We're not in Ruby-land anymore...

Debugging the client-side with DevTools

Goals

By the end of this lesson, you will:

- Understand how and when to use specific debugging tools
- Be able to debug errors in HTML, CSS and JavaScript
- Know where to look for and how to isolate JavaScript errors

Debugging the Front-End with DevTools

Debugging your front-end code can be an intimidating part of the development process, but it's also one of the most powerful skills you can acquire. Developers of all levels spend a significant amount of time troubleshooting code, but the more comfortable you are with debugging tools, the easier it will be to isolate, identify and fix broken code.

The front-end languages (HTML, CSS and JavaScript) are run entirely in the browser, so the technique for troubleshooting broken code can happen in many places. Luckily, modern browsers are aware of this and give us a collection of advanced tools to help us debug.

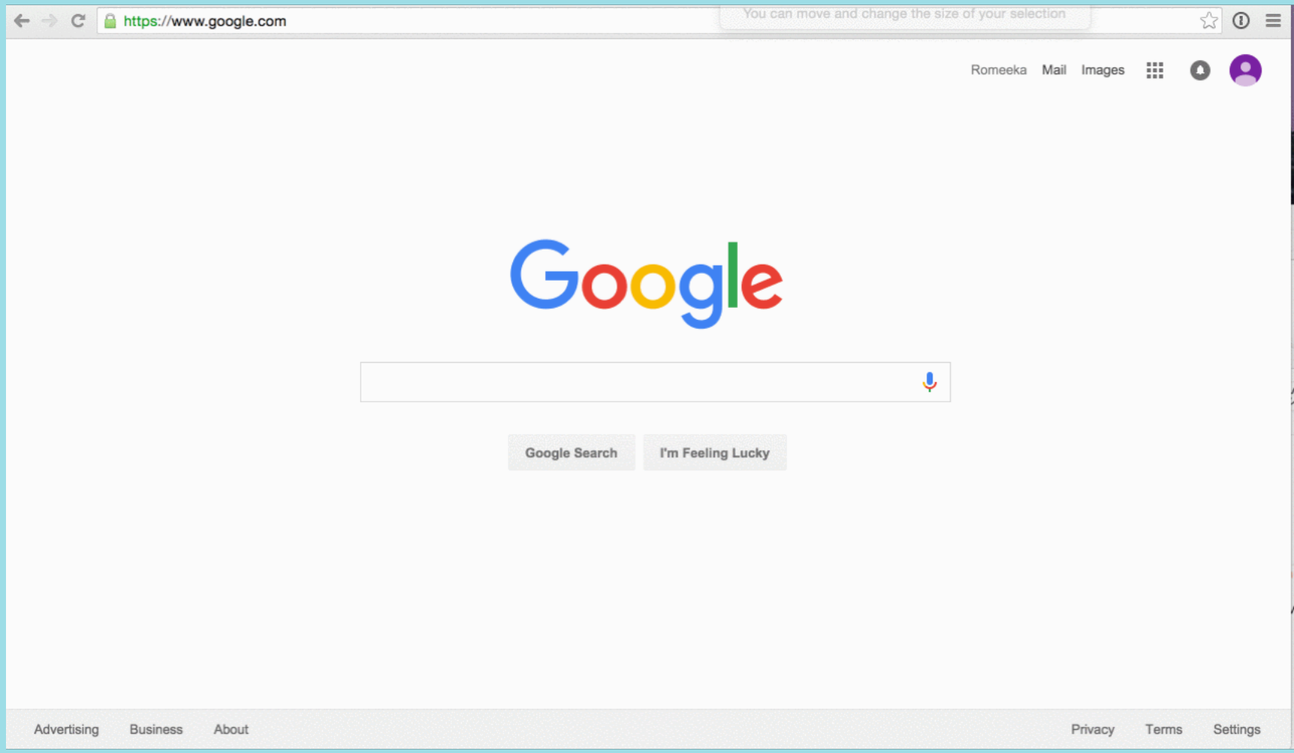
Developer Tools

One of the first tools you should familiarize yourself with when doing front-end development are the built-in browser DevTools. Though you can explore the browser DevTools on any webpage, we'll be using a simple expense tracking application to demonstrate how to use the DevTools. You can clone the application here, and follow the installation instructions in the README to get set up. The application has some pre-made bugs for us to solve as we go through the lesson.

Accessing DevTools

To open developer tools in Chrome:

- Mac: `Cmd + Opt + i`
- (or) Right click on the browser window and select `inspect`
- (or) Select `View` in the navbar, then `Developer`, then `Developer Tools`



The Panels

Once you have your DevTools window open, you'll notice a toolbar across the top with several different tabs. Clicking on these tabs will bring you to different debugging panels, each built to help troubleshoot specific areas of your front-end code.

As mentioned earlier, there are a lot of places on the front-end where code can go wrong. This means the first and most important step in solving a bug is **isolating the problem**. DevTools has already done some of this step for us by categorizing the most commmon areas where developers run into problems, and providing us with specific debugging panels for each.

For now, we're only going to focus on the first four panels: Elements, Console, Sources and Network. These are the primary panels you'll be working with most often.

The Elements Panel: Debugging HTML, CSS & DOM Events

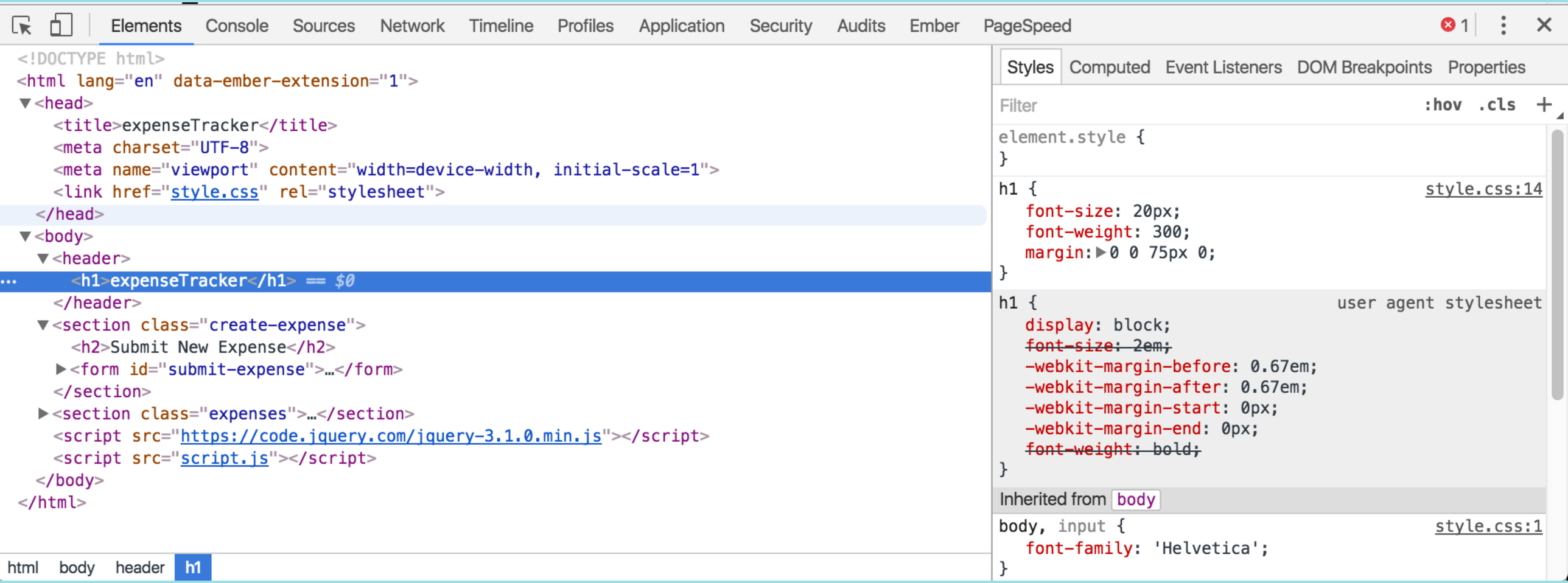
HELPFUL FOR:

- debugging layout and styling issues
- checking DOM events

The elements panel lets you view the entire HTML source of the current page you are viewing.

From here, you can edit, add or remove content and elements directly on the page.

Though your changes won't be saved (any changes made here will be lost upon refreshing the page), sometimes it's helpful to make tweaks directly in this panel so you can see what effect the changes will have on your application before you implement them.

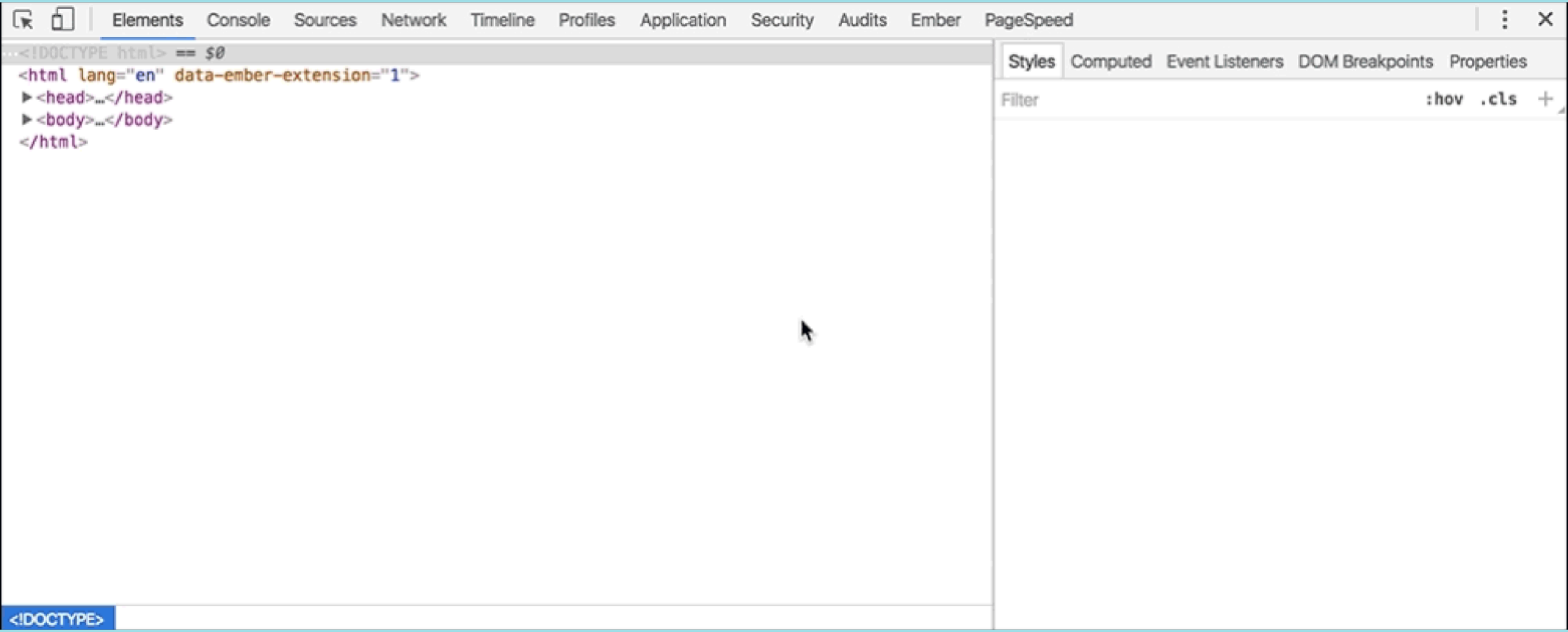

Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
1
<!DOCTYPE html>
<html lang="en" data-ember-extension="1">
<head>
<title>expenseTracker</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="style.css" rel="stylesheet">
</head>
<body>
<header>
<h1>expenseTracker</h1> == $0
</header>
<section class="create-expense">
<h2>Submit New Expense</h2>
<form id="submit-expense">…</form>
</section>
<section class="expenses">…</section>
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="script.js"></script>
</body>
</html>
html
body
header
h1
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov
.cls
element.style {
}
h1 {
style.css:14
font-size: 20px;
font-weight: 300;
margin: 0 0 75px 0;
}
h1 {
user agent stylesheet
display: block;
font-size: 2em;
-webkit-margin-before: 0.67em;
-webkit-margin-after: 0.67em;
-webkit-margin-start: 0px;
-webkit-margin-end: 0px;
font-weight: bold;
}
Inherited from
body
body, input {
style.css:1
font-family: 'Helvetica';
}

Selecting Elements to work with

You'll notice hovering over an HTML element in the devtools panel will also highlight that element on the page. This makes it easier to find and select the content you'd like to work with.

You can also select elements directly on the page by clicking the  icon in the toolbar, then hovering over the element on the page. This will automatically bring you to the corresponding code for that element in the devtools panel.

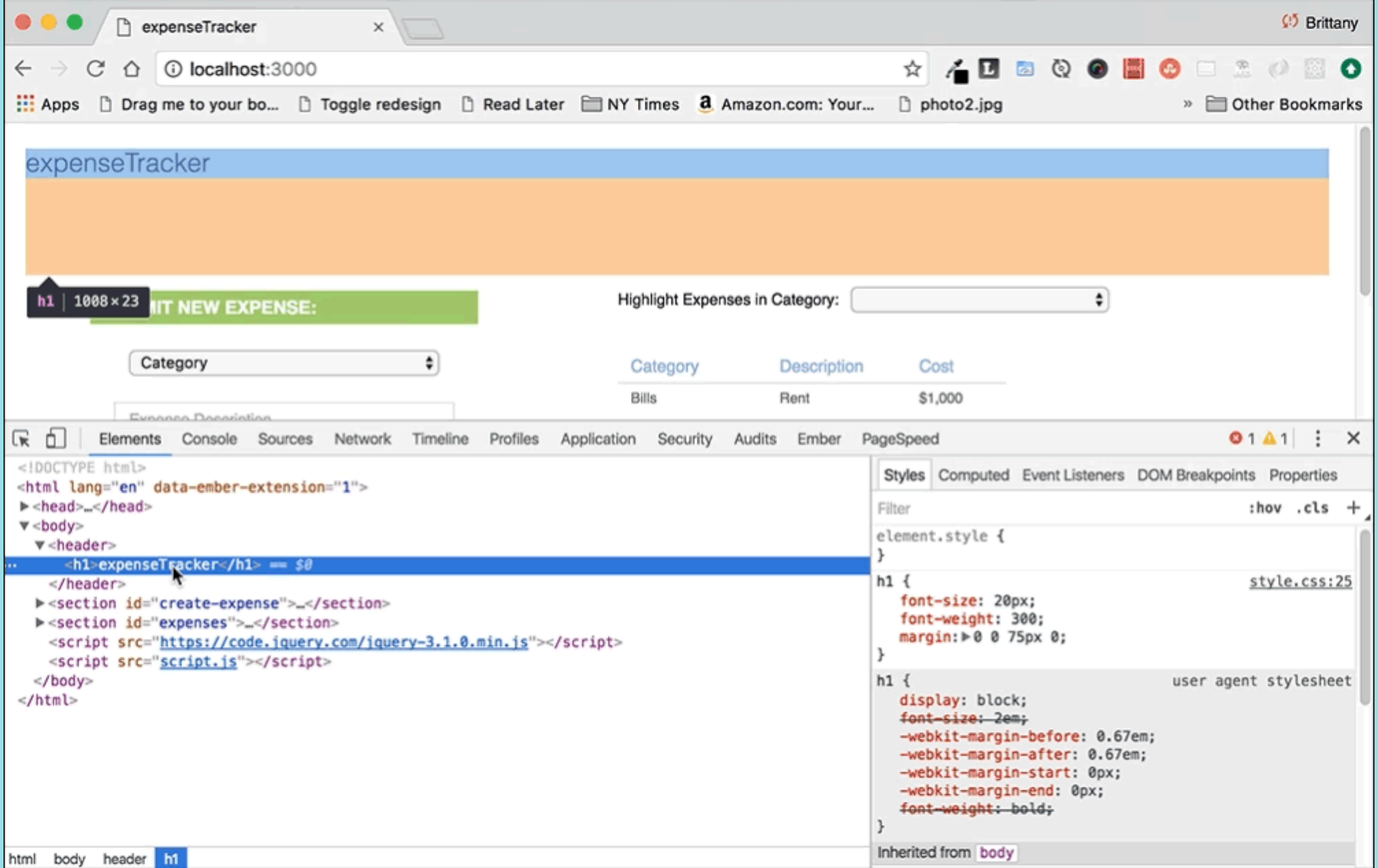
If you're having trouble finding the element you'd like to work with, you can search through the entire HTML with `Cmd + F`. You'll notice a searchbar appear at the bottom of the panel where you can enter any string to find a match. This is useful if you'd like to search for an element by a known ID or class.


Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
<!DOCTYPE html> == $0
<html lang="en" data-ember-extension="1">
<head>…</head>
<body>…</body>
</html>
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov
.cls
<!DOCTYPE>

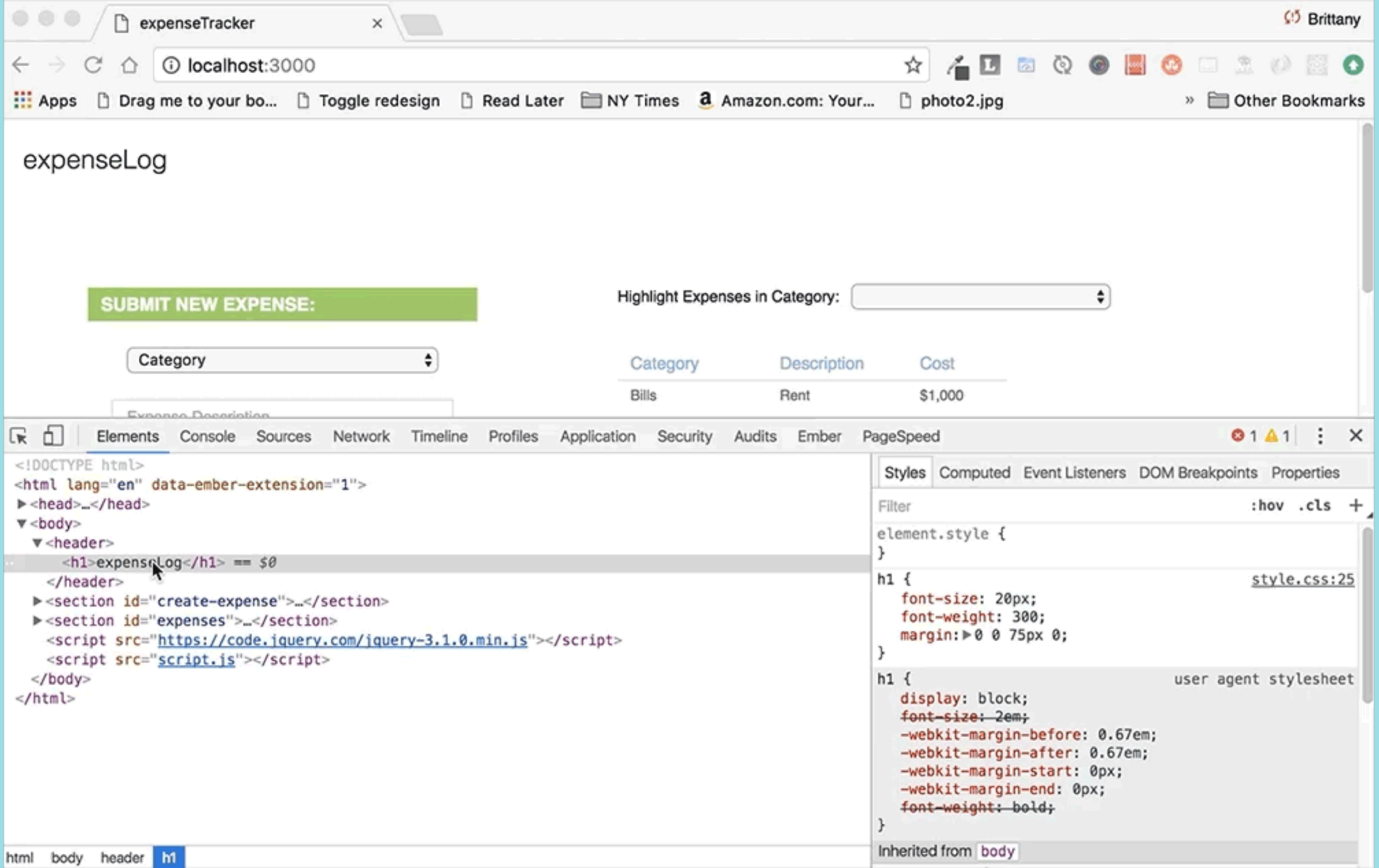
Editing Elements and Content

Directly from the elements panel, we can edit the HTML and see the changes reflected immediately. (Again, these changes won't be saved to your codebase, but sometimes it helps to see the tweaks as you make them before committing to them.)

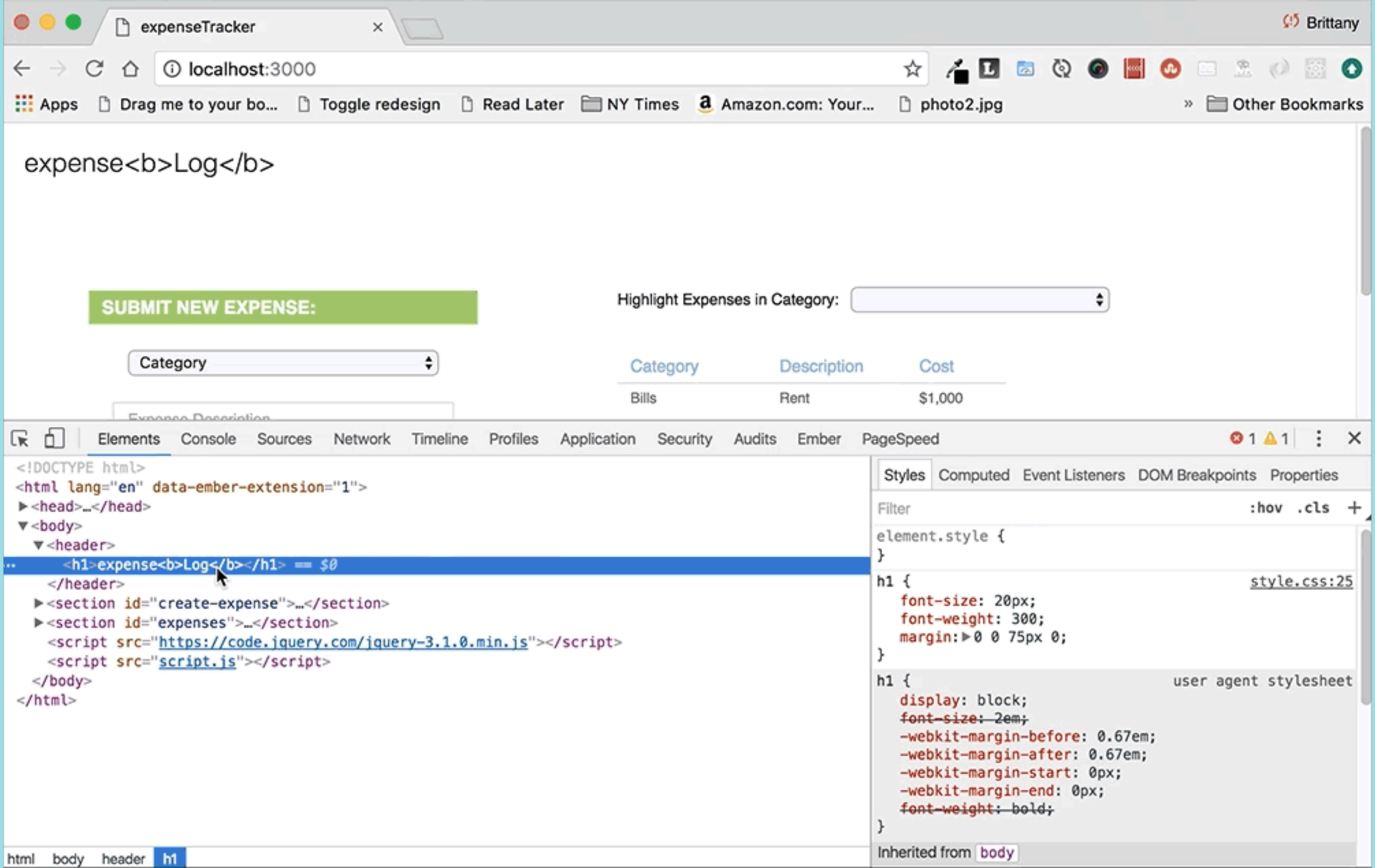
Let's say we want to edit the title of the application from 'expenseTracker' to 'expenseLog'. We can find that heading in the elements panel, double-click on it, and edit the text:


expenseTracker
Brittany
localhost:3000
Apps
Drag me to your bo...
Toggle redesign
Read Later
NY Times
Amazon.com: Your...
photo2.jpg
Other Bookmarks
expenseTracker
h1 | 1008×23
IT NEW EXPENSE:
Highlight Expenses in Category:
Category
Category
Description
Cost
Bills
Rent
$1,000
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
<!DOCTYPE html>
<html lang="en" data-ember-extension="1">
<head>…</head>
<body>
<header>
<h1>expenseTracker</h1> == $0
</header>
<section id="create-expense">…</section>
<section id="expenses">…</section>
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="script.js"></script>
</body>
</html>
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov .cls
element.style {
}
h1 {
style.css:25
font-size: 20px;
font-weight: 300;
margin: 0 0 75px 0;
}
h1 {
user agent stylesheet
display: block;
font-size: 2em;
-webkit-margin-before: 0.67em;
-webkit-margin-after: 0.67em;
-webkit-margin-start: 0px;
-webkit-margin-end: 0px;
font-weight: bold;
}
Inherited from body
html
body
header
h1

Now let's say we want to make just the word 'Log' in the title bold. If we wrap a `<b>` tag around 'Log', and hit the `enter` key to save the change, look what happens:


expenseTracker
Brittany
localhost:3000
Apps
Drag me to your bo...
Toggle redesign
Read Later
NY Times
Amazon.com: Your...
photo2.jpg
Other Bookmarks
expenseLog
SUBMIT NEW EXPENSE:
Category
Highlight Expenses in Category:
Category
Description
Cost
Bills
Rent
$1,000
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
<!DOCTYPE html>
<html lang="en" data-ember-extension="1">
<head>…</head>
<body>
<header>
<h1>expenseLog</h1> == $0
</header>
<section id="create-expense">…</section>
<section id="expenses">…</section>
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="script.js"></script>
</body>
</html>
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov .cls
element.style {
}
h1 {
style.css:25
font-size: 20px;
font-weight: 300;
margin: 0 0 75px 0;
}
h1 {
user agent stylesheet
display: block;
font-size: 2em;
-webkit-margin-before: 0.67em;
-webkit-margin-after: 0.67em;
-webkit-margin-start: 0px;
-webkit-margin-end: 0px;
font-weight: bold;
}
Inherited from body
html
body
header
h1

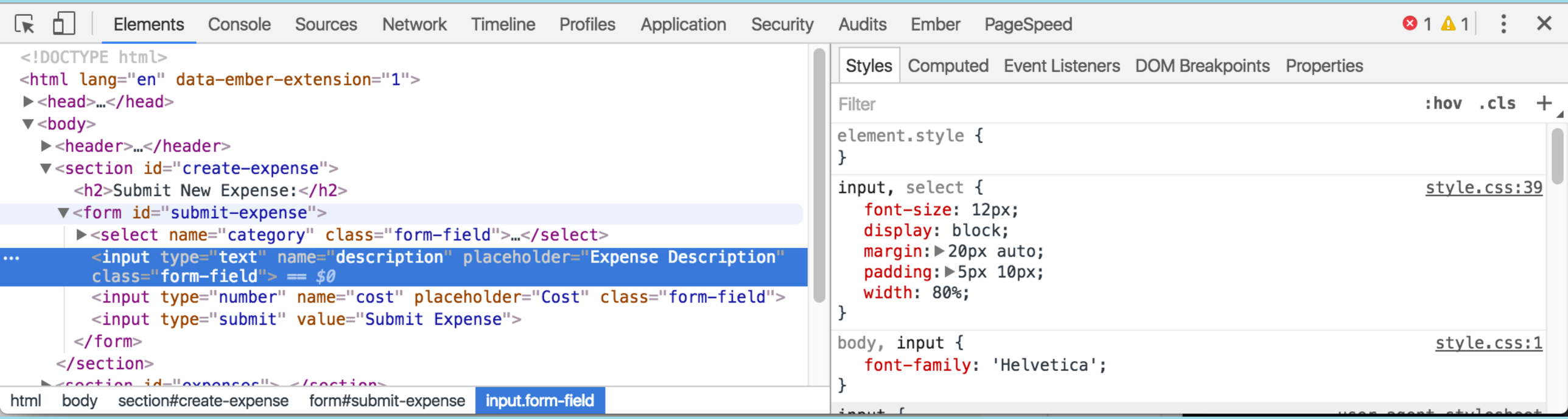
The browser didn't recognize that we wanted the <b> tag to be considered HTML, and it rendered it as plain text. In order to edit in HTML-mode, we must right-click on the element and select 'Edit as HTML'. Now we can wrap the word in <b> tags and it will render it bold:


expenseTracker
Brittany
localhost:3000
Apps
Drag me to your bo...
Toggle redesign
Read Later
NY Times
Amazon.com: Your...
photo2.jpg
Other Bookmarks
expenseLog
SUBMIT NEW EXPENSE:
Highlight Expenses in Category:
Category
Category
Description
Cost
Bills
Rent
$1,000
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
<!DOCTYPE html>
<html lang="en" data-ember-extension="1">
<head>…</head>
<body>
<header>
<h1>expenseLog</h1> == $0
</header>
<section id="create-expense">…</section>
<section id="expenses">…</section>
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="script.js"></script>
</body>
</html>
html
body
header
h1
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov .cls
element.style {
}
h1 {
style.css:25
font-size: 20px;
font-weight: 300;
margin:0 0 75px 0;
}
h1 {
user agent stylesheet
display: block;
font-size: 2em;
-webkit-margin-before: 0.67em;
-webkit-margin-after: 0.67em;
-webkit-margin-start: 0px;
-webkit-margin-end: 0px;
font-weight: bold;
}
Inherited from body

There are a lot of other options in the menu that appears when you right-click on an element. Play around with each of the options to see what else can be done.

Editing CSS

To the right of the HTML pane, there's a small sidebar that gives us styling information for the currently selected element. Similar to the HTML pane, we can add or remove styles and adjust CSS property values from this pane. You can click on any style property associated with the selected element and change its value. You can also use the blue checkbox to toggle the style on or off.

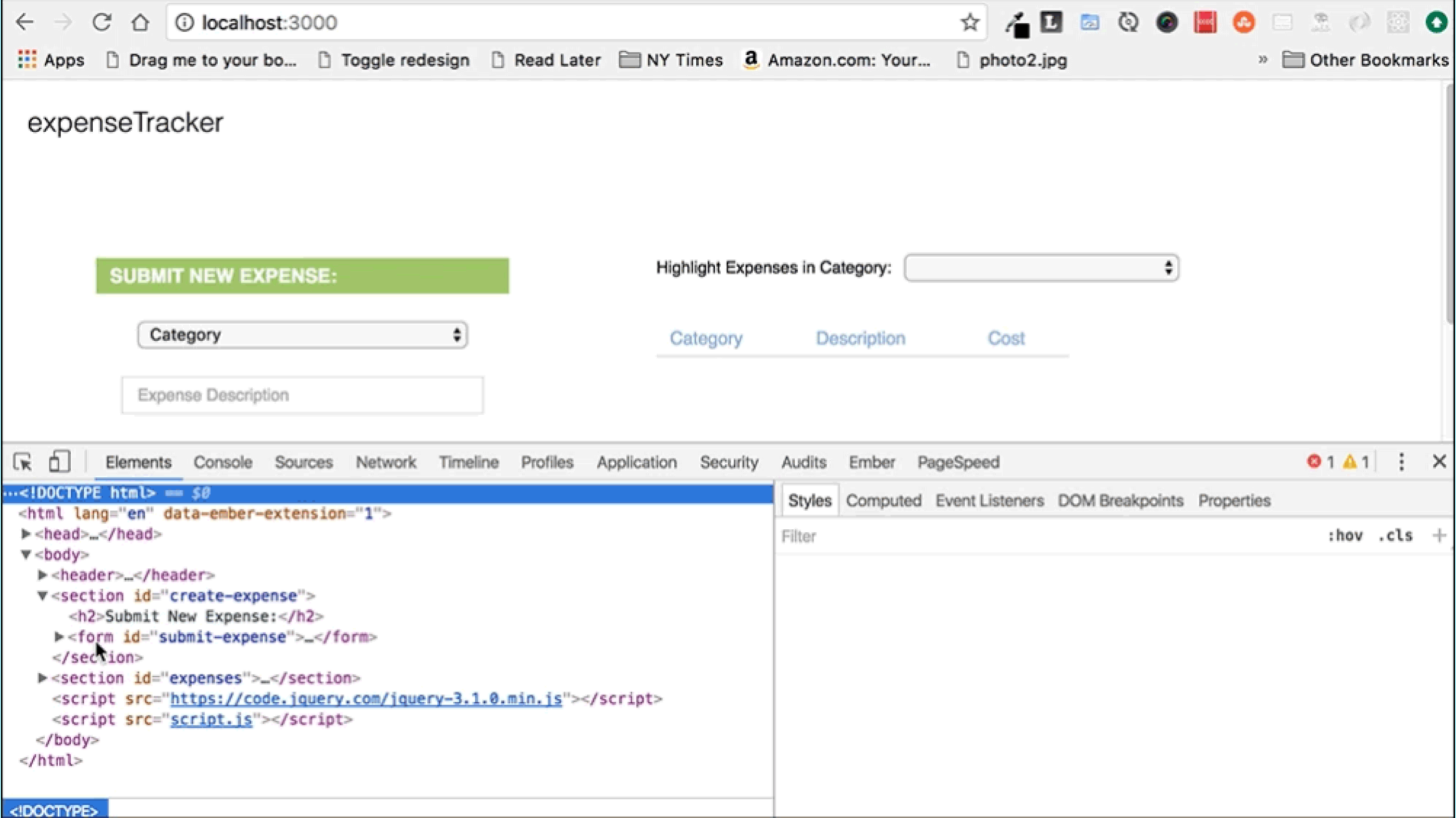

Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
1
1
<!DOCTYPE html>
<html lang="en" data-ember-extension="1">
▶<head>…</head>
▼<body>
▶<header>…</header>
▼<section id="create-expense">
<h2>Submit New Expense:</h2>
▼<form id="submit-expense">
▶<select name="category" class="form-field">…</select>
<input type="text" name="description" placeholder="Expense Description" class="form-field"> == $0
<input type="number" name="cost" placeholder="Cost" class="form-field">
<input type="submit" value="Submit Expense">
</form>
</section>
html
body
section#create-expense
form#submit-expense
input.form-field
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov
.cls
element.style {
}
input, select {
style.css:39
font-size: 12px;
display: block;
margin: ▶20px auto;
padding: ▶5px 10px;
width: 80%;
}
body, input {
style.css:1
font-family: 'Helvetica';
}

Inspecting DOM Events

Also in this sidebar is a tab labeled 'Event Listeners'. This is an important one for debugging user interactions on your application. In our expense tracker application, we've set up an event listener on our form for submitting a new expense:

```
$('#submit-expense').on('submit', (e) => {
  e.preventDefault();
  console.log('Submitting a new expense...');
});
```

We can verify that this event listener has been attached to its corresponding DOM node by selecting the `form` in the elements panel, and navigating to the 'Event Listeners' tab in the sidebar:


localhost:3000
Apps
Drag me to your bo...
Toggle redesign
Read Later
NY Times
Amazon.com: Your...
photo2.jpg
Other Bookmarks
expenseTracker
SUBMIT NEW EXPENSE:
Category
Expense Description
Highlight Expenses in Category:
Category
Description
Cost
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
1
1
<!DOCTYPE html> == $0
<html lang="en" data-ember-extension="1">
<head>…</head>
<body>
<header>…</header>
<section id="create-expense">
<h2>Submit New Expense:</h2>
<form id="submit-expense">…</form>
</section>
<section id="expenses">…</section>
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="script.js"></script>
</body>
</html>
Styles
Computed
Event Listeners
DOM Breakpoints
Properties
Filter
:hov
.cls
<!DOCTYPE>

The Console: Examining JavaScript Errors & Logging Information

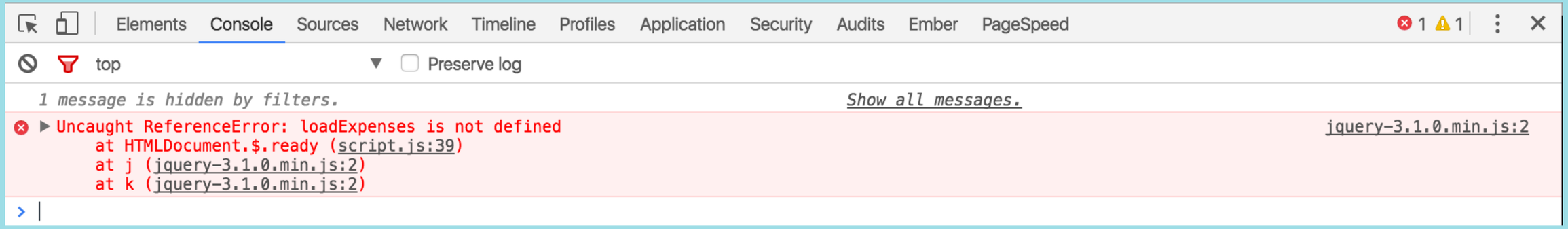
Many of the front-end bugs you encounter will be caused by the JavaScript in your application rather than your HTML or CSS. Tracking down these kinds of bugs can be tricky, but we can get a lot of information about the nature of a JavaScript error by using the console panel.

HELPFUL FOR:

- finding additional context about a JavaScript error
- tracing a JavaScript error to the exact line that caused it
- logging information about your code as it runs

Exploring JavaScript Errors in the console

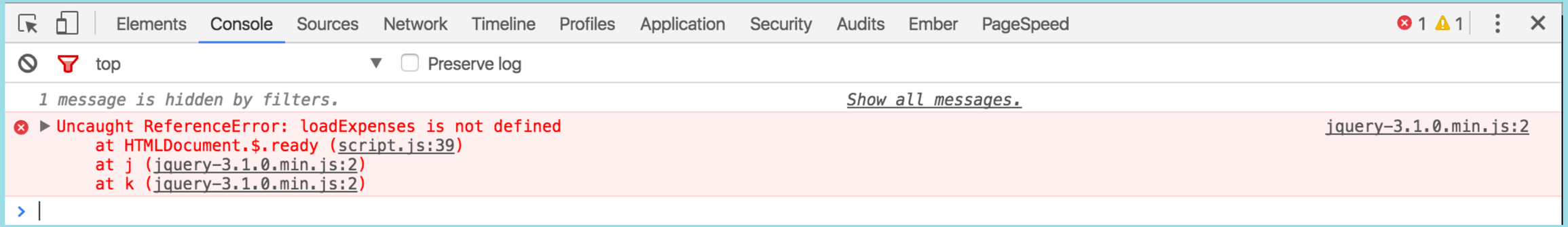
By default, the console panel will log any errors or warnings that it detects in your application. Warnings will be highlighted in yellow, and errors will be red. If you're following along with the expense tracker application, you'll notice we already have some errors in our console:



Each error in the console will provide you with the following information:

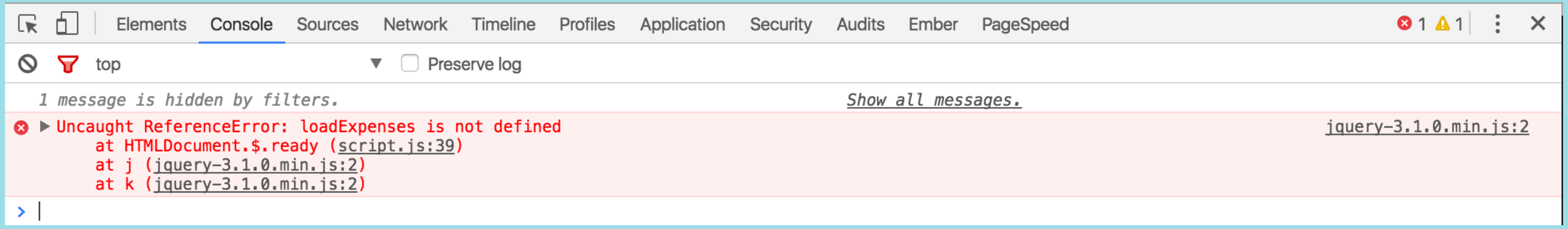
A description of the error

We seem to have a Reference Error to an undefined variable called `loadExpenses`



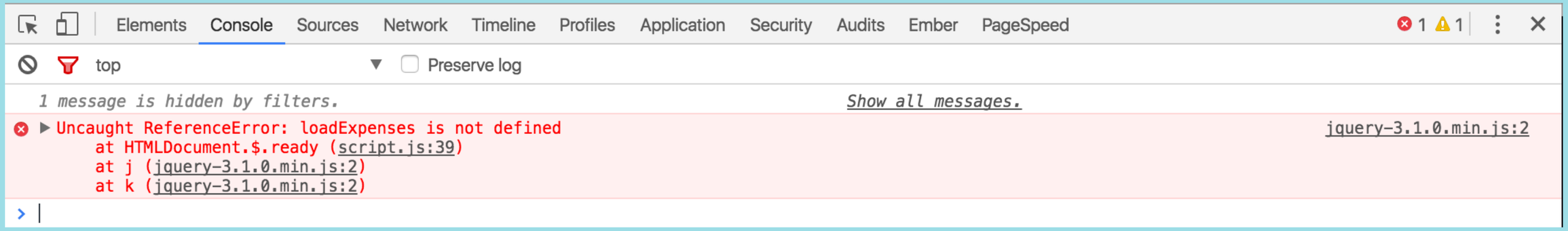
The file name and line number where the error occurred

On the right-hand side of the panel, it tells us the error is coming from a file called 'jquery-3.1.0.min.js' on line 2. But that doesn't seem right, because we didn't write that code and jQuery never has bugs! This is where the **stack trace** comes in.

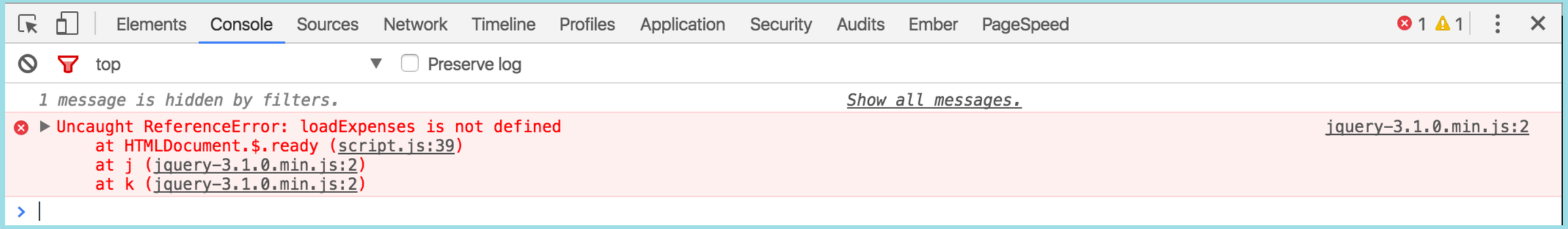


The stack trace allows you to follow the bug directly to where it originated

If you click the little triangle arrow next to the error in the console, you'll see some additional information expand. This is the line-by-line path of the bug that caused the code to throw an error. It's now much easier to see that the bug originated in our `script.js` file, on line 51.



We can quickly fix this error by looking through our code for a `loadExpenses` function. We don't seem to have one, which is why the console is telling us it's undefined. We do have a `getExpenses` function though. Let's call `getExpenses()` on line 51 instead of `loadExpenses()`.



Logging values with `console.log()`

Occassionally, you may want want to log custom information about the state of your code to the console. The console panel comes with a built-in `log` method that allows you to print any values you might be interested in seeing. You can think of the `console.log()` method as the equivalent of `puts` in Ruby.

Now that we've fixed the first JavaScript error we encountered, we can actually see one of our custom logging messages in in the console. Logging this success message was just a single line of code in our `getExpenses` function:

```
const getExpenses = () => {
  $.ajax({
    method: 'GET',
    dataType: 'json',
    url: '/expenses',
    success: function (data) {
      console.log('Expenses returned successfully!');
    },
    error: function (error) {
      console.error('There was some kind of error.');
    }
  });
};
```

It's nice to know that we have made a successful ajax call, but what if we want to log the data from our ajax response? Let's include the `data` argument from our success callback into our `console.log()`:

```
console.log('Expenses returned successfully: ', data);
```

You should now see the success message along with the array of expenses that were returned from that ajax request. This is a common way for developers to confirm the value of a variable or clarify what is being returned from a function. However, there are a couple of downsides to this method:

- **it encourages debugging by 'trial-and-error'** - you have to guess where in your code you should put the logging, and what values you need to log
- **logged data is stale** - your code continues to run even after the values have been logged, so you can't actually *do* anything with them
- **we tend to forget about them** - they'll sit around in our codebase and eventually get pushed to production

Luckily, there are alternative debugging strategies we can use that are a bit more powerful and efficient. We'll explore these further in the next section.

The Sources Panel: Live-Debugging of JavaScript Errors

The sources panel gives us an overview of all the files we have loaded into our application. We can examine our entire front-end codebase simply by clicking through the files in the left-hand sidebar. Being able to see our code line-by-line will help us isolate JavaScript errors and pinpoint exactly where our code is broken.

HELPFUL FOR:

- examining the files loaded into your application
- live-debugging JavaScript errors
- monitoring the values of application variables

Pausing Code Execution with `debugger`

Similar to the `console.log()` method we just learned about, we can also use `debugger` statements in our code to get feedback about our application's JavaScript.

The `debugger` statement is a bit more robust than `console.log()` because it will stop your app in its tracks whenever the browser reaches the line you placed it on.

For example, you can place a `debugger` within a function to pause the browser from running the script when it hits that portion of your code.

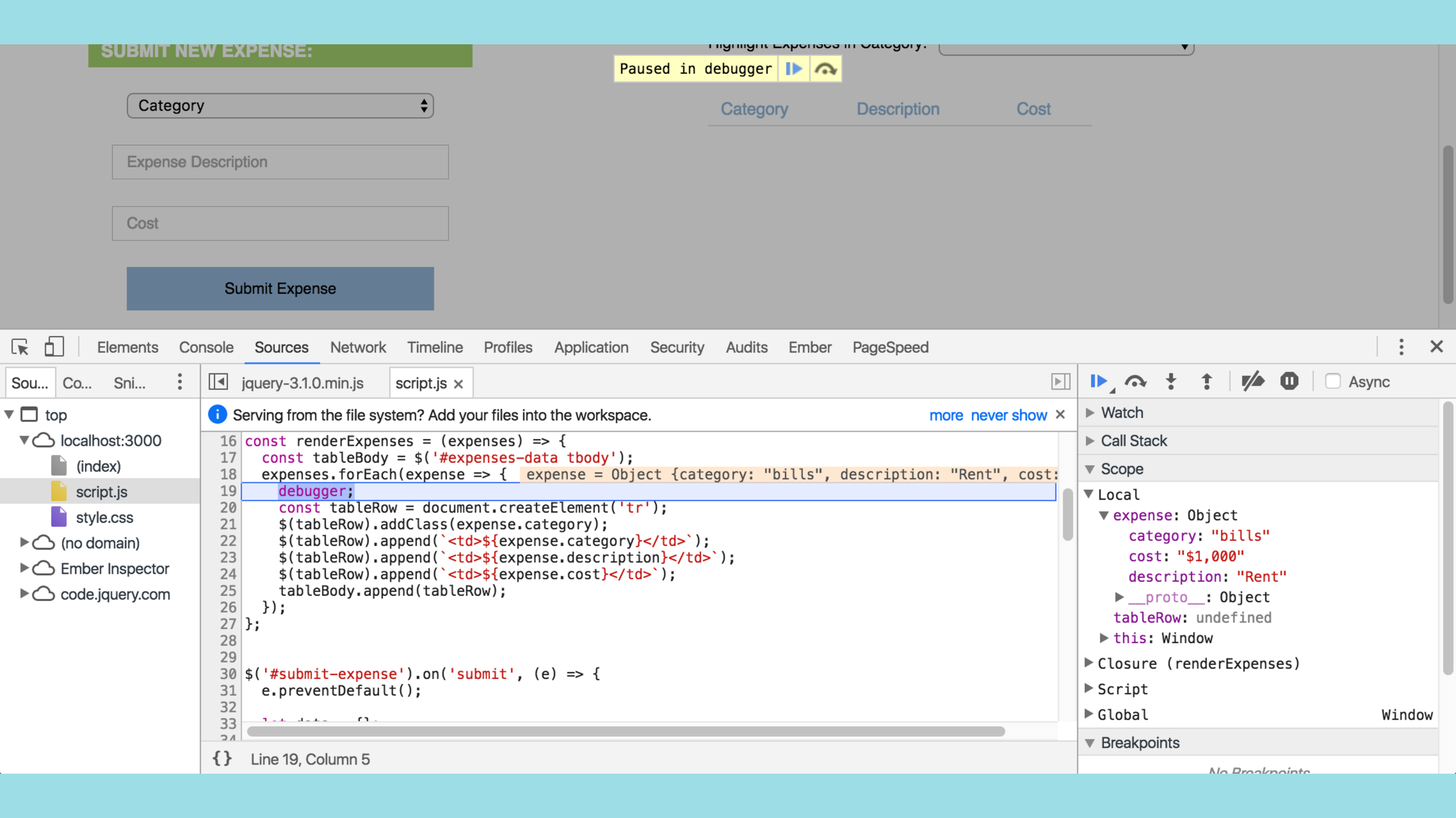
This gives you time to explore your code in the console, giving you access to any information your program has at that point in time.

NOTE - The console must be open for debugger to catch, otherwise the app will look normal and you won't get any error messages - if you get stuck, refresh your page while the console is open and go from there.

Let's put a `debugger` statement in our expenseTracker application. Now that we've successfully retrieved our expenses with an ajax call, we want to render them in the UI. We have a function called `renderExpenses` to handle this for us. It loops through each expense and appends it to the table body. Though we don't have a bug in this code, we can easily demonstrate how `debugger` statements pause our code execution by putting one in the loop:

```
const renderExpenses = (expenses) => {
  const tableBody = $('#expenses-data tbody');
  expenses.forEach(expense => {
    debugger;
    const tableRow = document.createElement('tr');
    $(tableRow).addClass(expense.category);
    $(tableRow).append(`<td>${expense.category}</td>`);
    $(tableRow).append(`<td>${expense.description}</td>`);
    $(tableRow).append(`<td>${expense.cost}</td>`);
    tableBody.append(tableRow);
  });
};
```

After putting the `debugger` in your loop, refresh the page and you should see the code pausing on each loop. Every time you resume the code execution (by clicking the blue arrow at the top of the browser window), you should see a new expense rendered to the table.


SUBMIT NEW EXPENSE:
Category
Expense Description
Cost
Submit Expense
Paused in debugger
Category
Description
Cost
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
Sou...
Co...
Sni...
jquery-3.1.0.min.js
script.js
top
localhost:3000
(index)
script.js
style.css
(no domain)
Ember Inspector
code.jquery.com
Serving from the file system? Add your files into the workspace.
more
never show
16 const renderExpenses = (expenses) => {
17 const tableBody = $('#expenses-data tbody');
18 expenses.forEach(expense => { expense = Object {category: "bills", description: "Rent", cost:
19 debugger;
20 const tableRow = document.createElement('tr');
21 $(tableRow).addClass(expense.category);
22 $(tableRow).append(`<td>${expense.category}</td>`);
23 $(tableRow).append(`<td>${expense.description}</td>`);
24 $(tableRow).append(`<td>${expense.cost}</td>`);
25 tableBody.append(tableRow);
26 });
27 };
28
29
30 $('#submit-expense').on('submit', (e) => {
31 e.preventDefault();
32
33
{}
Line 19, Column 5
Async
Watch
Call Stack
Scope
Local
expense: Object
category: "bills"
cost: "$1,000"
description: "Rent"
__proto__: Object
tableRow: undefined
this: Window
Closure (renderExpenses)
Script
Global
Window
Breakpoints

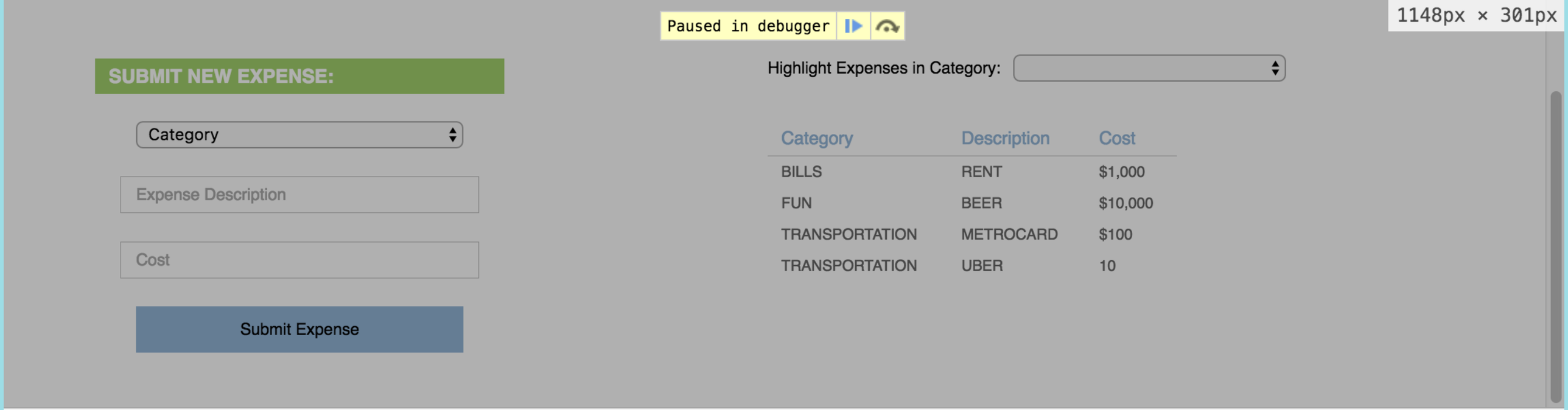
Pay special attention to the sidebar on the right of the devtool panel. Underneath the 'scope' section, you can explore all the variables that exist within your scope at the time of this pause. You'll notice the variable called expense will change as we iterate through each loop, because we are appending a new expense object to the DOM every time we pause our code.

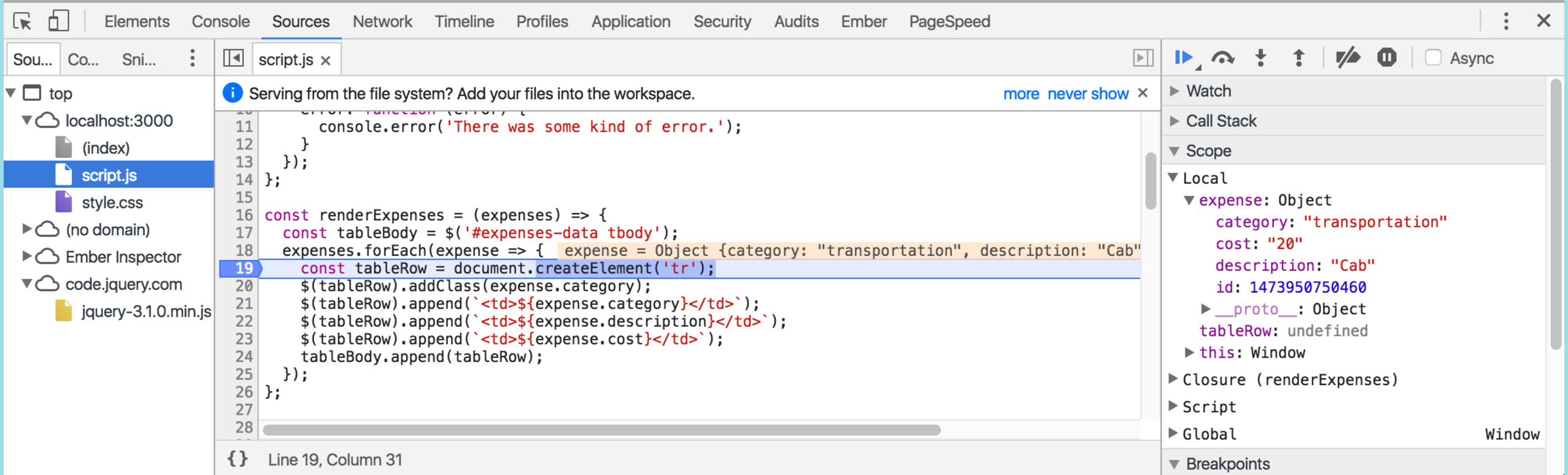
Setting Breakpoints

As mentioned, the `debugger` strategy has some advantages over using `console.log()`. The only setback with using `debugger` statements is that we have to continually toggle between our codebase and the browser, adding and removing debuggers from our code. The sources panel actually provides a way for us to streamline this debugging process a little bit.

When we want to set a `debugger` statement in a particular JavaScript file, we can simply open that file in the source panel by clicking on its name in the left-sidebar. Once we're viewing the code, we can set 'breakpoints' (essentially, these are debugger statements) by clicking on the line numbers where we want to pause code execution. This makes it easy and fast for us to add and remove as many breakpoints as we want.

Let's remove the `debugger` statement we adding in our codebase, and use a breakpoint from the sources panel instead. From the sources panel, click on `script.js` in the sidebar on the left. Then click on the line number corresponding to our first line in the loop function (`const tableRow = document.createElement('tr');`). If you've set the breakpoint correctly, the line number should turn blue. Refresh the page to see how breakpoints work in the same fashion as `debugger` statements:

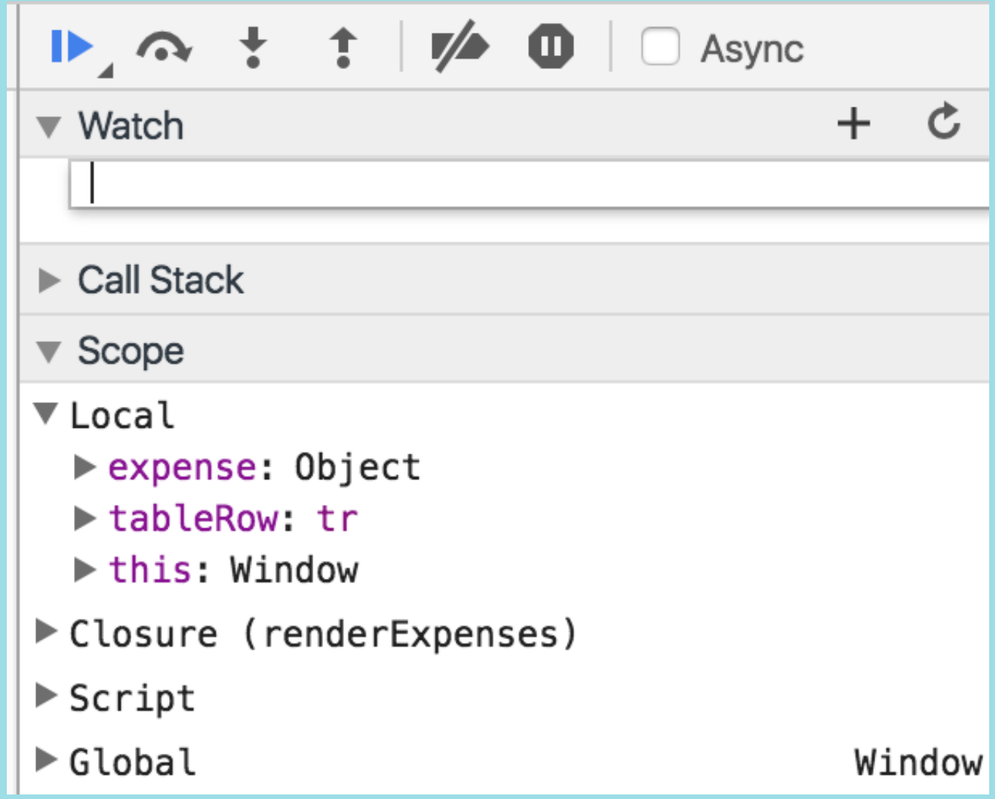

Paused in debugger
1148px × 301px
SUBMIT NEW EXPENSE:
Category
Expense Description
Cost
Submit Expense
Highlight Expenses in Category:
Category
Description
Cost
BILLS
RENT
$1,000
FUN
BEER
$10,000
TRANSPORTATION
METROCARD
$100
TRANSPORTATION
UBER
10


Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Ember
PageSpeed
Sou...
Co...
Sni...
script.js
top
localhost:3000
(index)
script.js
style.css
(no domain)
Ember Inspector
code.jquery.com
jquery-3.1.0.min.js
Serving from the file system? Add your files into the workspace.
more
never show
11 console.error('There was some kind of error.');
12 }
13 });
14 };
15
16 const renderExpenses = (expenses) => {
17 const tableBody = $('#expenses-data tbody');
18 expenses.forEach(expense => { expense = Object {category: "transportation", description: "Cab"
19 const tableRow = document.createElement('tr');
20 $(tableRow).addClass(expense.category);
21 $(tableRow).append(`<td>${expense.category}</td>`);
22 $(tableRow).append(`<td>${expense.description}</td>`);
23 $(tableRow).append(`<td>${expense.cost}</td>`);
24 tableBody.append(tableRow);
25 });
26 };
27
28
{} Line 19, Column 31
Async
Watch
Call Stack
Scope
Local
expense: Object
category: "transportation"
cost: "20"
description: "Cab"
id: 1473950750460
__proto__: Object
tableRow: undefined
this: Window
Closure (renderExpenses)
Script
Global
Window
Breakpoints

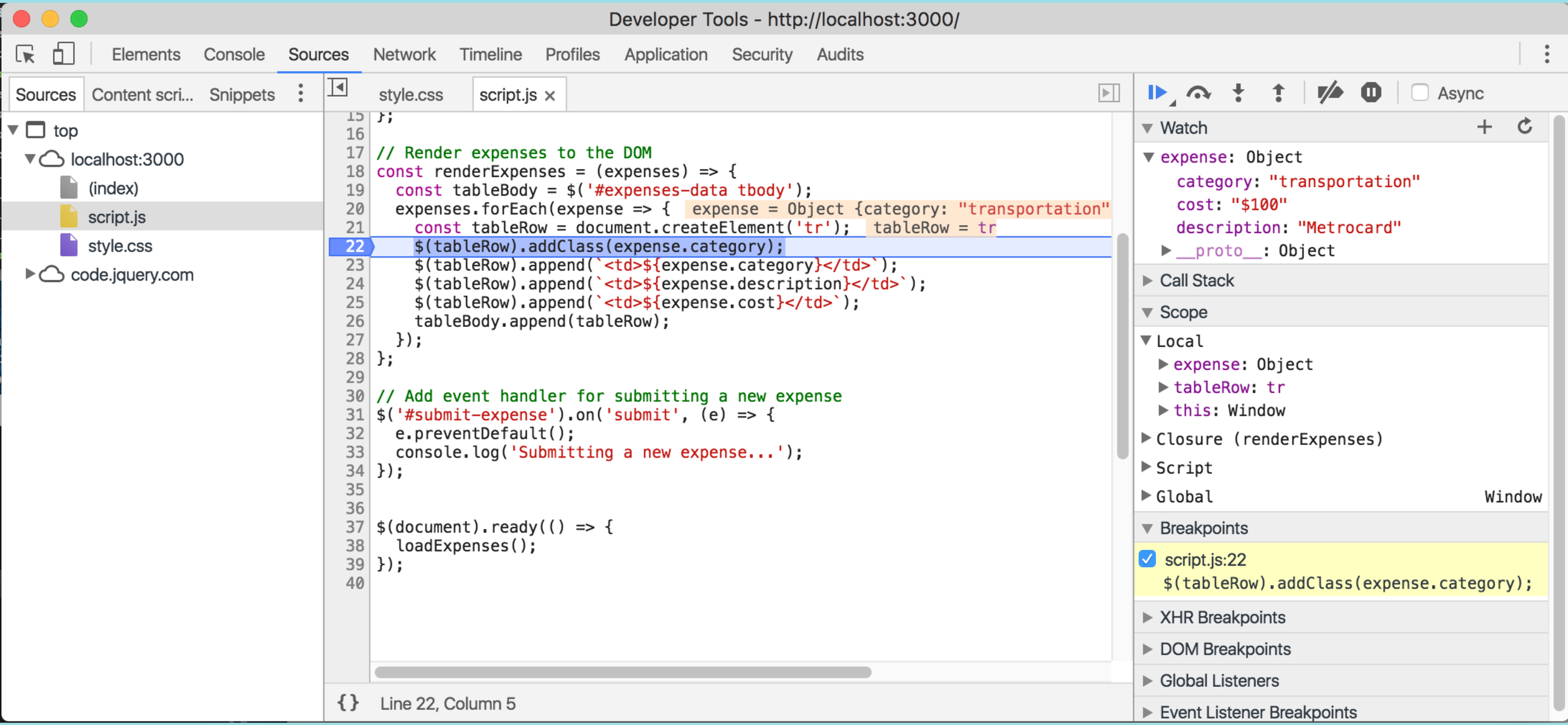
Watching Variables

Another way we can inspect the state of our application is by using watch expressions. In the sidebar above the 'scope' section we just explored, there is also a 'watch' section. In here, we can add the name of any variable in our application, and the panel will continuously monitor and display its value, no matter how many times it changes.

With our breakpoints still in place, let's add the variable expense to the watch section by clicking on the plus sign and typing 'expense' into the textbox that appears:



Now when we refresh our page and our code is paused on each loop, we should notice the expense variable in our watch expression updating with every loop:

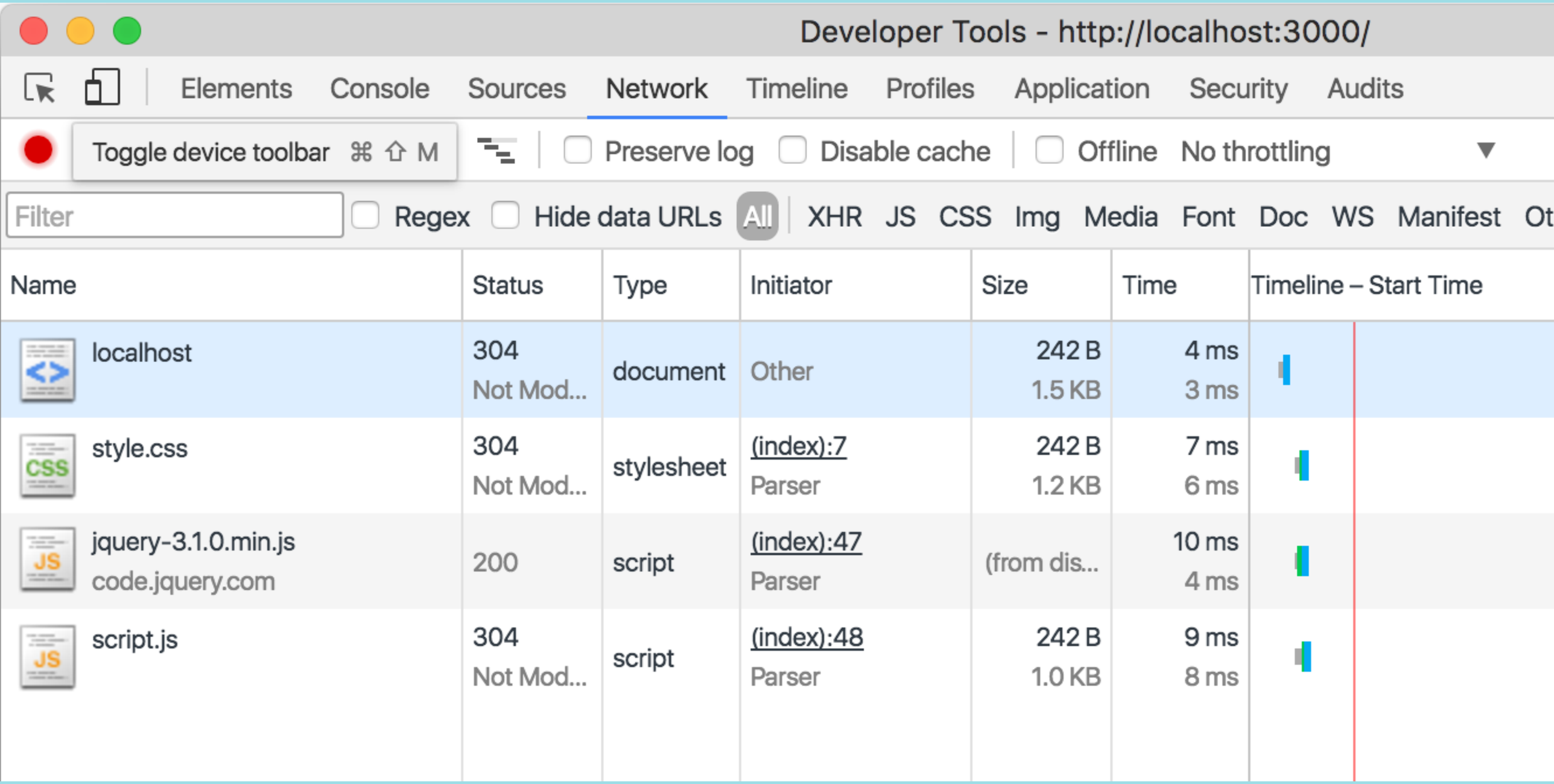

Developer Tools - http://localhost:3000/
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Sources
Content scri...
Snippets
top
localhost:3000
(index)
script.js
style.css
code.jquery.com
style.css
script.js
15 };
16
17 // Render expenses to the DOM
18 const renderExpenses = (expenses) => {
19 const tableBody = $('#expenses-data tbody');
20 expenses.forEach(expense => { expense = Object {category: "transportation"
21 const tableRow = document.createElement('tr'); tableRow = tr
22 $(tableRow).addClass(expense.category);
23 $(tableRow).append(`<td>${expense.category}</td>`);
24 $(tableRow).append(`<td>${expense.description}</td>`);
25 $(tableRow).append(`<td>${expense.cost}</td>`);
26 tableBody.append(tableRow);
27 });
28 };
29
30 // Add event handler for submitting a new expense
31 $('#submit-expense').on('submit', (e) => {
32 e.preventDefault();
33 console.log('Submitting a new expense...');
34 });
35
36
37 $(document).ready(() => {
38 loadExpenses();
39 });
40
{}
Line 22, Column 5
Async
Watch
expense: Object
category: "transportation"
cost: "$100"
description: "Metrocard"
__proto__: Object
Call Stack
Scope
Local
expense: Object
tableRow: tr
this: Window
Closure (renderExpenses)
Script
Global
Window
Breakpoints
script.js:22
$(tableRow).addClass(expense.category);
XHR Breakpoints
DOM Breakpoints
Global Listeners
Event Listener Breakpoints

The Network Panel: Debugging Communication with APIs

HELPFUL FOR:

- examining network requests and responses
- debugging issues with sending or receiving API data

The network tab shows you information about all the server requests that were made when loading or interacting with your application. Specifically, you can view the response and status code of each request and how long it took to make the trip to the server and back. This panel is most helpful when you are debugging data that is being sent to and received from an API.

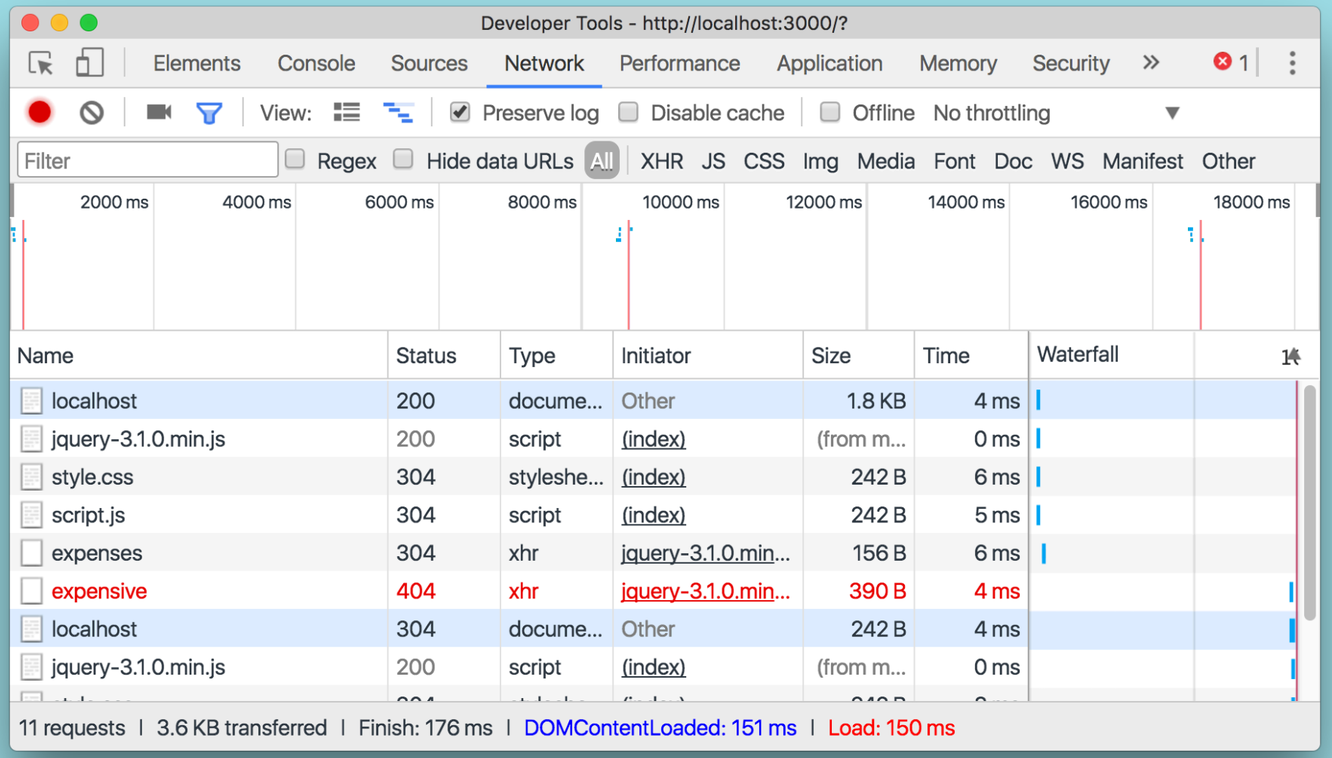

Developer Tools - http://localhost:3000/
Elements
Console
Sources
Network
Timeline
Profiles
Application
Security
Audits
Toggle device toolbar ⌘ ⇧ M
Preserve log
Disable cache
Offline
No throttling
Filter
Regex
Hide data URLs
All
XHR
JS
CSS
Img
Media
Font
Doc
WS
Manifest
Name
Status
Type
Initiator
Size
Time
Timeline – Start Time
localhost
304
Not Mod...
document
Other
242 B
1.5 KB
4 ms
3 ms
style.css
304
Not Mod...
stylesheet
(index):7
Parser
242 B
1.2 KB
7 ms
6 ms
jquery-3.1.0.min.js
code.jquery.com
200
script
(index):47
Parser
(from dis...
10 ms
4 ms
script.js
304
Not Mod...
script
(index):48
Parser
242 B
1.0 KB
9 ms
8 ms

One thing you'll notice about the network panel is that it won't automatically record any network requests unless you have the panel open at the time they're made. Additionally, the request list will refresh every time you navigate to a new page. You can change this functionality by selecting **Preserve Log** at the top of the panel.

HTTP Status Codes

The network panel is only useful if you understand what types of responses you can get back from a server. Familiarize yourself with all the different status codes you might receive from a network request. These status codes give you information about what happened during your request - was it successful? Did it fail? Why did it fail?

Lucky for us, the network panel automatically highlights any failed requests in red:



When we're trying to debug a network request and we see that it is highlighted in red in the network panel, we can click on the request under the 'Name' column to view more details about it. Here we'll be able to see more information about why it failed and what data was sent with the request:

Name

eval.js

× Headers Preview Response Timing

General

Request URL: chrome-extension://hbhhpaojmpfimakffndmpmpndcmonkfa/generated/eval.js
Request Method: GET
Status Code: 200 OK (from disk cache)
Referrer Policy: no-referrer-when-downgrade

Response Headers

Access-Control-Allow-Origin: *
cache-control: no-cache
Content-Security-Policy: script-src 'self' blob: filesystem: chrome-extension-resource:; object-src 'self' blob: filesystem:;
ETag: "Vi4RUR+RGdQdCa4H4CXgu6tdaJs="

Request Headers

⚠ **Provisional headers are shown**
Referer: http://frontend.turing.io/lessons/debugging-with-devtools.html
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3046.0 Safari/537.36

The most important tabs in this view are the HEADERS tab and the RESPONSE tab. The Headers tab will give you detailed information about what was sent in the request. The response tab will show you what you actually received back from the server. This could be the JSON data you requested, an error message, or even an entire HTML file.

Practice

Let's put this information into practice by trying to solve a couple of bugs in our expense application.
Read through the first buggy scenario here and checkout the `category-highlighting` branch to get started fixing that code.

Review

Review

- In what situations is `console.log()` a helpful tool?
  - When is it not?

Review

- In what situations is `console.log()` a helpful tool?
  - When is it not?

- In what situations is `debugger` a helpful tool?
  - When is it not?

Review

Review

- How can the `Elements` tab be used for debugging?

Review

- How can the `Elements` tab be used for debugging?
- How can the `Console` tab be used for debugging?

Review

- How can the `Elements` tab be used for debugging?
- How can the `Console` tab be used for debugging?
- How can the `Sources` tab be used for debugging?

Review

- How can the `Elements` tab be used for debugging?
- How can the `Console` tab be used for debugging?
- How can the `Sources` tab be used for debugging?
- How can the `Network` tab be used for debugging?

For more details and information about other ways to dig into your js, checkout the Chrome Documentation.