WORKING ISN'T GOOD ENOUGH

I see a lot of code that works but is of low quality

Working code is not the end of programming

Working code is the beginning of programming

You need to write for:

- Easy to change
- ...repeatedly
- Easy to understand
- ...with minimal effort

SEPARATION OF CONCERNS

A very common problem is lack of SOC

- Functions too "coupled" to the rest of the code

Changing some part (such as HTML)

- Should not require change everywhere

HTML should have nothing to do with calling a service

ALL coding expects SOC, not just JS, not just web dev

EXAMPLE HTML FOR POOR JS CODING

Sample HTML for a simple TODO list

```
<ul class="tasks">
  <li class="task">Do INFO6250 work ONTIME</li>
</ul>

<div class="to-add">
  <input name="taskName" class="task-to-add">
  <button class="add-task">
</div>
```

EXAMPLE OF POOR SERVICE CALL, PART 1

```
const addButton = document.querySelector('.add-task');
addButton.addEventListener('click', (e) => {
  e.preventDefault();
  const taskText = document.querySelector('.task-to-add');
  addButton.innerText = "...";
  addButton.disabled = true;
  fetch('/tasks', {
    method: 'POST',
    headers: new Headers({
      'content-type': 'application/json'
    }),
    body: JSON.stringify({ text: taskText });
  })
  //...more stuff
```

WHAT NEEDS TO HAPPEN?

- Attach an event listener
- Indicate call in-progress (spinner)
- ...here "..." text and disabled
- Read in data from form/input fields
- Send call
- Handle errors, OR
- Read results
- ...This will be a full list of tasks
- ...including the new one
- Update list of tasks
- Clear the form/input fields

FIRST PROBLEM

That's a lot!

...So do not try to do it all in one function!

FIXING THAT ISSUE (BUT NOT EVERYTHING)

```
const addButton = document.querySelector('.add-task');
addButton.addEventListener('click', (e) => {
  e.preventDefault();
  adjustButton(addButton);
  const formData = gatherFormInfo();
  addTask(formData, addButton);
});
```

This is more readable, but doesn't FIX the problems

- we pass `addButton` to `addTask()` to reset the button text/state
- But `addTask()` still coupled to the HTML
    - Still has to set the list
    - Has to report errors

BETTER SEPARATION

```
const addButton = document.querySelector('.add-task');
const taskList = document.querySelector('.tasks');

addButton.addEventListener('click', (e) => {
  e.preventDefault();
  const origText = setSpin({button: addButton, spin: true});
  const formData = gatherFormInfo();
  addTask(formData)
  .then( taskList => {
    refreshList(taskList);
    resetNewTaskInput();
  })
  //...
});
```

WHY/HOW IS THIS BETTER?

The real change is not here, it is inside `addTask`

- `addTask()` no longer touches ANY html
- It is given data, returns data
- Errors are rejected as data
- Caller can decide how to react to this data
- Can be reused for different purposes!
- Does not change if the HTML changes!

That is the "Separation" in "Separation of Concerns"

HERE'S THE REST OF CALLING ADDTASK

Notice `.fetch()` is **inside** `addTask()`

```
  addTask(formData)
  .then( taskList => {
    refreshList(taskList);
    resetNewTaskInput();
  })
  .catch( err => {
    reportError(err);
  })
  .then( () => {
    setSpin({
      button: addButton, text: origText, spin: false
    });
  });
});
```

But using **results** are **outside** `addTask()`

DETAILS

Some things required an extra step

- "spinner" was done before fetch and after .catch()

Most parts got easier!

- Doing less means fewer things to worry about!

And it all makes more sense

- All changes to HTML in the event handler

You want to minimize "side-effects"

- Code is more reusable
- Know what functions do without looking at code

A WELL-WRITTEN SERVICE CALL

- sends/gets data

That's all

That involves translating data (incl errors)

- Not reading data from HTML
- Not displaying data
- Not displaying errors

Promises make it easy to attach behaviors

- **IF** you return the promise!

SAMPLE ADDTASK

Notice we **return** the promise

- We don't add any behavior except data parsing/translation

```
function addTask( { taskText } ) {
  return fetch( '/tasks',
    method: 'POST',
    headers: new Headers({
      'content-type': 'application/json'
    }),
    body: JSON.stringify({ text: taskText });
  })
  // ...the rest
};
```

PARSING THE RESPONSE

There is not ONE way

```
//...the fetch call
.catch( err => Promise.reject('Network issues'))
.then( response => {
  if(response.ok) {
    return response.json();
  }
  return Promise.reject(response.statusCode);
})
// ...returned to caller
```

Here our errors are unstructured

- Always good to provide structure