DUY TÂN UNIVERSITY
International School

Capstone Project 2

CMU-SE 451

Code Standard

Version 1.0

Date: 10 May 2022

SENIOR PROJECT MANAGEMENT SYSTEM FOR INTERNATIONAL SCHOOL

Submitted by

Tien, Nguyen Van

Phuoc, Ha Duc

Huy, Truong Dong

Dat, Nguyen Thanh

Approved by

Chau, Truong Ngoc

Proposal Review Panel Representative:

_____

Name Signature Date

Capstone Project 2- Mentor:

_____

Name Signature Date

PROJECT INFORMATION

Project acronym	Senior Project Management System for International School		
Project Title	SPMS		
Start Date	18 Feb 2022	18 Feb 2022	18 Feb 2022
Lead Institution	International School, Duy Tan University		
Project Mentor	Chau, Truong Ngoc		
Scrum master / Project Leader & contact details	Tien, Nguyen Van Email: cnnguyenvantien@gmail.com Tel: 0704.042.832 Student ID: 24211208536		
Partner Organization			
Project Web URL			
Team members	**Name**	**Name**	**Name**
24211202634	Phuoc, Ha Duc	Phuoc, Ha Duc	Phuoc, Ha Duc
24211206538	Huy, Truong Dong	Huy, Truong Dong	Huy, Truong Dong
24211206470	Dat, Nguyen Thanh	Dat, Nguyen Thanh	Dat, Nguyen Thanh

REVISION HISTORY

Version	Date	Comments	Author	Approval
v1.0	10/05/2021	Initial Release	Phuoc, Ha Duc	x

Table of Contents

SIGNATURE

Document Approvals: *The following signatures are required for approval of this document.*

Chau, Truong Ngoc *Mentor*		Date:
Tien, Nguyen Van *Scrum Master*		Date:
Phuoc, Ha Duc *Product Owner*		Date:
Huy, Truong Dong *Member*		Date:
Dat, Nguyen Thanh *Member*		Date:

1. Introduction

This document aims to set out a standard to programming languages to Senior Project Management System for International School project that includes, HTML/CSS, ReactJS Framework and JavaScript should conform. It is expected that were appropriate changes be made to these standards to adapt them for the language in question.

2. HTML/CSS Coding Standard

2.1. HTML

2.1.1. HTML Style Rules

2.1.1.1. Document Type

- Use HTML5.
- HTML5 (HTML syntax) is preferred for all HTML documents: `<!DOCTYPE html>`.
- (It's recommended to use HTML, as text/html. Do not use XHTML. XHTML, as application/xhtml+xml, lacks both browser and infrastructure support and offers less room for optimization than HTML.)
- Although fine with HTML, do not close void elements, i.e. write
, not
.

2.1.1.2. HTML Validity

- Use valid HTML where possible.
- Use valid HTML code unless that is not possible due to otherwise unattainable performance goals regarding file size.
- Use tools such as the W3C HTML validator to test.
- Using valid HTML is a measurable baseline quality attribute that contributes to learning about technical requirements and constraints, and that ensures proper HTML usage.

```
<!-- Not recommended -->
<title>Test</title>
<article>This is only a test.

<!-- Recommended -->
<!DOCTYPE html>
<meta charset="utf-8">
<title>Test</title>
<article>This is only a test.</article
```

2.1.1.3. Semantics

- Use HTML according to its purpose.
- Use elements (sometimes incorrectly called “tags”) for what they have been created for. For example, use heading elements for headings, p elements for paragraphs, a elements for anchors, etc.
- Using HTML according to its purpose is important for accessibility, reuse, and code efficiency reasons.

```
<!-- Not recommended -->
<div onclick="goToRecommendations();">All recommendations</div>

<!-- Recommended -->
<a href="recommendations/">All recommendations</a>
```

2.1.1.4. Multimedia Fallback

- Provide alternative contents for multimedia.
- For multimedia, such as images, videos, animated objects via canvas, make sure to offer alternative access. For images that means use of meaningful alternative text (alt) and for video and audio transcripts and captions, if available.
- Providing alternative contents is important for accessibility reasons: A blind user has few cues to tell what an image is about without @alt, and other users may have no way of understanding what video or audio contents are about either.
- (For images whose alt attributes would introduce redundancy, and for images whose purpose is purely decorative which you cannot immediately use CSS for, use no alternative text, as in alt="".)

```
<!-- Not recommended -->
<img src="spreadsheet.png">

<!-- Recommended -->
<img src="spreadsheet.png" alt="Spreadsheet screenshot.">
```

2.1.1.5. Separation of Concerns

- Separate structure from presentation from behavior.
- Strictly keep structure (markup), presentation (styling), and behavior (scripting) apart, and try to keep the interaction between the three to an absolute minimum.
- That is, make sure documents and templates contain only HTML and HTML that is solely serving structural purposes. Move everything presentational into style sheets, and everything behavioral into scripts.

- In addition, keep the contact area as small as possible by linking as few style sheets and scripts as possible from documents and templates.
- Separating structure from presentation from behavior is important for maintenance reasons. It is always more expensive to change HTML documents and templates than it is to update style sheets and scripts.

```
<!-- Not recommended -->
<!DOCTYPE html>
<title>HTML sucks</title>
<link rel="stylesheet" href="base.css" media="screen">
<link rel="stylesheet" href="grid.css" media="screen">
<link rel="stylesheet" href="print.css" media="print">
<h1 style="font-size: 1em;">HTML sucks</h1>
<p>I've read about this on a few sites but now I'm sure:
        <u>HTML is stupid!!1</u>
<center>I can't believe there's no way to control the styling of
  my website without doing everything all over again!</center>

<!-- Recommended -->
<!DOCTYPE html>
<title>My first CSS-only redesign</title>
<link rel="stylesheet" href="default.css">
<h1>My first CSS-only redesign</h1>
<p>I've read about this on a few sites but today I'm actually
  doing it: separating concerns and avoiding anything in the HTML of
  my website that is presentational.
<p>It's awesome!
```

2.1.1.6. Entity References

- Do not use entity references.
- There is no need to use entity references like —, ”, or ☺, assuming the same encoding (UTF-8) is used for files and editors as well as among teams.
- The only exceptions apply to characters with special meaning in HTML (like < and &) as well as control or "invisible" characters (like no-break spaces).

```
<!-- Not recommended -->
The currency symbol for the Euro is &amp;ldquo;&amp;eur;&amp;rdquo;.
<!-- Recommended -->
The currency symbol for the Euro is "€".
```

2.1.1.7. Optional Tags

- Omit optional tags (optional).
- For file size optimization and scannability purposes, consider omitting optional tags. The HTML5 specification defines what tags can be omitted.
- (This approach may require a grace period to be established as a wider guideline as it's significantly different from what web developers are typically taught. For consistency and simplicity reasons it's best served omitting all optional tags, not just a selection.)

```
<!-- Not recommended -->
<!DOCTYPE html>
<html>
        <head>
                <title>Spending money, spending bytes</title>
        </head>
        <body>
                <p>Sic.</p>
        </body>
</html>
<!-- Recommended -->
<!DOCTYPE html>
<title>Saving money, saving bytes</title>
<p>Qed.
```

2.1.1.8. Type Attributes

- Omit type attributes for style sheets and scripts.
- Do not use type attributes for style sheets (unless not using CSS) and scripts (unless not using JavaScript).
- Specifying type attributes in these contexts is not necessary as HTML5 implies text/css and text/javascript as defaults. This can be safely done even for older browsers.

```
<!-- Not recommended -->
<link rel="stylesheet" href="https://www.google.com/css/maia.css"
        type="text/css">
<!-- Recommended -->
<link rel="stylesheet" href="https://www.google.com/css/maia.css">

<!-- Not recommended -->
<script src="https://www.google.com/js/gweb/analytics/autotrack.js"
        type="text/javascript"></script>
<!-- Recommended -->
<script src="https://www.google.com/js/gweb/analytics/autotrack.js"></script>
```

2.1.2. HTML Formatting Rules

2.1.2.1. General Formatting

- Use a new line for every block, list, or table element, and indent every such child element.
- Independent of the styling of an element (as CSS allows elements to assume a different role per display property), put every block, list, or table element on a new line.
- Also, indent them if they are child elements of a block, list, or table element.
- (If you run into issues around whitespace between list items it's acceptable to put all li elements in one line. A linter is encouraged to throw a warning instead of an error.)

```
<blockquote>
        <p><em>Space</em>, the final frontier.</p>
</blockquote>
<ul>
        <li>Moe
        <li>Larry
        <li>Curly
</ul>
<table>
         <thead>
                <tr>
                        <th scope="col">Income
                        <th scope="col">Taxes
        <tbody>
                <tr>
                        <td>$ 5.00
                        <td>$ 4.50
</table>
```

2.1.2.2. HTML Line-Wrapping

- Break long lines (optional).
- While there is no column limit recommendation for HTML, you may consider wrapping long lines if it significantly improves readability.
- When line-wrapping, each continuation line should be indented at least 4 additional spaces from the original line.

```
<md-progress-circular md-mode="indeterminate" class="md-accent"
            ng-show="ctrl.loading" md-diameter="35">
</md-progress-circular>
<md-progress-circular
            md-mode="indeterminate"
           class="md-accent"
            ng-show="ctrl.loading"
           md-diameter="35">
```

```
</md-progress-circular>
```

2.1.2.3. HTML Quotation Marks

- When quoting attributes values, use double quotation marks.
- Use double ("") rather than single quotation marks (') around attribute values.

```
<!-- Not recommended -->
<a class='maia-button maia-button-secondary'>Sign in</a>
<!-- Recommended -->
<a class="maia-button maia-button-secondary">Sign in</a>
```

2.2. CSS

2.2.1. CSS Style Rules

2.2.1.1. CSS Validity

- Use valid CSS where possible.
- Unless dealing with CSS validator bugs or requiring proprietary syntax, use valid CSS code.
- Use tools such as the W3C CSS validator to test.
- Using valid CSS is a measurable baseline quality attribute that allows to spot CSS code that may not have any effect and can be removed, and that ensures proper CSS usage.

2.2.1.2. ID and Class Naming

- Use meaningful or generic ID and class names.
- Instead of presentational or cryptic names, always use ID and class names that reflect the purpose of the element in question, or that are otherwise generic.
- Names that are specific and reflect the purpose of the element should be preferred as these are most understandable and the least likely to change.
- Generic names are simply a fallback for elements that have no particular or no meaning different from their siblings. They are typically needed as “helpers.”
- Using functional or generic names reduces the probability of unnecessary document or template changes.

```
/* Not recommended: meaningless */
#yee-1901 {}

/* Not recommended: presentational */
.button-green {}
.clear {}
/* Recommended: specific */
```

```
#gallery {}
#login {}
.video {}

/* Recommended: generic */
.aux {}
.alt {}
```

2.2.1.3. ID and Class Name Style

- Use ID and class names that are as short as possible but as long as necessary.
- Try to convey what an ID or class is about while being as brief as possible.
- Using ID and class names this way contributes to acceptable levels of understandability and code efficiency.

```
/* Not recommended */
#navigation {}
.atr {}
/* Recommended */
#nav {}
.author {}
```

2.2.1.4. Type Selector

- Avoid qualifying ID and class names with type selectors.
- Unless necessary (for example with helper classes), do not use element names in conjunction with IDs or classes.
- Avoiding unnecessary ancestor selectors is useful for performance reasons.

```
/* Not recommended */
ul#example {}
div.error {}
/* Recommended */
#example {}
.error {}
```

2.2.1.5. Shorthand Properties

- Use shorthand properties where possible.
- CSS offers a variety of shorthand properties (like font) that should be used whenever possible, even in cases where only one value is explicitly set.
- Using shorthand properties is useful for code efficiency and understandability.

```
/* Not recommended */
border-top-style: none;
font-family: palatino, georgia, serif;
font-size: 100%;
line-height: 1.6;
padding-bottom: 2em;
```

```
padding-left: 1em;
padding-right: 1em;
padding-top: 0;
/* Recommended */
border-top: 0;
font: 100%/1.6 palatino, georgia, serif;
padding: 0 1em 2em;
```

2.2.1.6. 0 and Units

- Omit unit specification after "0" values, unless required.
- Do not use units after 0 values unless they are required.

```
flex: 0px; /* This flex-basis component requires a unit. */
flex: 1 1 0px; /* Not ambiguous without the unit, but needed in IE11. */
margin: 0;
padding: 0;
```

2.2.1.7. Leading 0s

- Omit leading "0"s in values.
- Do not put 0s in front of values or lengths between -1 and 1.

```
font-size: .8em;
```

2.2.1.8. Hexadecimal Notation

- Use 3 character hexadecimal notation where possible.
- For color values that permit it, 3 character hexadecimal notation is shorter and more succinct.

```
/* Not recommended */
color: #eebbcc;
/* Recommended */
color: #ebc;
```

2.2.1.9. Prefixes

- Prefix selectors with an application-specific prefix (optional).
- In large projects as well as for code that gets embedded in other projects or on external sites use prefixes (as namespaces) for ID and class names. Use short, unique identifiers followed by a dash.
- Using namespaces helps preventing naming conflicts and can make maintenance easier, for example in search and replace operations.

```
.adw-help {} /* AdWords */
#maia-note {} /* Maia */
```

2.2.1.10. ID and Class Name Delimiters

- Separate words in ID and class names by a hyphen.

- Do not concatenate words and abbreviations in selectors by any characters (including none at all) other than hyphens, in order to improve understanding and scannability.

```
/* Not recommended: does not separate the words “demo” and “image” */
.demoimage {}

/* Not recommended: uses underscore instead of hyphen */
.error_status {}
/* Recommended */
#video-id {}
.ads-sample {}
```

2.2.1.11. Hacks

- Avoid user agent detection as well as CSS “hacks”—try a different approach first.
- It’s tempting to address styling differences over user agent detection or special CSS filters, workarounds, and hacks. Both approaches should be considered last resort in order to achieve and maintain an efficient and manageable code base. Put another way, giving detection and hacks a free pass will hurt projects in the long run as projects tend to take the way of least resistance. That is, allowing and making it easy to use detection and hacks means using detection and hacks more frequently—and more frequently is too frequently.

2.2.2. CSS Formatting Rules

2.2.2.1. Declaration Order

- Alphabetize declarations.
- Put declarations in alphabetical order in order to achieve consistent code in a way that is easy to remember and maintain.
- Ignore vendor-specific prefixes for sorting purposes. However, multiple vendor-specific prefixes for a certain CSS property should be kept sorted (e.g. -moz prefix comes before -webkit).

```
background: fuchsia;
border: 1px solid;
-moz-border-radius: 4px;
-webkit-border-radius: 4px;
border-radius: 4px;
color: black;
text-align: center;
text-indent: 2em;
```

2.2.2.2. Block Content Indentation

- Indent all block content.
- Indent all block content, that is rules within rules as well as declarations, so to reflect hierarchy and improve understanding.

```
@media screen, projection {
        html {
                background: #fff;
                 color: #444;
        }
}
```

2.2.2.3. Declaration Stops

- Use a semicolon after every declaration.
- End every declaration with a semicolon for consistency and extensibility reasons.

```
/* Not recommended */
.test {
        display: block;
        height: 100px
}
/* Recommended */
.test {
        display: block;
        height: 100px;
}
```

2.2.2.4. Property Name Stops

- Use a space after a property name's colon.
- Always use a single space between property and value (but no space between property and colon) for consistency reasons.

```
/* Not recommended */
h3 {
        font-weight:bold;
}
/* Recommended */
h3 {
        font-weight: bold;
}
```

2.2.2.5. Declaration Block Separation

- Use a space between the last selector and the declaration block.
- Always use a single space between the last selector and the opening brace that begins the declaration block.
- The opening brace should be on the same line as the last selector in a given rule.

```
/* Not recommended: missing space */
#video{
        margin-top: 1em;
}

/* Not recommended: unnecessary line break */
#video
{
        margin-top: 1em;
}
/* Recommended */
#video {
        margin-top: 1em;
}
```

2.2.2.6. Selector and Declaration Separation

- Separate selectors and declarations by new lines.
- Always start a new line for each selector and declaration.

```
/* Not recommended */
a:focus, a:active {
        position: relative; top: 1px;
}
/* Recommended */
h1,
h2,
h3 {
        font-weight: normal;
        line-height: 1.2;
}
```

2.2.2.7. Rule Separation

- Separate rules by new lines.
- Always put a blank line (two line breaks) between rules.

```
html {
        background: #fff;
}

body {
        margin: auto;
```

```
        width: 50%;
}
```

2.2.2.8. CSS Quotation Marks

- Use single (') rather than double ("") quotation marks for attribute selectors and property values.
- Do not use quotation marks in URI values (url()).
- Exception: If you do need to use the @charset rule, use double quotation marks—single quotation marks are not permitted.

```
/* Not recommended */
@import url("https://www.google.com/css/maia.css");

html {
        font-family: "open sans", arial, sans-serif;
}
/* Recommended */
@import url(https://www.google.com/css/maia.css);

html {
        font-family: 'open sans', arial, sans-serif;
}
```

2.2.3. CSS Meta Rules

2.2.3.1. Section Comments

- Group sections by a section comment (optional).
- If possible, group style sheet sections together by using comments. Separate sections with new lines.

```
/* Header */

#adw-header {}

/* Footer */

#adw-footer {}

/* Gallery */

.adw-gallery {}
```

3. JavaScript Coding Standard

3.1. General JavaScript Guidelines

3.1.1. Use expanded syntax

For JavaScript we use expanded syntax, with each line of JS on a new line, the opening brace of a block on the same line as its associated statement, and the closing brace on a new line. This maximizes readability, and again, promotes consistency on MDN.

```
/* Recommended */
function myFunc() {
        console.log('Hello!');
};

/* Not recommended */
function myFunc() { console.log('Hello!'); };
```

In addition, keep these specifics in mind:

- Don't include padding spaces after opening brackets or before closing brackets — (myVar), not (myVar).
- All statements must end with semicolons (";"). We require them in all of our code samples even though they're technically optional in JavaScript because we feel that it leads to code that is clearer and more precise about where each statement ends.
- Use single quotes in JavaScript, wherever single quotes are needed in syntax.
- There should be no space between a control statement keyword, function, or loop keyword and its opening parenthesis (e.g. if() { ... }, function myFunc() { ... }, for(...) { ... }).
- There should be a space between the parentheses and the opening curly brace in such cases as described in the previous bullet.

3.1.2. JavaScript comments

- Use JS-style comments to comment code that isn't self-documenting:

```
function myFunc() {
        // Output the string 'Hello' to the browser's JS console
        console.log('Hello');
         // Create a new paragraph, fill it with content, and append it to the <body>
        let para = document.createElement('p');
        para.textContent = 'My new paragraph';
        document.body.appendChild(para);
}
```

- Also note that you should leave a space between the slashes and the comment, in each case.

3.1.3. Use modern JS features

For general usage, you can use modern well-supported JS features (such as arrow functions, promises, async/await, let/const, template literals, and spread syntax) in MDN examples. We include them in many places in these guidelines, as we believe the web industry has generally gotten to the point where such features are familiar enough to be understandable. And for those that don't use them yet, we'd like to play our part in helping people to evolve their skills.

3.2. Variable

3.2.1. Variable naming

For variable names use lowerCamelCasing, and use concise, human-readable, semantic names where appropriate.

```
/* Recommended */
let playerScore = 0;

let speed = distance / time;

/* Not recommended */
let thisIsaveryLONGVariableThatRecordsPlayerscore345654 = 0;

let s = d/t;
```

3.2.2. Declaring variables

- When declaring variables and constants, use the let and const keywords, not var.
- If a variable will not be reassigned, prefer const:

```
const myName = 'Chris';
console.log(myName);
```

- Otherwise, use let:

```
let myAge = '40';
myAge++;
console.log('Happy birthday!');
```

3.3. Operators and comparison

3.3.1. Ternary operators

- Ternary operators should be put on a single line:

```
let status = (age >= 18) ? 'adult' : 'minor';
```

- Not nested:

```
let status = (age >= 18)
  ? 'adult'
  : 'minor';
```

- This is much harder to read.

3.3.2. Use strict equality

- Always use strict equality and inequality.

```
/* Recommended */
name === 'Chris';
age !== 25;

/* Not recommended */
name == 'Chris';
age != 25;
```

3.3.3. Use shortcuts for boolean tests

Use shortcuts for boolean tests — use x and !x, not x === true and x === false.

3.4. Control statements

```
/* Recommended */
if(iceCream) {
        alert('Woo hoo!');
}

/* Not recommended */
if (iceCream){
        alert('Woo hoo!');
}
```

Also bear in mind:

- There should be no space between a control statement keyword and its opening parenthesis.
- There should be a space between the parentheses and the opening curly brace.

3.5. Strings

3.5.1. Use template literals

For inserting values into strings, use string literals.

```
/* Recommended */
let myName = 'Chris';
console.log(`Hi! I'm ${myName}!`);

/* Not recommended */
let myName = 'Chris';
console.log('Hi! I\'m' + myName + '!');
```

3.5.2. Use textContent, not innerHTML

When inserting strings into DOM nodes, use Node.textContent:

```
let text = 'Hello to all you good people';
const para = document.createElement('p');
para.textContent = text;
```

Not Element.innerHTML:

```
let text = 'Hello to all you good people';
const para = document.createElement('p');
para.innerHTML = text;
```

textContent is a lot more efficient, and less error-prone than innerHTML.

3.6. Conditionals

3.6.1. General purpose looping

- When loops are required, feel free to choose an appropriate loop out of the available ones (for, for...of, while, etc.) Just make sure to keep the code as understandable as possible.
- When using for/for...of loops, make sure to define the initializer properly, with a let keyword:

```
/* Recommended */
let cats = ['Athena', 'Luna'];
for(let i of cats) {
        console.log(i);
}

/* Not recommended */
let cats = ['Athena', 'Luna'];
for(i of cats) {
        console.log(i);
}
```

- Also bear in mind:

- There should be no space between a loop keyword and its opening parenthesis.
- There should be a space between the parentheses and the opening curly brace.

3.6.2. Switch statements

Format switch statements like this:

```
let expr = 'Papayas';
switch(expr) {
	case 'Oranges':
		console.log('Oranges are $0.59 a pound.');
		break;
	case 'Papayas':
		console.log('Mangoes and papayas are $2.79 a pound.');
		// expected output: "Mangoes and papayas are $2.79 a pound."
		 break;
	default:
		console.log(`Sorry, we are out of ${expr}`);
}
```

3.7. Functions and objects

3.7.1. Function naming

- For function names use lowerCamelCasing, and use concise, human-readable, semantic names where appropriate.

```
/* Recommended */
function sayHello() {
	alert('Hello!');
};

/* Not recommended */
function SayHello() {
	alert('Hello!');
};

function notVeryObviousName() {
	alert('Hello!');
};
```

3.7.2. Defining functions

- Where possible, use the function declaration to define functions over function expressions:

```
/* Recommended */
function sum(a, b) {
  return a + b;
}

 /* Not recommended */
```

```
let sum = function(a, b) {
  return a + b;
}
```

- When using anonymous functions inside a method that requires a function as a parameter, it is acceptable (although not required) to use an arrow function to make the code shorter and cleaner.
- So instead of this:

```
const array1 = [1, 2, 3, 4];
let sum = array1.reduce(function(a, b) {
  return a + b;
});
```

- you could write this:

```
const array1 = [1, 2, 3, 4];
let sum = array1.reduce((a, b) =>
  a + b
);
```

- Also bear in mind:

- There should be no space between a function name and its opening parenthesis.
- There should be a space between the parentheses and the opening curly brace.

3.7.3. Creating objects

Use literals — not constructors — for creating general objects (i.e., when classes are not involved):

```
/* Recommended */
let myObject = { };

/* Not recommended */
let myObject = new Object();
```

3.7.4. Object classes

- Use ES class syntax for objects, not old-style constructors.

```
class Person {
        constructor(name, age, gender) {
                this.name = name;
                this.age = age;
                this.gender = gender;
        }

        greeting() {
                console.log(`Hi! I'm ${this.name}`);
        };
}
```

- Use extends for inheritance:

```
class Teacher extends Person {
  ...
}
```

3.7.5. Object naming

- When defining an object class (as seen above), use UpperCamelCasing (also known as PascalCasing) for the class name, and lowerCamelCasing for the object property and method names.

- When defining an object instance, either a literal or via a constructor, use lowerCamelCase for the instance name:

```
let hanSolo = new Person('Han Solo', 25, 'male');

let hanSolo = {
            name: 'Han Solo',
            age: 25,
            gender: 'male'
}
```

3.8. Arrays

3.8.1. Creating arrays

- Use literals — not constructors — for creating arrays:

```
/* Recommended */
let myArray = [ ];

/* Not recommended */
let myArray = new Array(length);
```

3.8.2. Adding to an array

- When adding items to an array, use push(), not direct assignment. Given the following array:

```
const pets = [];

/* Recommended */
pets.push('cat');

/* Not recommended */
pets[pets.length] = 'cat';
```

3.9. Error handling

- If certain states of your program throw uncaught errors, they will halt execution and potentially reduce the usefulness of the example. You should therefore catch errors using a try...catch block:

```
try {
  console.log(results);
}
catch(e) {
  console.error(e);
}
```

4. ReactJS Coding Standard

4.1. Basic Rules

- Only include one React component per file.
- Always use JSX syntax.
- Do not use React.createElement unless you're initializing the app from a file that is not JSX.

4.2. Naming

- File- and component name need to be identical.
- Use PascalCase naming convention for filename as well as component name, e.g. GlobalHeader.js

```
// Bad
// Filename: foo.js

class Foo extends React.Component {}

export default Foo;


// Good
// Filename: Foo.js

class Foo extends React.Component {}

export default Foo;
```

4.3. Ordering

- Ordering for class extends React.Component:

- constructor
- optional static methods
- getChildContext
- componentWillMount
- componentDidMount
- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- componentDidUpdate
- componentWillUnmount
- clickHandlers or eventHandlers like onClickSubmit() or onChangeDescription()
- getter methods for render like getSelectReason() or getFooterContent()
- Optional render methods like renderNavigation() or renderProfilePicture()
- render

- How to define propTypes, defaultProps, contextTypes, etc...

```
import React, { Component, PropTypes } from 'react';

const propTypes = {
    id: PropTypes.number.isRequired,
    url: PropTypes.string.isRequired,
    text: PropTypes.string,
};

const defaultProps = {
    text: 'Hello World',
};

export default class Link extends Component {
    static methodsAreOk() {
        return true;
    }

    render() {
        return <a href={this.props.url} data-
id={this.props.id}>{this.props.text}</a>
    }
}
```

```
Link.propTypes = propTypes;
Link.defaultProps = defaultProps;
```

4.4. Alignment

Follow these alignment styles for JSX syntax

```
// bad
<Foo superLongParam="bar"
         anotherSuperLongParam="baz" />

// good
<Foo
      superLongParam="bar"
      anotherSuperLongParam="baz"
/>

// if props fit in one line then keep it on the same line
<Foo bar="bar" />

// children get indented normally
<Foo
         superLongParam="bar"
         anotherSuperLongParam="baz"
>
         <Spazz />
</Foo>
```

4.5. Quotes

Always use double quotes (") for JSX attributes, but single quotes for all other JS.

```
// bad
<Foo bar='bar' />

// good
<Foo bar="bar" />

// bad
<Foo style={{ left: "20px" }} />

// good
<Foo style={{ left: '20px' }} />

## Props
- Always use camelCase for prop names.

```javascript
// bad
<Foo
    UserName="hello"
    phone_number={12345678}
/>
```
```

```
// good
<Foo
    userName="hello"
   phoneNumber={12345678}
/>
```

4.6. Tags

- Always self-close tags that have no children.

```
// bad
<Foo classNam="stuff"></Foo>

// good
<Foo classNam="stuff" />
```

- If your component has multi-line properties, close its tag on a new line.

```
// bad
<Foo
    bar="bar"
    baz="baz" />

// good
<Foo
   bar="bar"
    baz="baz"
/>
```

4.7. Stateless function components

For stateless components use the function syntax, introduced in React 0.14.

```
// Using an ES2015 (ES6) arrow function:
var Aquarium = (props) => {
    var fish = getFish(props.species);
    return <Tank>{fish}</Tank>;
};

// Or with destructuring and an implicit return, simply:
var Aquarium = ({species}) => (
    <Tank>
        {getFish(species)}
    </Tank>
);

// Then use: <Aquarium species="rainbowfish" />
```

4.8. PropTypes Declaration

- Setting propTypes declarations is mandatory
- Group them into required/none-required
- Alphabetically sort each group
- Separate them by a new line

```
static propTypes = {
    blank: React.PropTypes.bool.isRequired,
    block: React.PropTypes.bool.isRequired,
    size: React.PropTypes.string.isRequired,
    to: React.PropTypes.string.isRequired,
    disabled: React.PropTypes.bool,
};
```

4.9. Prefixing none React methods

Prefix all none React methods within a component with an underscore.

```
class Foo extends React.Component {

   componentDidMount() {
       this._update();
    }

   _update() {
        // e.g. update position
    }

    render() {
       return (
            <div>foo</div>
        );
    }
}
```

4.10. Prefixing component wide variables

In the exception that you do not want to place a component wide variables on the state, you have to prefix it with an underscore.

```
class Foo extends React.Component {

    componentDidMount() {
        this._el = React.FindDOMNode(this.refs.foo);
    }

    render() {
        return (
            <div>foo</div>
        );
    }
```

```
}
```

4.11. Using handler methods

- Name methods using '_handle' + triggering event, e.g. _handleClick
- Bind handler using the ES6 arrow syntax, so inside the callback it has always the right context

```
class Foo extends React.Component {

    _handleClick = (e) => {
        this.setState(
           {
                clicked: true
            }
        );
    }

    render() {
        return (
            <button
onClick={this._handleClick}>Submit</button>
        );
   }
}
```

4.12. Using “container” components for loading data from Stores

```
// CommentListContainer.js

class CommentListContainer extends React.Component {
    constructor() {
        super();
        this.state = { comments: [] }
    }
    componentDidMount() {
        $.ajax({
            url: "/my-comments.json",
            dataType: 'json',
            success: function(comments) {
            this.setState({comments: comments});
            }.bind(this)
        });
    }
    render() {
        return <CommentList comments={this.state.comments} />;
    }
}



// CommentList.js
```

```
class CommentList extends React.Component {
    constructor(props) {
        super(props);
    }
    _renderComment({body, author}) {
        return <li>{body}—{author}</li>;
    }
    render() {
        return <ul> {this.props.comments.map(_renderComment)}
</ul>;
    }
}
```

4.13. Closing Components without children

```
render() {
    return (
        <Foo>
            <Bar />
        </Foo>
    );
}
```

4.14. List iterations

When rendering a list of components from an array, do it inline if it makes sense. If the map function is too long or complicated, consider extracting it out into its own method on the component class.

```
render() {
    return (
        <ul>
            {this.state.fooList.map(fooItem =>
<FooItem>{fooItem}</FooItem>)}
        </ul>
    );
}
```

4.15. Formatting Attributes

```
<input
    type="text"
    value={this.state.foo}
    onChange={this._handleInputChange.bind(this, 'foo')}
/>
```

4.16. Inline CSS styles

Static properties should be set in the SCSS, dynamic ones in JS.

```
.Foo {
    background-color: #ff0;
}
class Foo extends React.Component {

    render() {

        const styles = {
            'transform': 'translateX(' + this.state.position +
' + px)'
        };

        return (
            <div className="Foo" styles={classes}>Foo
Header</div>
        )
    };

}
```

4.17. Use "classnames" to set CSS classes

Use the classnames node module for setting CSS classes on an element.

```
import React from 'react';
import classnames from 'classnames';

class Foo extends React.Component {

    render() {

        const classes = classnames('FooHeader', {
            'is-fixed': this.state.fixed,
            'is-visible': this.state.visible
        });

        return (
            <div className={classes}>Foo Header</div>
        )
    };

}
```

5. Reference

[1] https://google.github.io/styleguide/htmlcssguide.html.

[2] https://developer.mozilla.org/en-US/docs/MDN/Guidelines/Code_guidelines/JavaScript#general_javascript_guidelines

[3] https://developer.wordpress.org/coding-standards/wordpress-coding-standards/javascript/.

[4] https://github.com/pillarstudio/standards/blob/master/reactjs-guidelines.md.

[5] https://medium.com/m/global-identity?redirectUrl=https%3A%2F%2Fbetterprogramming.pub%2F21-best-practices-for-a-clean-react-project-df788a682fb.