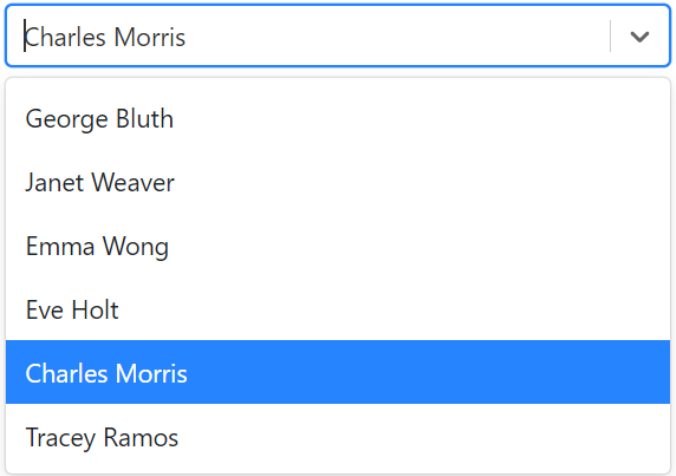
# CS385 Lecture 15

## Drop down lists, deactivating buttons, textbox validation, regular expressions

# Drop down lists (select lists) are very useful UI components

- They allow for the control of user input
- They allow you to constrain the set of values a user can select.
- Most users are familiar with these lists
- Can help reduce the amount of code required to clean and check user input



# Drop down lists also generate an event object when they are changed

- This is very similiar to text boxes – **when the user selects from the drop down list an event object is created**.
- This also creates an `onChange` event which must be handled by writing a function to deal with the consequences of this event.
- Let's look at an example – we'll use Bootstrap to style our drop-down list as part of a `form`

```
3   const [myChoice, setMyChoice] = useState([null]);
4
5   let desserts = [
6     {dessert: "Tiramisu" },{dessert: "Mousse" },{dessert: "Macarons" },
7     {dessert: "Ice-cream" },{dessert: "Cupcake" },{dessert: "Apple Pie" }];
8
9   function handleListChange(event) {
10    setMyChoice(event.target.value);
11  }
12  return (
13    <>
14      <div class="container-fluid">
15        <h1>CS385 Desserts</h1>
16        <form>
17          <label for="mydessertlist" class="form-label">Pick dessert</label>
18          <select onChange={handleListChange}
19            class="form-control" id="mydessertlist">
20
21            {desserts.map((sweet, key) => (
22              <option key={key} value={sweet.dessert}>
23               <strong> {sweet.dessert}</strong>
24              </option>
25            ))}
26
27          </select>
28        </form>
29        {myChoice && <p>Your choice: {myChoice}</p>}
30      </div>
```

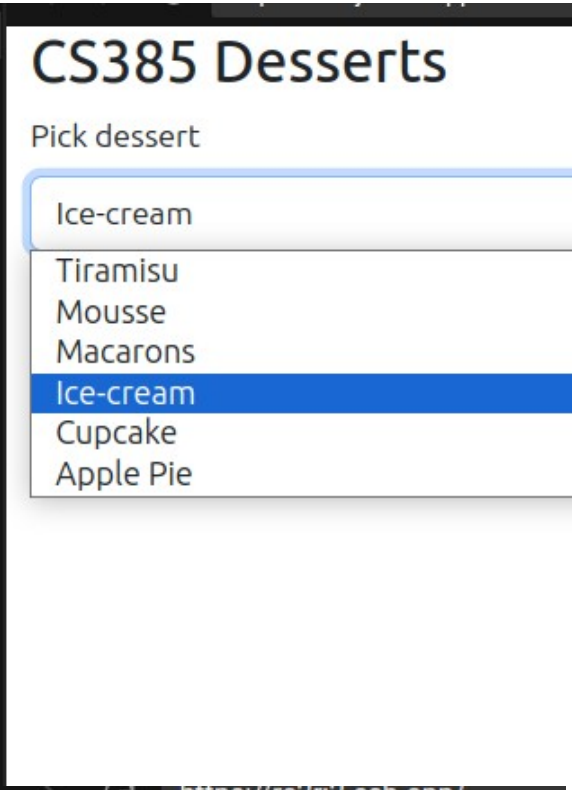
Line 18-19 This **<select>** is your list

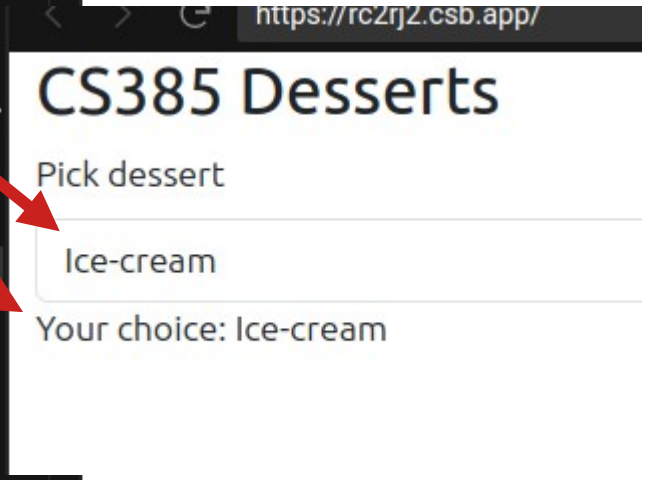
Boostrap is used via ‘form-control’

We also use the <label> tag so bootstrap correctly places the label for the list on the UI

Line 21-25 A map function is used to create the values for each **<option>** in the **<select>** list.
The **desserts** array is used

```
const [myChoice, setMyChoice] = useState(null);

let desserts = [
  {dessert: "Tiramisu" },{dessert: "Mousse" },{dessert: "Macarons" },
  {dessert: "Ice-cream" },{dessert: "Cupcake" },{dessert: "Apple Pie" }];

function handleListChange(event) {
  setMyChoice(event.target.value);
}
return (
  <>
    <div class="container-fluid">
      <h1>CS385 Desserts</h1>
      <form>
        <label for="mydessertlist" class="form-label">Pick dessert</label>
        <select onChange={handleListChange}
          class="form-control" id="mydessertlist">

          {desserts.map((sweet, key) => (
            <option key={key} value={sweet.dessert}>
             <strong> {sweet.dessert}</strong>
            </option>
          ))}

        </select>
      </form>
      {myChoice && <p>Your choice: {myChoice}</p>}
    </div>
```

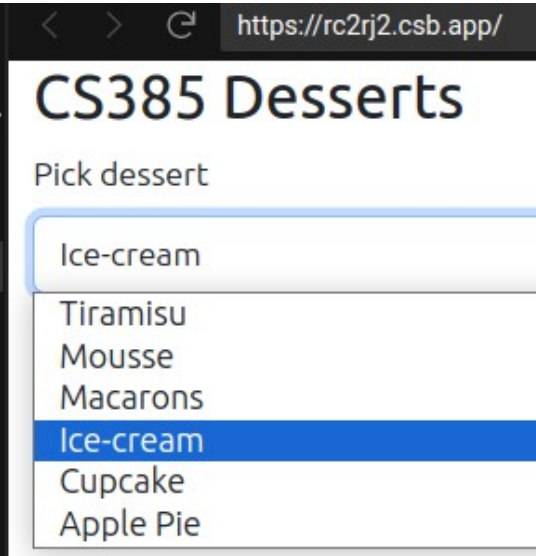


# Sorting your drop-down list is good practice – helps user navigation

- With our approach – using a map function – the order of options is the same as our array

```
function App() {
  const [myChoice, setMyChoice] = useState(null);

  let desserts = [
    {dessert: "Tiramisu" },{dessert: "Mousse" },{dessert: "Macarons" },
    {dessert: "Ice-cream" },{dessert: "Cupcake" },{dessert: "Apple Pie" }];

  function handleListChange(event) {
    setMyChoice(event.target.value);
  }
  return (
    <>
```




- We can easily **add a sorting comparator** to ensure that our data is presented to the user in some logical sorted order

# The sorted list options is much more user friendly

```
let desserts = [
  {dessert: "Tiramisu" },{dessert: "Mousse" },{dessert: "Macarons" },
  {dessert: "Ice-cream" },{dessert: "Cupcake" },{dessert: "Apple Pie" }];

  function sortDesserts(dx, dy) {
    let DX = dx.dessert.toUpperCase();
    let DY = dy.dessert.toUpperCase();
    if (DX > DY) return 1;
    // alphabetical order
    else if (DX < DY) return -1;
    else return 0;
  }
function handleListChange(event) {
  setMyChoice(event.target.value);
}
return (
  <>
    <div class="container-fluid">
      <h1>CS385 Desserts</h1>
      <form>
        <label for="mydessertlist" class="form-label">Pick dessert</label>
        <select onChange={handleListChange}
          class="form-control" id="mydessertlist">
          {desserts.sort(sortDesserts).map((sweet, key) => (
            <option key={key} value={sweet.dessert}>
             <strong> {sweet.dessert}</strong>
            </option>
          ))}
```

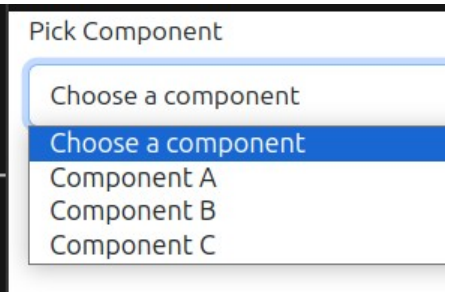

**NOTE:** As discussed previously, it is application dependent what sorting order you impose.
You may need to sort data coming from an API source

# Drop down lists can also be useful for management of component selection

- In our organic shop example (up to now) we have used buttons to control the selection of various components.
- **Drop down lists offer an alternative way to manage the display or hiding of component based on the user's selection.**
- Let's look at a very simple example

# Drop Down lists – component control example

```
  );
}

function ComponentA() {
  return (
    <>
      <h1>Hi, I'm Component A</h1>
    </>
  );
}
function ComponentB() {
  return (
    <>
      <h1>Hi, I'm Component B</h1>
    </>
  );
}
function ComponentC() {
  return (
    <>
      <h1>Hi, I'm Component C</h1>
    </>
  );
}

export default App;
```




- Let's declare 3 very simple components – remember these can be fully featured components if necessary
- Let's use the drop down list to switch between the display of the three components (all child components within App parent)

# Drop Down lists – component control example

```
function App() {
  const [myChoice, setMyChoice] = useState(null);

  function handleListChange(event) {
    setMyChoice(event.target.value);
  }
  return (
    <>
      <div class="container-fluid">
        <form>
          <label for="mycomplist" class="form-label">Pick Component</label>

          <select onChange={handleListChange} class="form-control"  id="mycomplist" >
            <option key="0" selected>Choose a component</option>
            <option key="A" value="ComponentA">Component A</option>
            <option key="B" value="ComponentB">Component B</option>
            <option key="C" value="ComponentC">Component C</option>
          </select>
        </form>
        {!myChoice && <h1>Choose a component please</h1>}
        {myChoice === "ComponentA" && <ComponentA />}
        {myChoice === "ComponentB" && <ComponentB />}
        {myChoice === "ComponentC" && <ComponentC />}
      </div>
    </>
  );
}
```

Pick Component

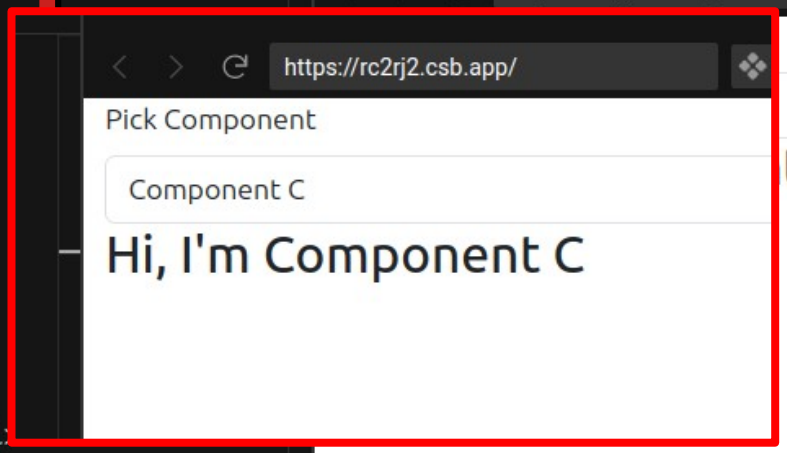
Choose a component

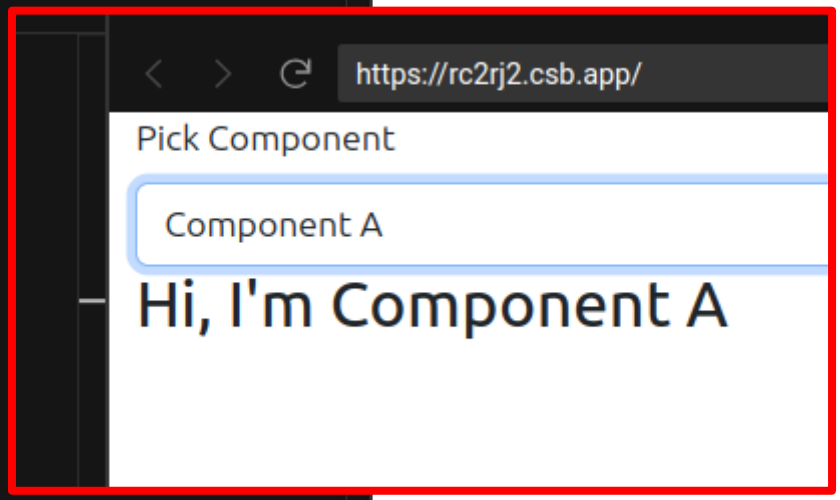
Choose a component

- We create the **<option>** items manually
- Note that the first option is **selected** but has no value (**null**)
- Then we use **conditional rendering** to display the component based on the value stored in the state variable **myChoice**

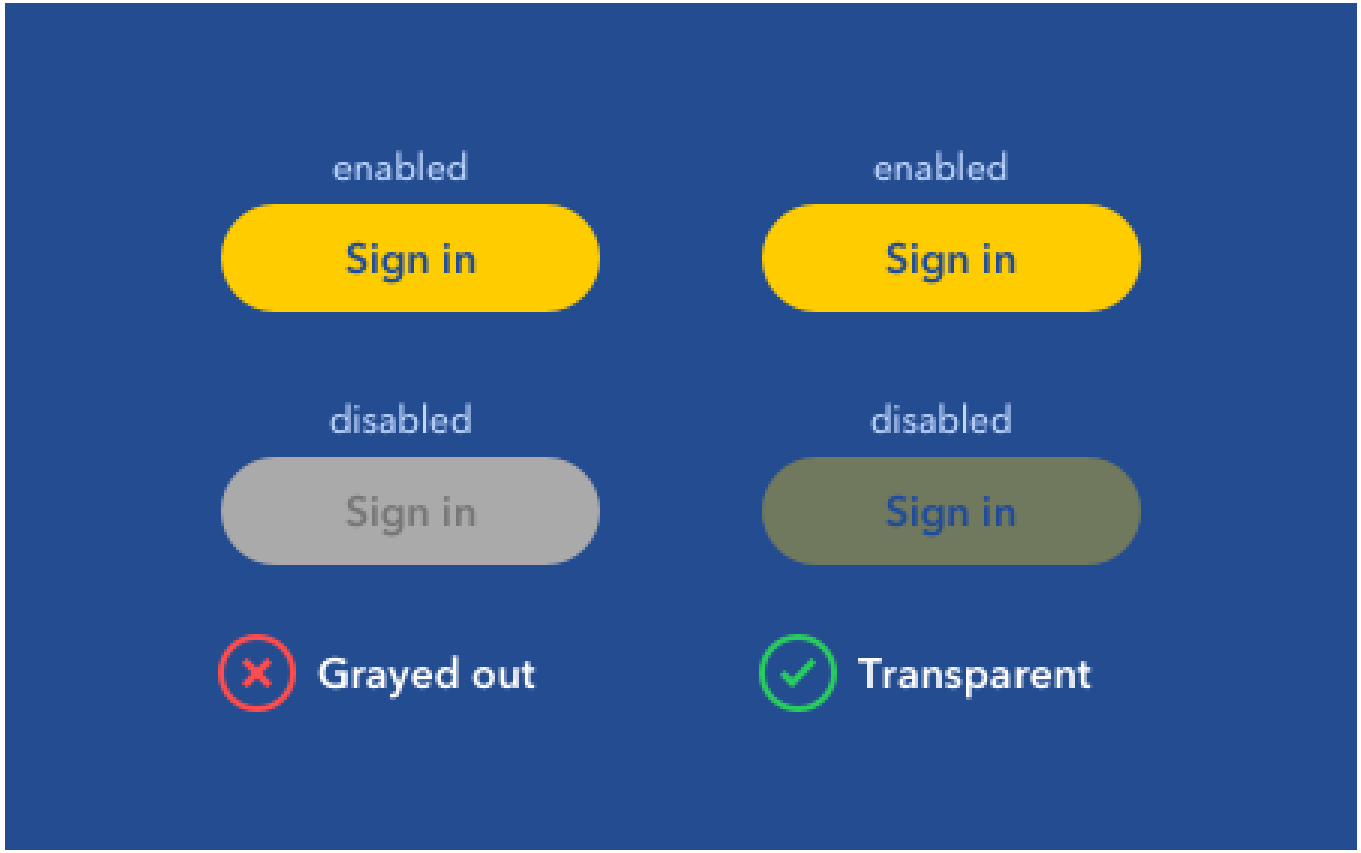
# Drop Down lists – component control example

```
function App() {
  const [myChoice, setMyChoice] = useState(null);

  function handleListChange(event) {
    setMyChoice(event.target.value);
  }
  return (
    <>
      <div class="container-fluid">
        <form>
          <label for="mycomplist" class="form-label">Pick Component</label

          <select onChange={handleListChange} class="form-control"  id="mycomplist" >
            <option key="0" selected>Choose a component</option>
            <option key="A" value="ComponentA">Component A</option>
            <option key="B" value="ComponentB">Component B</option>
            <option key="C" value="ComponentC">Component C</option>
          </select>
        </form>
        {!myChoice && <h1>Choose a component please</h1>}
        {myChoice === "ComponentA" && <ComponentA />}
        {myChoice === "ComponentB" && <ComponentB />}
        {myChoice === "ComponentC" && <ComponentC />}
      </div>
    </>
  );
}
```



# Activating and Deactivating buttons on the User Interface



# In certain situations, we need to disable a button from usage

- This could be required if a user has not provided certain data via a textbox
- It could be that a user has not completed some set of steps or actions.
- **Every HTML button has a property called `disabled` which takes a boolean value. It is defaulted to `false`.**
- We can use a variable in our Javascript to control or 'toggle' the corresponding button as **disabled**

# Example – deactivate a button when `clicks = 10`

```
2  function App() {
3    const [clicks, setClicks] = useState(0);
4    const [active, setActive] = useState(true);
5
6    function handleButtonClick() {
7      if (clicks < 10) setClicks(clicks + 1);
8      else setActive(false);
9    }
10   function handleButtonReset() {
11     setClicks(0);
12     setActive(true);
13   }
14
15   return (
16     <>
17       <div class="container-fluid">
18         <h1>Activating and Deactivating Buttons</h1>
19         <button disabled={!active}
20           type="button" class="btn btn-primary btn-lg"
21           onClick={handleButtonClick}>Next+</button>
22
23         <button
24           type="button" class="btn btn-warning btn-lg"
25           onClick={handleButtonReset}>Reset</button>
26       </div>
27       <h1>Current value of clicks is {clicks}</h1>
28     </>
```

Activating and Deactivating Buttons

Next+ Reset

Current value of clicks is 10

- This example allows us to deactive the **Next+** button when the value of **clicks = 10**. The reset button returns **clicks** to zero.
- NOTICE Line 19 – the use of the reserved button property `disabled`.
- This has a **boolean value** – we control it using the `active` boolean value from state (line 4)

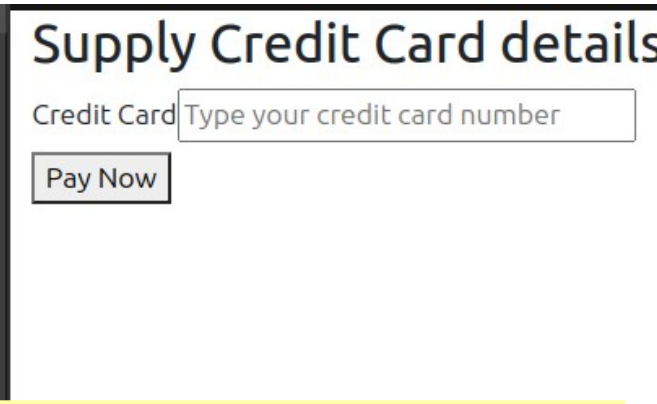
# Form or input validation

# We need to validate user input – because users cannot be trusted

- Users will never actually do what you ask them to do using your UI.
- All applications, where OPEN TEXT fields are used will need to perform **VALIDATION** or **DATA CLEANING** or **FORM VALIDATION** on all user inputs.
- Let's see a very simple example of this where we ask the user for their credit card number.

# Form validation example – let's setup our state variables and validation checks

```
function App() {
  const [credit, setCredit] = useState("");
  const [error, setError] = useState(null);
  // when the user presses the submit button.
  function handleSubmit(e) {
    //used to stop the default behavior of an HTML form from taking place
    e.preventDefault();
    // Regular expression to check for credit card numbers (simplified)
    const cardPattern = /^[0-9]{10,12}$/;
    // test the regular expression
    if (cardPattern.test(credit) === false) {
      setError(true);
    } else {
      setError(false);
    }
  }
  // when the user is typing in their credit card number
  function handleCreditCardInput(e) {
    setCredit(e.target.value);
  }
```




- We have a 'submit' button or "Pay Now" which when the user presses this button, we need to check if their credit card is a valid set of numerical values (we don't check if the number exists here)
- We use **a simple regular expression** to look for 10 -12 digits only.
- We set an **`error`** variable based on the results of this **`test`** from the regular expression

# Form validation example – let's setup our HTML form (with an onSubmit event)

```
23
24    return (
25      <div class="container-fluid">
26        <form onSubmit={handleSubmit} autocomplete="off">
27          <h1>Supply Credit Card details</h1>
28          <div className="formInput">
29            <label for="ccard" class="form-label">Credit Card</label>
30            <input id="ccard" type="text" name="credit" value={credit}
31              placeholder="Type your credit card number"
32              onChange={handleCreditCardInput}/>
33          </div>
34          <button>Pay Now</button>
35
36          {error === true && <h1>ErrorCredit card numbers must be 10 -12 digits
37          {error === false && (
38            <>
39              <h1>Your credit card number is valid</h1>{" "}
40              <h2>Your card is <mark>{credit}</mark></h2>
41            </>
42          )}
43        </form>
```

Supply Credit Card details

Credit Card Type your credit card number

Pay Now

- **Conditional rendering** is used on line 36 and 37 to control the supply of error messages to the user.

- Here we have a **`<form>`** with an **`onSubmit`** event. So, React assumes that the **`<button>`** is the submit button.
- The **`handleSubmit`** function will only be invoked when the button is pressed.
- We have, as before, our **`onChange`** event from the input box.

# Form validation – example output

App.js

```
22    }
23
24    return (
25      <div class="container-fluid">
26        <form onSubmit={handleSubmit} autocomplete="off">
27          <h1>Supply Credit Card details</h1>
28          <div className="formInput">
29            <label for="ccard" class="form-label">Credit Card</label>
30            <input id="ccard" type="text" name="credit" value={credit}
31              placeholder="Type your credit card number"
32              onChange={handleCreditCardInput}/>
33          </div>
34          <button>Pay Now</button>
35
36          {error === true && <h1>Error: Credit card numbers must be 10 -12 digits</h1>}
37          {error === false && (
38            <>
39              <h1>Your credit card number is valid</h1>{" "}
40              <h2>Your card is <mark>{credit}</mark></h2>
41            </>
42          )}
43        </form>
44      </div>
45    );
46  }
47
48  export default App;
```

Browser Tests Terminal

https://rc2rj2.csb.app/

Supply Credit Card details

Credit Card 123456789

Pay Now

Error: Credit card numbers must be 10 -12 digits

https://rc2rj2.csb.app/

Supply Credit Card details

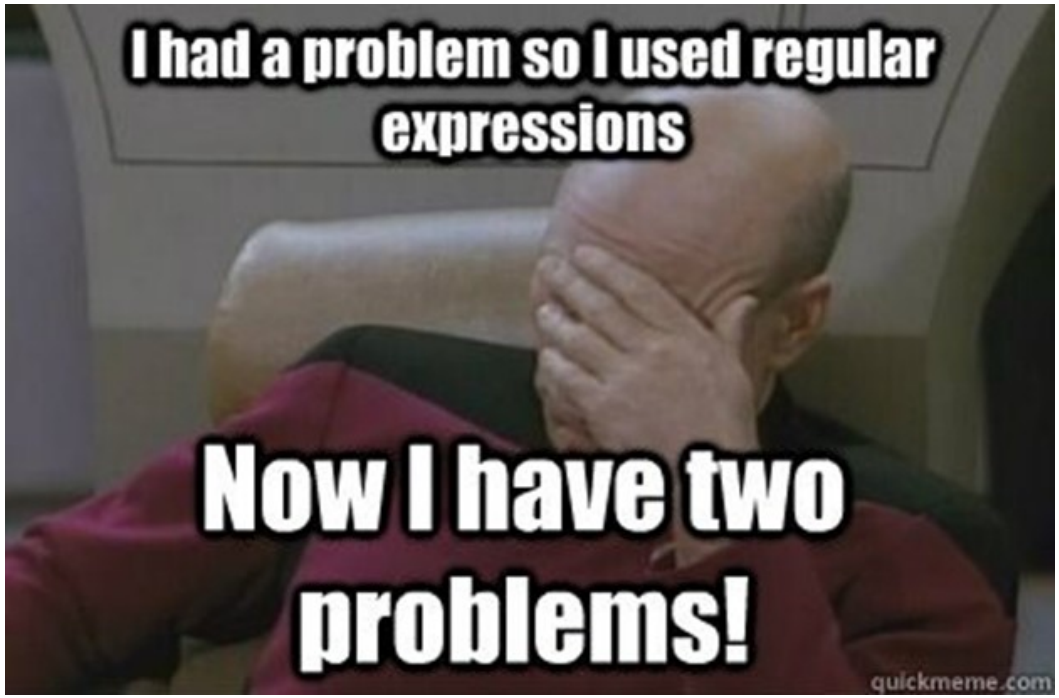
Credit Card 98765432101

Pay Now

Your credit card number is valid

Your card is 98765432101

# Testing your regular expressions

# It can be very useful to write a little test harness for regex

```
function App() {
  function testRegularExpression(testStr) {
    // Regular expression to check MU email addresses
    // 5 or more characters followed by at least one digit
    const strPattern = /^[A-Za-z]{5,}\d{1,}@mu.ie$/;

    if (strPattern.test(testStr)) {
      return "Valid string: " + testStr;
    } else {
      return "Invalid string: " + testStr;
    }
  }

  return (
    <>
      <h1>{testRegularExpression("peter12@mu.ie")}</h1>
      <h1>{testRegularExpression("cs385@mu.ie")}</h1>
    </>
  );
}
```

Valid string: peter12@mu.ie

Invalid string: cs385@mu.ie

# Very quick guide to regex [1]

- `/^.*$/` – `^` is the start of the string, `$` is the end of the string.
- The quantifier `.*` matches ANY character or digit (including punctuation)
- **\d** – matches a digit 0,1,2,3,4,5,6,7,8,9
- **\w** – matches an alphabetical character or digit
- **\s** – matches a space character

# Very quick guide to regex [2]

- **Quantifiers** are ways to match one or more characters
- `A{2}` will match two upper case A characters
- `M{3,5}` will match between 3 and 5 upper case M characters
- `P{4,}` will match at least 4 (or more) upper case P characters

# Very quick guide to regex [3]

- **Character classes** are very useful ways to express multiple characters
- `[a-z]{3}` will match 3 lower case characters a to z inclusive.
- `[P-Z]{2}` will match 2 upper case characters P to Z inclusive
- `[0-9]{4,}` will match at least 4 or more digit characters

# Very quick guide to regex [4]

- Conditional matching
- You can use the `|` character
- `(a|e|i|o|u){2,}` will match two or more lowercase vowel characters

# Example – MU email addresses for student email.

- STEPHEN.JONES.2024@mumail.ie
- FATIMA.KELLY.2024@mumail.ie
- Upper case letters – a period – upper case letters a period – 4 digits – ampersand – lowercase letters – period – ie

# Example – MU email addresses for student email.

- STEPHEN.JONES.2024@mumail.ie
- FATIMA.KELLY.2024@mumail.ie

```
function App() {
  function testRegularExpression(testStr) {
    // Regular expression to check MU email addresses

    const strPattern = /^[A-Z]{1,}\.[A-Z]{1,}\.\d{4}@mumail.ie$/;

    if (strPattern.test(testStr)) {
      return "Valid string: " + testStr;
    } else {
      return "Invalid string: " + testStr;
    }
  }

  return (
    <>
      <h1>{testRegularExpression("STEPHEN.JONES.2024@mumail.ie")}</h1>
      <h1>{testRegularExpression("FATIMA.KELLY.2024@mumail.ie")}</h1>
      <h1>{testRegularExpression("PETER.MOONEY@mumail.ie")}</h1>
    </>
  );
}
```

Valid string: STEPHEN.JONES.2024@mumail.ie

Valid string: FATIMA.KELLY.2024@mumail.ie

Invalid string: PETER.MOONEY@mumail.ie

# We shall have one regex question in the Lab Exam 2

```
function App() {
  function testRegularExpression(testStr) {
    // Regular expression to check credit card numbers

    const strPattern = /^\d{4}-\d{4}-\d{4}-\d{4}$/;

    if (strPattern.test(testStr)) {
      return "Valid string: " + testStr;
    } else {
      return "Invalid string: " + testStr;
    }
  }

  return (
    <>
      <h1>{testRegularExpression("4561-1234-2345-8790")}</h1>
      <h1>{testRegularExpression("4561_1234_2345_8790")}</h1>
      <h1>{testRegularExpression("4561_1234_2345_87901")}</h1>
    </>
  );
}
```

Valid string: 4561-1234-2345-8790

Invalid string: 4561_1234_2345_8790

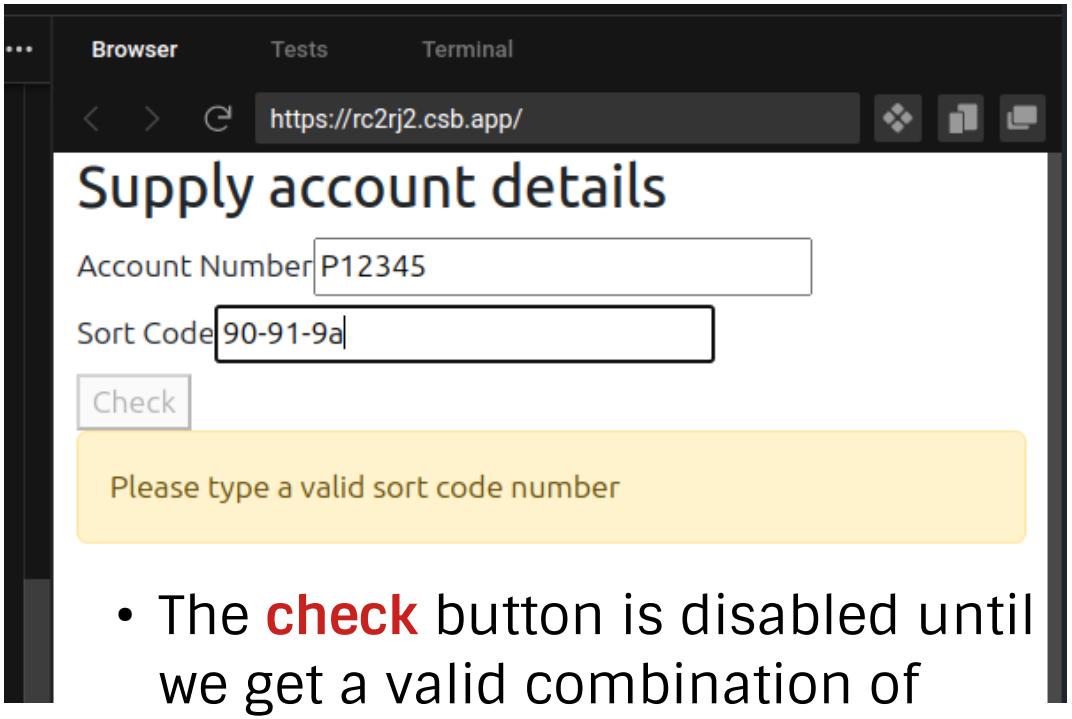
Invalid string: 4561_1234_2345_87901

# Validating in real-time

- The previous example for validation seen us wait until the user had pressed the Submit Button before validation was undertaken.
- **Let's look at a different example where we validate the user input – as they type.**
- This approach is more recognisable to what we experience in apps and online forms.

# Example – a bank account details checker

- **Assumptions** – **account numbers** have structure UpperCase letter then 5 digits
- **Sort codes** have 3 pairs of digits separated by hyphens.
- **For simplicity we don't check these against a database of accounts**




- The **check** button is disabled until we get a valid combination of **account number** and **sort code**.
- **We don't have any action associated with the Check button (for simplicity)**

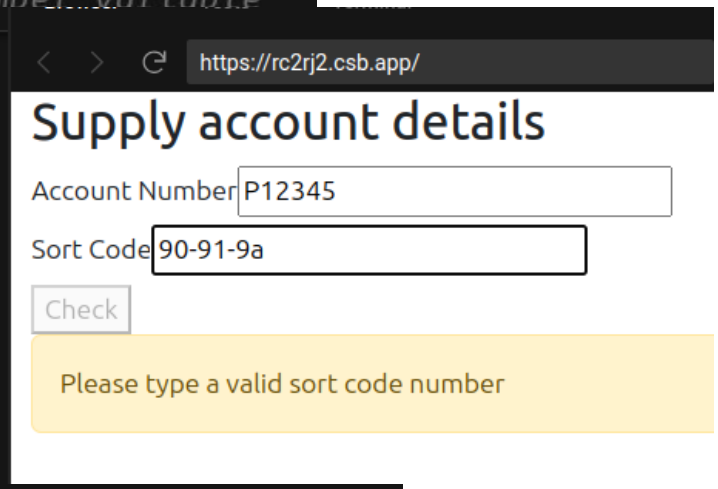
# Starting off – we'll declare our state variables

- We shall have **an error variable for both text boxes**. This is a simple approach but much easier to maintain the overall logic when you have a lot of conditions or text boxes.

```
JS App.js
import React, { useState } from "react";

function App() {
  const [account, setAccount] = useState("");
  const [sortCode, setSortCode] = useState("");
  const [errorAcc, setErrorAcc] = useState(true);
  const [errorSort, setErrorSort] = useState(true);
```

# Next – let's declare our checks for both account and sort code

```
9   //account numbers must have capital letter than 5 digits
10  function handleAccountInput(e) {
11    setAccount(e.target.value); // update the account number variable
12    // check the pattern
13    const accPattern = /^[A-Z]{1}\d{5}$/;
14    if (accPattern.test(e.target.value)) {
15      setErrorAcc(false);
16    } else {
17      setErrorAcc(true);
18    }
19  }
```

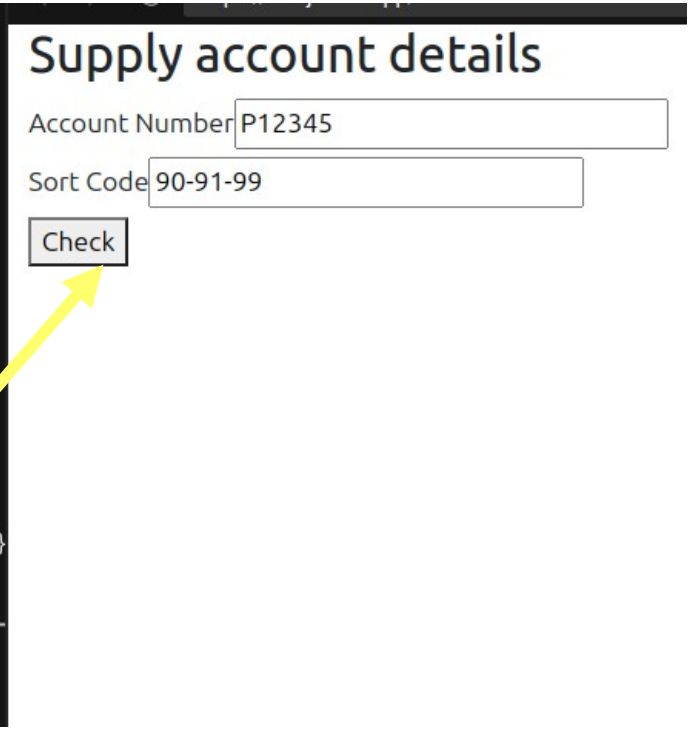



```
21    // sort code must be digits only - in pairs
22    // separated by hyphens. For example, 14-56-17
23    function handleSortCodeInput(e) {
24      setSortCode(e.target.value); // update the sort code variable
25      // check the pattern
26      const sortPattern = /^\d{2}-\d{2}-\d{2}$/;
27      if (sortPattern.test(e.target.value)) {
28        setErrorSort(false);
29      } else {
30        setErrorSort(true);
31      }
32    }
```

# Our input form – with bootstrap stylying

- **Important** – notice line 49 – there is an OR condition for **`errorAcc OR errorSort`** – we need both to be false so **`disabled = false`**

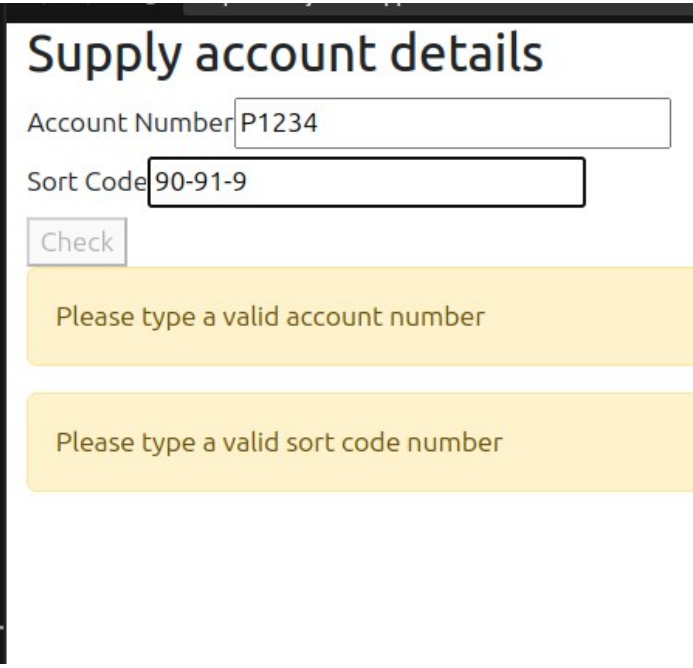
```
33
34    return (
35      <div class="container-fluid">
36        <form>
37          <h1>Supply account details</h1>
38          <div class="formInput">
39            <label for="account" class="form-label">Account Number</label>
40            <input id="account" type="text" name="account" value={account}
41              placeholder="Type your account number here"
42              onChange={handleAccountInput}/>
43            <br/>
44            <label for="sortCode" class="form-label">Sort Code</label>
45            <input id="sortCode" type="text" name="sortCode" value={sortCode}
46              placeholder="Type your sort code here" onChange={handleSortCodeInput}
47          </div>
48
49          <button disabled={errorAcc || errorSort}>Check</button>
50        </form>
```



# Finally, our error messages for each of the input boxes

- We use conditional rendering with both **`errorAcc`** and **`errorSort`**

```
48
49        <button disabled={errorAcc || errorSort}>Check</button>
50      </form>
51
52      {errorAcc && (
53        <><div class="alert alert-warning" role="alert">
54            Please type a valid account number
55          </div></>)}
56
57      {errorSort && (
58        <><div class="alert alert-warning" role="alert">
59            Please type a valid sort code number
60          </div></>)}
61    </div>
62  );
63 }
```



# This example shows validation "in real time"

- For brevity, we don't "do anything" with the two inputs. For the scope of the example we don't need this.
- However, you could use form validation like this for many different input scenarios.
- **REMEMBER – always validate input – and ensure that you have checks for empty strings**

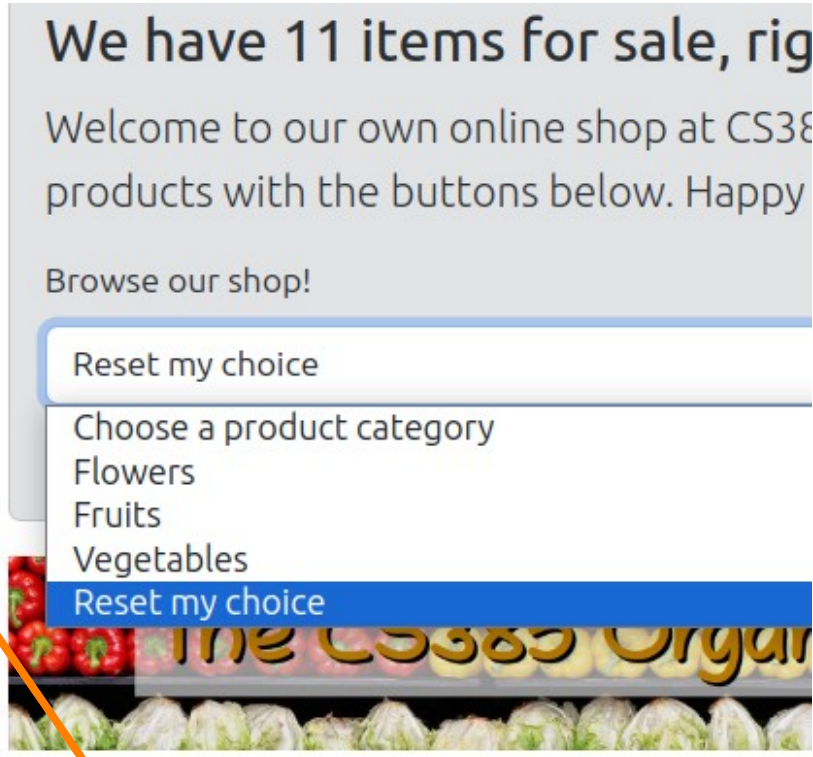
# Back to our Organic Shop

- We could include some button activation/deactivation around the checkout or basket functionality.
- **A drop-down-list is an option to include**

```
rn (

<div class="container-fluid">
  <img src={logoBanner} class="img-fluid" alt="CS385 branding" />
  <div class="alert alert-secondary" role="alert">
    <h2>We have {inventory.length} items for sale, right now!</h2>
    <p class="lead">
      Welcome to our own online shop at CS385. You can browse our
      products with the buttons below. Happy Shopping!{" "}
    </p>
    <form>
      <label for="productlist" class="form-label">Browse our shop!</label>

      <select class="form-control" id="productlist">
        <option key="0" selected>Choose a product category</option>
        <option key="A" value="Flowers">Flowers</option>
        <option key="B" value="Fruit">Fruit</option>
        <option key="C" value="Vegetables">Vegetables</option>
        <option key="C">Reset my choice</option></select>
    </form>
```


# We have to include our onChange handler

```
<select
  onChange={changeProductCategoryFromDropDown}
  class="form-control"
  id="productlist"
>
  <option key="0" selected value="Reset">
    Choose a product category
  </option>
  <option key="A" value="Flowers">
    Flowers
  </option>
  <option key="B" value="Fruits">
    Fruits
  </option>
  <option key="C" value="Vegetables">
    Vegetables
  </option>
  <option key="C" value="Reset">
    Reset my choice
  </option>
```




```
// Allow for switching between different product categories
function changeProductCategoryFromDropDown(e) {
  if (e.target.value === "Reset")
  setProductChoice(null);
  else
  setProductChoice(e.target.value);
}
```

# Next we decide – do we keep the buttons or the drop down list?

```
JS App.js ×

138        products with the buttons below. Happy Shopping!{" "}
139      </p>
140      <form>
141        <label for="productlist" class="form-label">
142          Browse our shop!
143        </label>
144
145        <select
146          onChange={changeProductCategoryFromDropDown}
147          class="form-control"
148          id="productlist"
149        >
150          <option key="0" selected value="Reset">
151            Choose a product category
152          </option>
153          <option key="A" value="Flowers">
154            Flowers
155          </option>
156          <option key="B" value="Fruits">
157            Fruits
158          </option>
159          <option key="C" value="Vegetables">
160            Vegetables
161          </option>
162          <option key="C" value="Reset">
163            Reset my choice
164          </option>
165        </select>
166      </form>
167       
168      {basket.length > 0 && (
```

Browser Tests Terminal

https://fxkqlt.csb.app/


## We have 11 items for sale, right now!

Welcome to our own online shop at CS385. You can browse our products with the buttons below. Happy Shopping!

Browse our shop!

Flowers

## Our Flowers products (4 items)

| | | |
|---|---|---|
| Daffodil Bulbs (100) | €20.00 | Add to basket |
| Dahlia Bulbs (10) | €10.00 | Add to basket |
| David Austin Rose | €20.00 | Add to basket |
| Royal Red Tulips | €25.00 | Add to basket |