Original Repository: airbnb/javascript

Airbnb JavaScript Style Guide() {

Ein vernünftiger Ansatz für einen JavaScript-Style-Guide

Inhaltsverzeichnis

Datentypen

- **Primitive Typen**: Bei primitiven Datentypen wird immer direkt auf deren Wert zugegriffen.
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`

```
var foo = 1
    ,bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- **Komplexe Typen**: Bei komplexen Datentypen wird immer auf eine Referenz zugegriffen.
  - `object`
  - `array`

- `function`

```
var foo = [1, 2]
    ,bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

[⬆]

Objekte

- Benutze die `literal syntax`, um Objekte zu erzeugen.

```
// schlecht
var item = new Object();

// gut
var item = {};
```

- Benutze keine reservierten Wörter für Attribute.

```
// schlecht
var superman = {
  class: 'superhero'
  ,default: { clark: 'kent' }
  ,private: true
};

// gut
var superman = {
  klass: 'superhero'
  ,defaults: { clark: 'kent' }
  ,hidden: true
};
```

[⬆]

Arrays

- Benutze die `literal syntax`, um Arrays zu erzeugen.

```
// schlecht
var items = new Array();

// gut
var items = [];
```

- Wenn du die Array-Länge nicht kennst, benutze `Array#push`.

```
var someStack = [];

// schlecht
someStack[someStack.length] = 'abracadabra';

// gut
someStack.push('abracadabra');
```

- Wenn du ein Array kopieren möchtest, benutze `Array#slice`. jsPerf

```
var len = items.length
    ,itemsCopy = []
    ,i;
```

```
// schlecht
for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// gut
itemsCopy = Array.prototype.slice.call(items);
```

[⬆]

Zeichenketten

- Benutze einfache Anführungszeichen `''` für Zeichenketten

```
// schlecht
var name = "Bob Parr";

// gut
var name = 'Bob Parr';

// schlecht
var fullName = "Bob " + this.lastName;

// gut
var fullName = 'Bob ' + this.lastName;
```

- Zeichenketten die länger als 80 Zeichen lang sind, sollten mit Hilfe von `string concatenation` auf mehrere Zeilen aufgeteilt werden.

- Beachte: Benutzt man `string concatenation` zu oft kann dies die performance beeinträchtigen. jsPerf & Discussion

```
// schlecht
var errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about hou

// schlecht
var errorMessage = 'This is a super long error that \
was thrown because of Batman. \
When you stop to think about \
how Batman had anything to do \
with this, you would get nowhere \
fast.';

// gut
var errorMessage = 'This is a super long error that ' +
  'was thrown because of Batman.' +
  'When you stop to think about ' +
  'how Batman had anything to do ' +
  'with this, you would get nowhere ' +
  'fast.';
```

- Wenn man im Programmverlauf eine Zeichenkette dynamisch zusammensetzen muss, sollte man `Array#join` einer `string concatenation` vorziehen. Vorallem für den IE. jsPerf.

```
var items
    ,messages
    ,length, i;

messages = [{
    state: 'success'
    ,message: 'This one worked.'
},{
    state: 'success'
    ,message: 'This one worked as well.'
},{
    state: 'error'
    ,message: 'This one did not work.'
}];
```

```
length = messages.length;

// schlecht
function inbox(messages) {
  items = '<ul>';

  for (i = 0; i < length; i++) {
    items += '<li>' + messages[i].message + '</li>';
  }

  return items + '</ul>';
}

// gut
function inbox(messages) {
  items = [];

  for (i = 0; i < length; i++) {
    items[i] = messages[i].message;
  }

  return '<ul><li>' + items.join('</li><li>') + '</li></ul>';
}
```

[⬆]

Funktionen

- Funktionsausdrücke:

```
// anonyme Fuktionsausdrücke
var anonymous = function() {
  return true;
};

// benannte Funktionsausdrücke
var named = function named() {
  return true;
};

// direkt ausgeführte Funktionsausdrücke (IIFE)
(function() {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

- Vermeide Funktionen in `non-function blocks` zu deklarieren. Anstelle sollte die Funktion einer Variablen zugewiesen werden. Dies hat den Grund, dass die verschiedenen Browser dies unterschiedlich interpretieren.

- **Beachte:** ECMA-262 definiert einen Block als eine Abfolge/Liste von Statements. Eine Funktion hingegen ist **kein** Statement. Read ECMA-262's note on this issue.

```
// schlecht
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// gut
if (currentUser) {
  var test = function test() {
    console.log('Yup.');
  };
}
```

- Benenne einen Parameter nie `arguments`, denn dies wird das `arguments`-Objekt, dass in jedem Funktionskörper zur Verfügung

steht, überschreiben.

```
// schlecht
function nope(name, options, arguments) {
  // ...stuff...
}

// gut
function yup(name, options, args) {
  // ...stuff...
}
```

[⬆]

Eigenschaften

- Benutze die Punktnotation, um auf die Eigenschaften eines Objekts zuzugreifen.

```
var luke = {
  jedi: true
  ,age: 28
};

// schlecht
var isJedi = luke['jedi'];

// gut
var isJedi = luke.jedi;
```

- Benutze die Indexnotation `[]`, um auf die Eigenschaften eines Objekts zuzugreifen, sofern der Index eine Variable ist.

```
var luke = {
  jedi: true
  ,age: 28
};

function getProp(prop) {
  return luke[prop];
}

var isJedi = getProp('jedi');
```

[⬆]

Variablen

- Benutze immer `var`, um Variablen zu deklarieren. Tut man dies nicht, werden die Variablen im globalen Namespace erzeugt – was nicht gewüscht werden sollte.

```
// schlecht
superPower = new SuperPower();

// gut
var superPower = new SuperPower();
```

- Benutze immer nur ein `var`, um mehrere aufeinanderfolgende Variablen zu deklarieren. Deklariere jede Variable auf einer eigenen Zeile.

```
// schlecht
var items = getItems();
var goSportsTeam = true;
var dragonball = 'z';

// gut
var items = getItems()
```

```
    ,goSportsTeam = true
    ,dragonball = 'z';
```

- Deklariere Variablen ohne direkte Zuweisung immer als letztes. Dies ist vorallem hilfreich, wenn man später eine Variable anhand einer zuvor deklarierten Variable initialisieren möchte.

```
// schlecht
var i, len, dragonball
    ,items = getItems()
    ,goSportsTeam = true;

// schlecht
var i, items = getItems()
    ,dragonball
    ,goSportsTeam = true
    ,len;

// gut
var items = getItems()
    ,goSportsTeam = true
    ,dragonball
    ,i
    ,length;
```

- Weise den Wert einer Variable, wenn möglich, immer am Anfang des Gültigkeitsbereichs zu. Dies hilft Problemen mit der Variablendeklaration vorzubeugen.

```
// schlecht
function() {
  test();
  console.log('doing stuff..');

  //..other stuff..

  var name = getName();

  if (name === 'test') {
    return false;
  }

  return name;
}

// gut
function() {
  var name = getName();

  test();
  console.log('doing stuff..');

  //..other stuff..

  if (name === 'test') {
    return false;
  }

  return name;
}

// schlecht
function() {
  var name = getName();

  if (!arguments.length) {
    return false;
  }

  return true;
}
```

```
// gut
function() {
  if (!arguments.length) {
    return false;
  }

  var name = getName();

  return true;
}
```

[⬆]

Hoisting

- Variablendeklarationen werden vom Interpreter an den Beginn eines Gültigkeitbereichs genommen, genannt (`hoisting`). Wohingegen die Zuweisung an der ursprünglichen Stelle bleibt.

```
// Dies wird nicht funktionen (angenommen
// notDefined ist keine globale Variable)
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// Wird eine Variable nach seiner ersten
// Referenzierung deklariert, funktioniert
// dies dank des hoistings.
// Beachte aber, dass die Zuweisung von true
// erst nach der Referenzierung stattfindet.
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// Der Interpreter nimmt die Variablendeklaration
// an den Beginn des Gültigkeitbereichs.
// So kann das Beispiel wiefolgt umgeschrieben
// werden:
function example() {
  var declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}
```

- Anonyme Funktionen `hoisten` ihren Variablennamen, aber nicht die Funktionszuweisung.

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function() {
    console.log('anonymous function expression');
  };
}
```

- Benannte Funktionen `hoisten` ihren Variablennamen, aber nicht der Funktionsname oder Funktionskörper.

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
```

```
    console.log('Flying');
  };

  // Das gleiche gilt, wenn der Funktionsname
  // derselbe ist, wie der Variablenname
  function example() {
    console.log(named); // => undefined

    named(); // => TypeError named is not a function

    var named = function named() {
      console.log('named');
    };
  }
}
```

- Funktionsdeklarationen `hoisten` ihren Namen und ihren Funktionskörper.

```
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}
```

- Für weitere Informationen siehe hier: JavaScript Scoping & Hoisting by Ben Cherry

[⬆]

Bedingungen und Gleichheit

- Ziehe `===` und `!==` gegenüber `==` und `!=` vor.
- Bedingungsausdrücke werden immer gezwungen der `ToBoolean` Methode ausgewertet zu werden. Diese folgt den folgenden einfachen Grundregeln:
  - **Objekte** werden als **true** gewertet
  - **Undefined** wird als **false** gewertet
  - **Null** wird als **false** gewertet
  - **Booleans** werden als **der Wert des Booleans** gewertet
  - **Zahlen** werden als **false** gewertet sofern **+0, -0, or NaN**, ansonsten als **true**
  - **Zeichenketten** werden als **false** gewertet, sofern sie leer ist `''`, ansonsten als **true**

```
if ([0]) {
  // true
  // Arrays sind Objekte und Objekte werden als true ausgewertet
}
```

- Benutze `shortcuts`

```
// schlecht
if (name !== '') {
  // ...stuff...
}

// gut
if (name) {
  // ...stuff...
}

// schlecht
if (collection.length > 0) {
  // ...stuff...
}

// gut
if (collection.length) {
```

```
  // ...stuff...
}
```

- Für weitere Informationen siehe hier: Truth Equality and JavaScript by Angus Croll

[⬆]

Blöcke

- Benutze geschweifte Klammern für alle mehrzeiligen Blöcke.

```
// schlecht
if (test)
  return false;

// gut
if (test) return false;

// gut
if (test) {
  return false;
}

// schlecht
function() { return false; }

// gut
function() {
  return false;
}
```

[⬆]

Kommentare

- Benutze `/** ... */` für mehrzeilige Kommentare. Daran kann eine Beschreibung, eine Typendefinition und Werte für alle Parameter und den Rückgabewert angegeben werden.

```
// schlecht
// make() returns a new element
// based on the passed in tag name
//
// @param <String> tag
// @return <Element> element
function make(tag) {

  // ...stuff...

  return element;
}

// gut
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param <String> tag
 * @return <Element> element
 */
function make(tag) {

  // ...stuff...

  return element;
}
```

- Benutze `//` für einzeilige Kommentare. Platziere den Kommentar auf einer separaten Zeile oberhalb der beschriebenen Zeile. Vor

den Kommentar kommt eine Leerzeile.

```
// schlecht
var active = true;  // is current tab

// gut
// is current tab
var active = true;

// schlecht
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}

// gut
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}
```

[⬆]

Whitespace

- Benutze weiche Tabulatoren (`soft tabs`) mit 2 Leerzeichen.

```
// schlecht
function() {
····var name;
}

// schlecht
function() {
·var name;
}

// gut
function() {
··var name;
}
```

- Platziere ein Leerzeichen vor einer öffnenden Klammer.

```
// schlecht
function test(){
  console.log('test');
}

// gut
function test() {
  console.log('test');
}

// schlecht
dog.set('attr',{
  age: '1 year'
  ,breed: 'Bernese Mountain Dog'
});

// gut
```

```
dog.set('attr', {
  age: '1 year'
  ,breed: 'Bernese Mountain Dog'
});
```

- Platziere eine Leerzeile an das Ende der Datei.

```
// schlecht
(function(global) {
  // ...stuff...
})(this);
```

```
// gut
(function(global) {
  // ...stuff...
})(this);
```

- Rücke bei langen Methodenverkettungen ein.

```
// schlecht
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// gut
$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();

// schlecht
var leds = stage.selectAll('.led').data(data).enter().append("svg:svg").class('led', true)
    .attr('width',  (radius + margin) * 2).append("svg:g")
    .attr("transform", "translate(" + (radius + margin) + "," + (radius + margin) + ")")
    .call(tron.led);

// gut
var leds = stage.selectAll('.led')
    .data(data)
  .enter().append("svg:svg")
    .class('led', true)
    .attr('width',  (radius + margin) * 2)
  .append("svg:g")
    .attr("transform", "translate(" + (radius + margin) + "," + (radius + margin) + ")")
    .call(tron.led);
```

[⬆]

Führende Kommas

- **Ja.**

```
// schlecht
var once,
    upon,
    aTime;

// gut
var once
    ,upon
    ,aTime;

// schlecht
var hero = {
  firstName: 'Bob',
  lastName: 'Parr',
```

```
  heroName: 'Mr. Incredible',
  superPower: 'strength'
};

// gut
var hero = {
    firstName: 'Bob'
    ,lastName: 'Parr'
    ,heroName: 'Mr. Incredible'
    ,superPower: 'strength'
};
```

[⬆]

Semikolons

- **Ja.**

```
// schlecht
(function() {
  var name = 'Skywalker'
  return name
})()

// gut
(function() {
  var name = 'Skywalker';
  return name;
})();
```

[⬆]

Typumwandlung

- Benutze `type coercion` am Anfang eines Statements.
- Bei Zeichenketten:

```
//  => this.reviewScore = 9;

// schlecht
var totalScore = this.reviewScore + '';

// gut
var totalScore = '' + this.reviewScore;

// schlecht
var totalScore = '' + this.reviewScore + ' total score';

// gut
var totalScore = this.reviewScore + ' total score';
```

- Benutze immer `parseInt` für Zahlen und gebe immer eine Basis für die Typumwandlung an.
- Wenn man aus Performancegründen kein `parseInt` verwenden will und ein `Bitshifting` benutzt, sollte man einen Kommentar hinterlassen, wieso dies gemacht wurde.

```
var inputValue = '4';

// schlecht
var val = new Number(inputValue);

// schlecht
var val = +inputValue;

// schlecht
var val = inputValue >> 0;
```

```
// schlecht
var val = parseInt(inputValue);

// gut
var val = Number(inputValue);

// gut
var val = parseInt(inputValue, 10);

// gut
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
var val = inputValue >> 0;
```

- Bei Booleans:

```
var age = 0;

// schlecht
var hasAge = new Boolean(age);

// gut
var hasAge = Boolean(age);

// gut
var hasAge = !!age;
```

[⬆]

Namenskonventionen

- Benutze keine `einzeichigen` Namen. Die Namen sollten beschreibend sein.

```
// schlecht
function q() {
  // ...stuff...
}

// gut
function query() {
  // ..stuff..
}
```

- Benutze `camelCase`, um Objekte, Funktionen und Instanzen zu benennen.

```
// schlecht
var OBJEcttssss = {};
var this_is_my_object = {};
var this-is-my-object = {};
function c() {};
var u = new user({
  name: 'Bob Parr'
});

// gut
var thisIsMyObject = {};
function thisIsMyFunction() {};
var user = new User({
  name: 'Bob Parr'
});
```

- Benutze `PascalCase`, um Klassen und Konstrukturen zu benennen.

```
// schlecht
function user(options) {
  this.name = options.name;
}

var bad = new user({
  name: 'nope'
});

// gut
function User(options) {
  this.name = options.name;
}

var good = new User({
  name: 'yup'
});
```

- Benutze führende Untenstriche `_`, um private Eigenschaften zu benennen.

```
// schlecht
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// gut
this._firstName = 'Panda';
```

- Um eine Referenz an `this` zuzuweisen, benutze `_this`.

```
// schlecht
function() {
  var self = this;
  return function() {
    console.log(self);
  };
}

// schlecht
function() {
  var that = this;
  return function() {
    console.log(that);
  };
}

// gut
function() {
  var _this = this;
  return function() {
    console.log(_this);
  };
}
```

- Gib deinen Funktionen einen Namen. Dies ist hilfreich für den `stack trace`.

```
// schlecht
var log = function(msg) {
  console.log(msg);
};

// gut
var log = function log(msg) {
  console.log(msg);
};
```

[⬆]

Zugriffsmethoden

- Zugriffsmethoden für Objekteigenschaften sind nicht von Nöten.
- Macht man dennoch Zugriffsmethoden, benutze `getVal()` und `setVal('hello')`.

```
// schlecht
dragon.age();

// gut
dragon.getAge();

// schlecht
dragon.age(25);

// gut
dragon.setAge(25);
```

- Wenn die Eigenschaft ein Boolean ist, benutze `isVal()` oder `hasVal()`.

```
// schlecht
if (!dragon.age()) {
  return false;
}

// gut
if (!dragon.hasAge()) {
  return false;
}
```

- Es ist in Ordnung `get()`- und `set()`-Methoden zu erstellen, aber sei konsistent.

```
function Jedi(options) {
  options || (options = {});
  var lightsaber = options.lightsaber || 'blue';
  this.set('lightsaber', lightsaber);
}

Jedi.prototype.set = function(key, val) {
  this[key] = val;
};

Jedi.prototype.get = function(key) {
  return this[key];
};
```

[⬆]

Konstruktoren

- Weise die Methoden dem `prototype` des Objektes zu, anstelle den `prototype` mit einem neuen Objekt zu überschreiben. Wenn man den `prototype` überschreibt wird eine Vererbung unmöglich, denn damit wird die Basis überschrieben!

```
function Jedi() {
  console.log('new jedi');
}

// schlecht
Jedi.prototype = {
  fight: function fight() {
    console.log('fighting');
  }

  ,block: function block() {
    console.log('blocking');
  }
};
```

```
// gut
Jedi.prototype.fight = function fight() {
  console.log('fighting');
};

Jedi.prototype.block = function block() {
  console.log('blocking');
};
```

- Methoden können `this` zurückgeben, um eine Methodenverkettung zu ermöglichen.

```
// schlecht
Jedi.prototype.jump = function() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
};

var luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20) // => undefined

// gut
Jedi.prototype.jump = function() {
  this.jumping = true;
  return this;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
  return this;
};

var luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- Es ist in Ordnung eine eigene `toString()`-Methode zu schreiben, aber man sollte sicherstellen, dass diese korrekt funktioniert und keine Nebeneffekte hat.

```
function Jedi(options) {
  options || (options = {});
  this.name = options.name || 'no name';
}

Jedi.prototype.getName = function getName() {
  return this.name;
};

Jedi.prototype.toString = function toString() {
  return 'Jedi - ' + this.getName();
};
```

[⬆]

Module

- Ein Modul sollte mit einem `!` beginnen. Dies stellt sicher, dass wenn in einem Modul das abschliessende Semikolon vergessen wurde, keine Fehler entstehen, wenn die Scripte zusammengeschnitten werden.
- Eine Datei sollte in `camelCase` benannt sein, in einem Ordner mit dem selben Namen liegen und dem Namen entsprechen mit dem es exportiert wird.
- Benutze eine Methode `noConflict()`, welche das exportierte Modul auf die vorhergehende Version setzt und diese zurück gibt.

- Deklariere immer `'use strict';` am Anfang des Moduls.

```
// fancyInput/fancyInput.js

!function(global) {
  'use strict';

  var previousFancyInput = global.FancyInput;

  function FancyInput(options) {
    this.options = options || {};
  }

  FancyInput.noConflict = function noConflict() {
    global.FancyInput = previousFancyInput;
    return FancyInput;
  };

  global.FancyInput = FancyInput;
}(this);
```

[⬆]

jQuery

- Stelle allen jQuery-Objektvariablen ein `$` voran.

```
// schlecht
var sidebar = $('.sidebar');

// gut
var $sidebar = $('.sidebar');
```

- Speichere `jQuery lookups`, sofern sie mehrmals gebraucht werden.

```
// schlecht
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// gut
function setSidebar() {
  var $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- Für DOM-Abfragen benutze `Cascading`: `$('.sidebar ul')` oder parent > child `$('.sidebar > .ul')`. jsPerf

- Benutze `find` mit `scoped jQuery object queries`

```
// schlecht
$('.sidebar', 'ul').hide();

// schlecht
$('.sidebar').find('ul').hide();
```

```
// gut
$('.sidebar ul').hide();

// gut
$('.sidebar > ul').hide();

// gut (langsamer)
$sidebar.find('ul');

// gut (schneller)
$($sidebar[0]).find('ul');
```

[⬆]

ECMAScript 5 Kompatibilität

- Verweis auf Kangax's ES5 Kompatibilitätstabelle

[⬆]

Testing

- **Ja.**

```
function() {
  return true;
}
```

[⬆]

Performance

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

[⬆]

Ressourcen

Lese dieses

- Annotated ECMAScript 5.1

Andere Styleguides

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

Andere Styles

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks

Bücher

- JavaScript: The Good Parts - Douglas Crockford

- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch

Blogs

- DailyJS
- JavaScript Weekly
- JavaScript, JavaScript...
- Bocoup Weblog
- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- Dustin Diaz
- nettuts

[⬆]

In the Wild

Dies ist eine Liste von Organisationen, welche diesen Style Guide benutzen. Sende uns einen `Pull request` oder öffne einen `issue` und wir werden dich der Liste hinzufügen.

- **Airbnb**: airbnb/javascript
- **American Insitutes for Research**: AIRAST/javascript
- **ExactTarget**: ExactTarget/javascript
- **GoCardless**: gocardless/javascript
- **GoodData**: gooddata/gdc-js-style
- **How About We**: howaboutwe/javascript
- **MinnPost**: MinnPost/javascript
- **National Geographic**: natgeo/javascript
- **Razorfish**: razorfish/javascript-style-guide
- **Shutterfly**: shutterfly/javascript

[⬆]

Übersetzungen

Dieser Styleguide ist in den folgenden Sprachen erhältlich:

- :en: **Englisch**: airbnb/javascript
- :jp: **Japanisch**: mitsuruog/javacript-style-guide

[⬆]

The JavaScript Style Guide Guide

- Reference

[⬆]

Contributors

- View Contributors

[⬆]

Lizenz

[⬆]

};