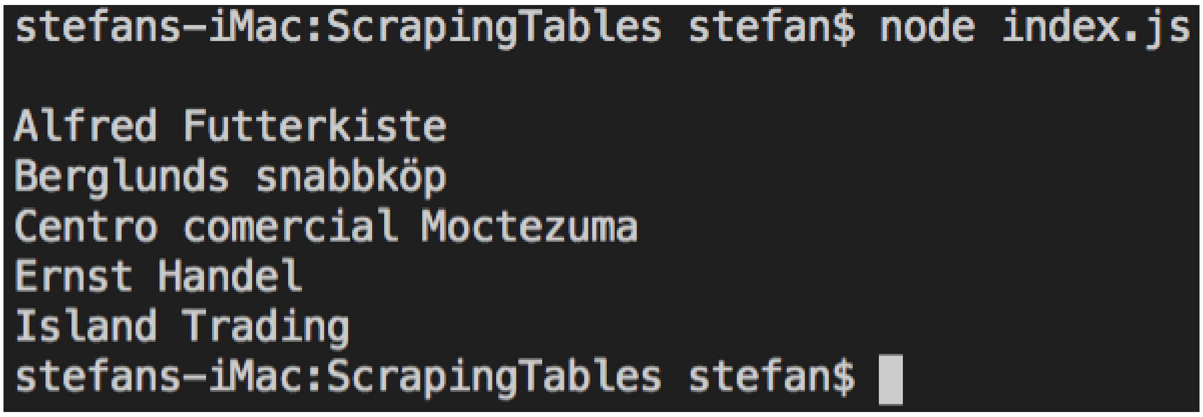


```
     prev: [Circular],
     next:
      { type: 'tag',
        name: 'td',
        namespace: 'http://www.w3.org/1999/xhtml',
        attribs: [Object: null prototype] {},
        'x-attribsNamespace': [Object: null prototype] {},
        'x-attribsPrefix': [Object: null prototype] {},
        children: [Array],
        parent: [Object],
        prev: [Circular],
        next: [Object] } } }
stefans-iMac:ScrapingTables stefan$ 
```

DOM elements printed out in Nodejs console

We're basically getting the jQuery elements as output, and not the HTML we saw in Chrome Tools. To get the actual text content of these DOM elements, we need to do

```
console.log($($(element).find("td")[0]).text());

});
```



```
stefans-iMac:ScrapingTables stefan$ node index.js

Alfred Futterkiste
Berglunds snabbköp
Centro comercial Moctezuma
Ernst Handel
Island Trading
stefans-iMac:ScrapingTables stefan$
```

with .text() the text content of DOM elements is printed out instead

Now this actually gives us all the text content company names, not just the DOM elements.

The code is getting a little weird now with the double dollar signs, what's going on here?

Well, .find("td") actually returns an array of DOM elements. To be able to get the text content of these DOM elements, we have to use the jQuery library again, hence the double $ signs.

But we can make this a little more readable like this:

```
$("body > table > tbody > tr").each((index, element) => {

const tds = $(element).find("td");

const company = $(tds[0]).text();

console.log(company);

});
```

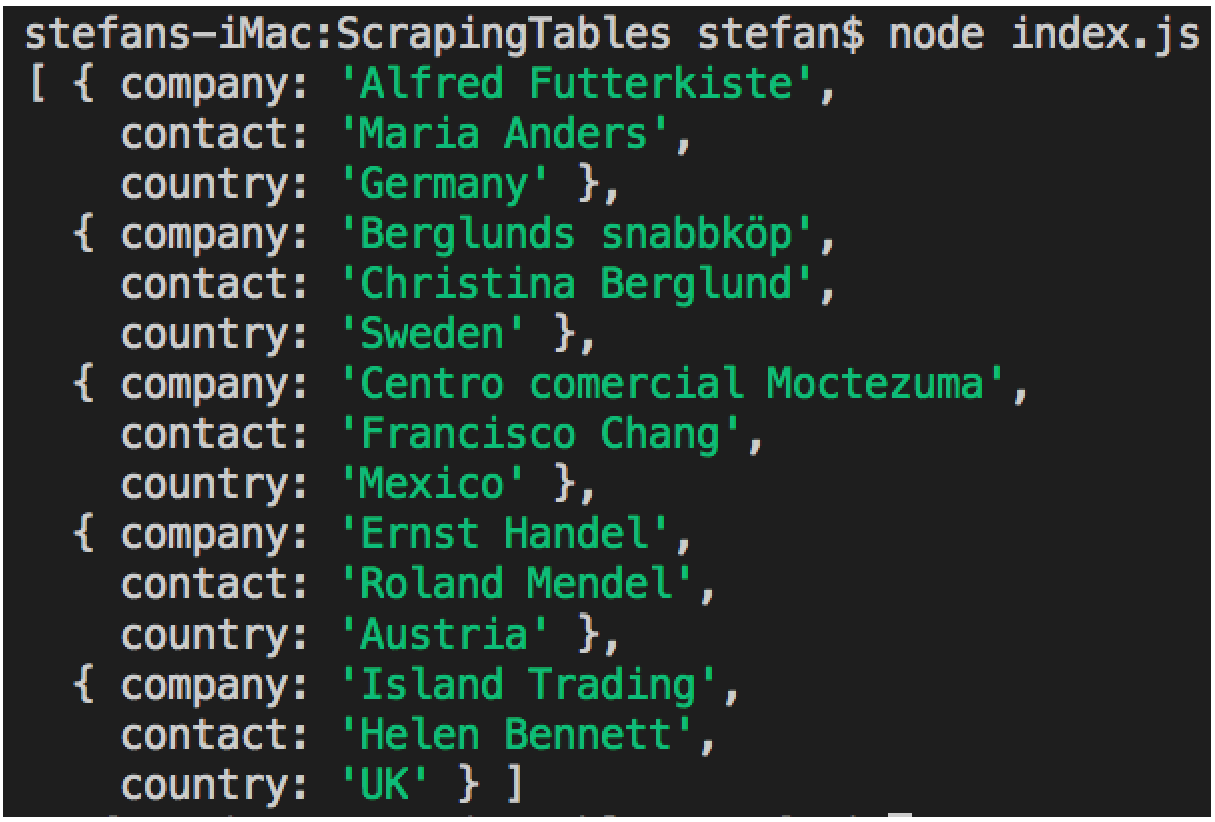
```
$(“body > table > tbody > tr”).each((index, element) => {

if (index === 0) return true;

const tds = $(element).find(“td”);

const company = $(tds[0]).text();

console.log(company);

});
```

Now getting the rest of the table headers and creating the data structure is pretty easy from here, let’s look at that in the next section!

Getting the rest of the columns

Now all we need to do is to take the rest of the table data fields in each table row and assign them to variables and our objects. Then initialize an empty array and push our objects with each table row data, and we’re done!

```
const request = require(“request-promise”);

const cheerio = require(“cheerio”);

async function main() {

    const result = await
request.get(“http://codingwithstefan.com/table-example");

    const $ = cheerio.load(result);

    const scrapedData = [];

    $(“body > table > tbody > tr”).each((index, element) => {

        if (index === 0) return true;
        const country = $(tds[2]).text();

        const tableRow = { company, contact, country };

        scrapedData.push(tableRow);

    });

    console.log(scrapedData);

}

main();
```



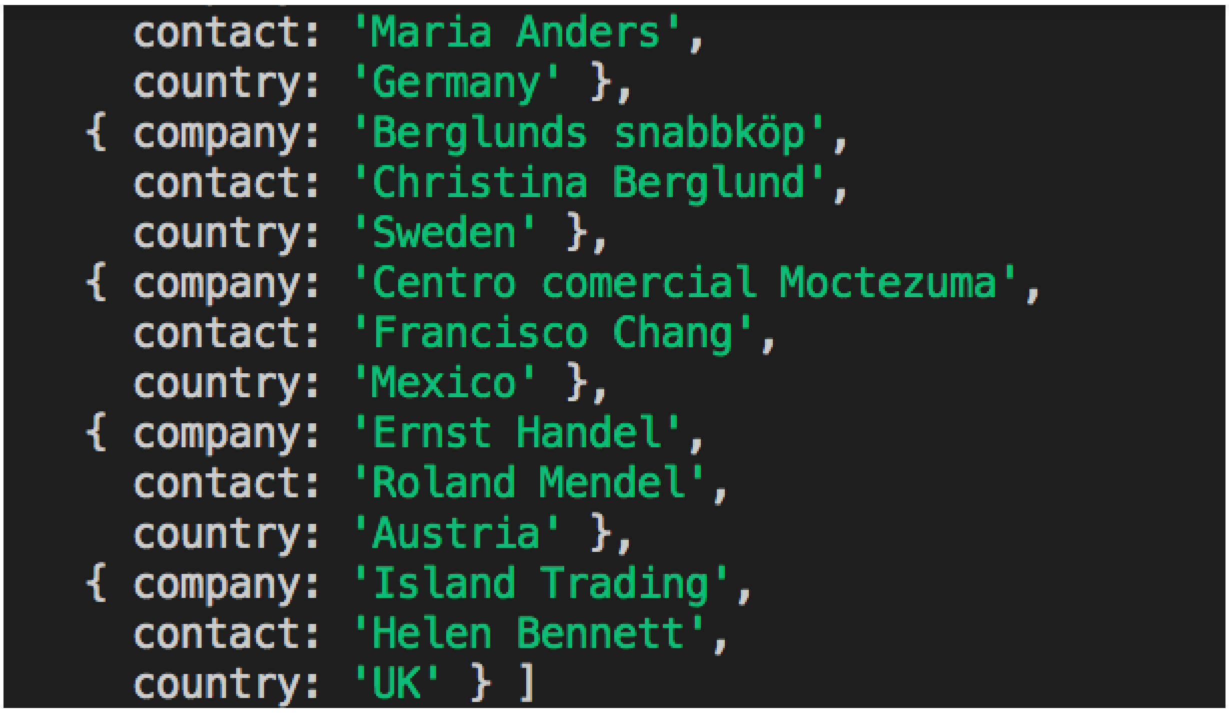
```
stefans-iMac:ScrapingTables stefan$ node index.js
[ { company: 'Alfred Futterkiste',
    contact: 'Maria Anders',
    country: 'Germany' },
  { company: 'Berglunds snabbköp',
    contact: 'Christina Berglund',
    country: 'Sweden' },
  { company: 'Centro comercial Moctezuma',
    contact: 'Francisco Chang',
    country: 'Mexico' },
  { company: 'Ernst Handel',
    contact: 'Roland Mendel',
    country: 'Austria' },
  { company: 'Island Trading',
    contact: 'Helen Bennett',
    country: 'UK' } ]
```

How cool is that! Now we got our HTML data in a format that we can use for anything inside Nodejs!

BONUS — Scraping Tables Dynamically

If we wanted our code to be more dynamic and use the table's header names automatically for the object property names, we could write something like this:

428 | 4



```
    contact: 'Maria Anders',
    country: 'Germany' },
  { company: 'Berglunds snabbköp',
    contact: 'Christina Berglund',
    country: 'Sweden' },
  { company: 'Centro comercial Moctezuma',
    contact: 'Francisco Chang',
    country: 'Mexico' },
  { company: 'Ernst Handel',
    contact: 'Roland Mendel',
    country: 'Austria' },
  { company: 'Island Trading',
    contact: 'Helen Bennett',
    country: 'UK' } ]
```

In this case if we are on the first table row, which is the table headers, we push all the table headers names into an array. Then we use each tableHeader as a variable name for the properties inside the object.

You might argue that the other code with hard coded column names is more verbose and easier to read, but if you have a lot of different, but similar structured tables, making something dynamic like this can save you a lot of typing (and debugging in the end). Or if the table have a lot of columns.

I would probably still recommend to use the "simpler" and hardcoded version if you are only scraping a table or two with a normal amount of columns.

Thank you for reading the article, I hope you enjoyed it!

Again, if you'd like to see a video version of this article or like to see more Nodejs Web Scraping instructions, check out my 10 hourNodejs Web Scraping Course on Udemy!

-Stefan