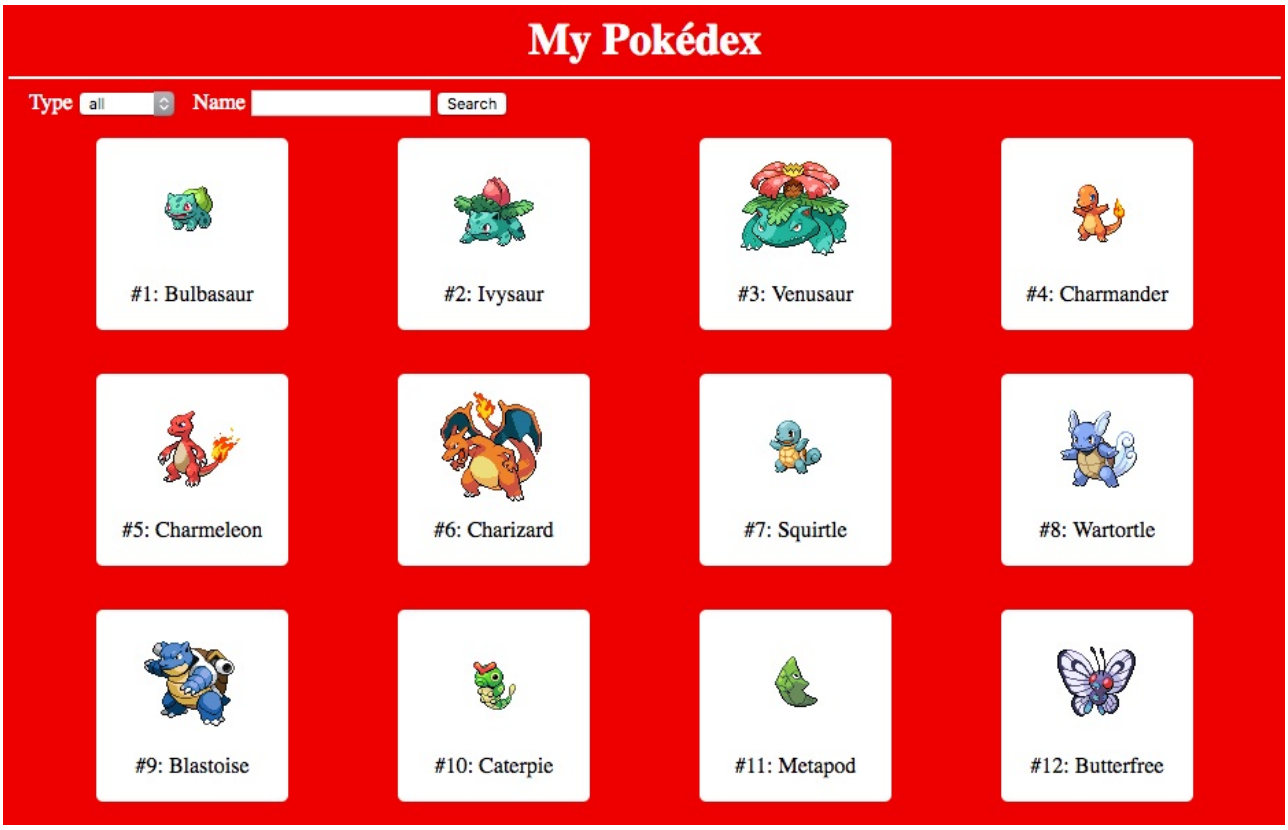
Table of Contents

1 In these Sushi Cards you'll be building an app to get information from an **Application Programming Interface (API)** and display it to your user. Then you'll let them filter the information too. In the case of these Sushi Cards you'll be using the PokeAPI. It will let you get information about, and pictures of, Pokémon to display to your users. Once you're done, it'll look like this:



2 Since these are the Advanced JavaScript cards, you'll find the explanations are a bit less detailed than the earlier series, but you should be familiar with the basic pieces of JavaScript used in this program from the earlier series. I'll be talking more about what the code does, and why, than exactly *how* it manages it.

3 Go to dojo.soy/trinket and click "Sign Up For Your Free Account" if you do not already have an account. You will need an email address to sign up.

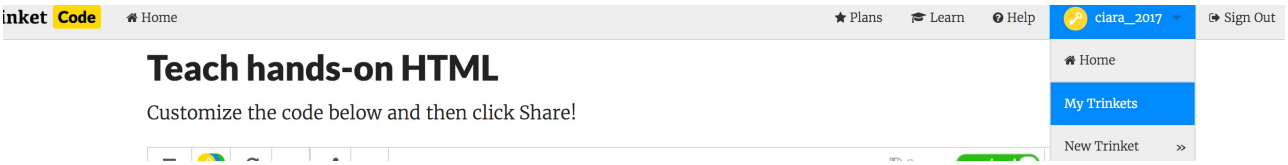
4 Enter your email address and choose a password, or ask somebody to do this for you.

5 Creating an account allows you to save your work and access it from any computer. It also allows you to make a copy of a project somebody else has shared with you so you can make your own changes to it!

6 Go to dojo.soy/js-a-start. You will see a box containing an example JavaScript website project. On the right hand side is the website, and on the left hand side is the code that makes the website. If you are not signed in, you will need to enter your email address and password to be able to **Remix** the project.

7 Click the "Remix" button at the top right of the project (if it is not green, you have to sign in and then click it again). This creates a copy of the project for you to work with. It should say "remixed" after you click it

8 Next to the "Sign Out" button at the very top right corner of the page you should see your username and a drop-down menu (the tiny triangle tells you there is a drop-down). Click on it to show the menu and then select "My Trinkets".



In Trinket (this website), projects are called "Trinkets"

9 The project you just remixed will be shown together with some example projects for other programing languages. It will be called "Advanced JavaScript Remix". Click on it to begin editing!

10 You should also click on the drop-down arrow beside the “Autorun” button and select “Click To Run” instead. This makes sure that the code will only run when you click this button, not when you're halfway through typing it!

1 If you're going to display information about Pokémon, you're going to need some to display! Luckily, some nice people have put together an API you can use to ask for that information. There are APIs for everything from Google to Instagram to YouTube to the weather! That's where you're going to start: Asking for the information. Before you can even do that, though, you're going to need somewhere to put it once you've got it! In `script.js` create three empty arrays, like this:

```
var pokemonInfo = []
var pokePics = []
var pokemon
```

2 Next, you need to create a function that can `fetch` info about a Pokémon from PokeAPI. You do this by requesting a **URL** (web address) and storing the **response**. In the case of PokeAPI, the **URL** is `https://pokeapi.co/api/v2/pokemon/[id]/`, where `[id]` is the id number of the Pokémon you want. Pikachu, for example, is number 25. Here's a function that does this:

```
async function fetchPokemon(pokemonId) {
 var response = await fetch('https://pokeapi.co/api/v2/pokemon/'+pokemonId.toString()+'/')
 var poke = await response.json()
 pokemonInfo.push(poke)
}
```

This function looks relatively normal, taking `pokemonId` as a parameter, fetching the **URL**, taking the response and `push`-ing it into `pokemonInfo`. However, look at a couple of unusual things here:

- Instead of `function`, this is declared as `async function`
- Twice, JavaScript is told to `await` something

These two **keywords** always come as a pair: Only an `async` function can contain `await`. What `await` means is that, instead of sending off the `fetch` and forgetting about the response back from PokeAPI, JavaScript should first `await` the `fetch` being completed and then `await` the `response` being retrieved as `json`. This means the code doesn't get ahead of itself and try to use the information before PokeAPI has sent it back! There will be more about JSON on the next card, so we're going to skip over it for now.

3 Now that you can `fetch` and store one Pokémon, it's time to get a bunch of them (gotta catch 'em all!). As you might guess, this involves a `for` loop! Add another function in to call your previous one, like this:

```
async function fetchManyPokemon(pokemonCount) {
 for(var i = 1; i <= pokemonCount; i++){
   await fetchPokemon(i)
 }
}
```

Notice that, because you're fetching Pokémon by their IDs, which start from `1`, your `for` loop needs to start from `1` instead of the usual `0`.

4 Next, you need to get the image for the Pokémon, download it into JavaScript (this bit is unusual, but you'll see why later!) and store it in an array. The function to get a single Pokémon image, given the information about it that you've stored in `pokemonInfo` looks like this:

```
async function fetchPokemonImage(pokemonInfo) {
 var imageUrl = pokemonInfo.sprites.front_default

 var imageResponse = await fetch(imageUrl)
 var image = await imageResponse.blob()

 var base64 = await getBase64(image)

 pokePics.push(base64)
}
```

What this does is get the **URL** the image is living at (PokeAPI gave you that as part of the response, earlier), the, just like `fetchPokemon`, it will `fetch` the response (see that it's `await`-ing it again!) and then take the `blob` of the response. Image files, like the `.png` files you're getting here, are `blob` files. What this means, basically, is that they're not text files.

5 Now that you can `fetch` one image, you want to do the same for every Pokémon you've got. Just like before, this is a `for` loop that calls your `function`, but a normal one since you're looping over an array. Notice that it's using `await` again, since there's a `fetch` inside the function it's calling. Also, notice that the **parameter** passed to `fetchPokemonImage` is one value from `pokemonInfo`, selected based on the value of `i` and that you can do all of that in one line to save yourself creating a **variable** you'd only use once.

```
async function fetchPokemonImages() {
 for(var i = 0; i < pokemonInfo.length; i++){
   await fetchPokemonImage(pokemonInfo[i])
 }
}
```

6 Now, wrap all the fetching functions you've written up into one called `getPokemon`, like this:

```
async function getPokemon(pokemonCount){
   pokemon = []
   await fetchManyPokemon(pokemonCount)
   await fetchPokemonImages()
}
```

7 Finally, for this card, you need to create a `function` that will call your other functions and display some output so you can check everything worked. Since your other functions are `async`, this one has to be too, and has to `await` them. Add this at the very bottom of your file and then run it:

```
async function buildDex(pokemonCount){
 await getPokemon(pokemonCount)


 var display =  document.getElementById("display")
 display.innerHTML = ""
 var image = document.createElement("img")
 image.src = pokePics[0]
 display.appendChild(image)
}


buildDex(1)
```

If everything worked there should be a bit of a wait while the information gets loaded and then you'll see a picture of a Pokémon!

1

As proimsed, this card will look at **JSON**. You won't be writing any actual code here, just learning about this very important part of **JavaScript**. If you've already done the Intermediate or even Beginner JavaScript Sushi, you've worked with JSON before and even created some, I just didn't tell you what it was called! So let's look at some from the Intermediate cards first:

```
[
   {
     text: "My",
     completed: false
   },
   {
     text: "to-do",
     completed: true
   },
   {
     text: "list",
     completed: true
   }
 ]
```

JSON stands for **JavaScript Object Notation** and it's basically just bits of **JavaScript** used to send information. There you've got an **array** with some **objects** in it. Each **object** has a few **variables** with values. If you wanted to look up the `text` of the first one, assuming this was stored in a **variable** named `toDos` you could do so with `toDos[0].text`. You could look up the `completed` value of the second one with `toDos[1].completed`.

2

You use **JSON** whenever you want to store information that is somehow connected. For example, if you have a list of information about a sports team, you might want to store things like:

- Their name, as a text variable
- Which league they play in, probably as a **JSON object** (you can **nest** them inside each other) with the name of the league and the **URL** where you can find even more **JSON** with the full info on the league, like a list (**array**) of all the teams in it. If the team is in more than one league, say a football team in the Premiership and the Champion's League, then you'd have an **array** of those league **objects**.
- An array of player **objects**, including things like their name, age, position, etc.
- A fixtures **array**, with the dates and times of their upcoming games and the name and **URL** (to the same kind of JSON object) for the team they're playing against
- Maybe an **array** of history: which games and leagues they won and when

You can see that this can quickly turn into a large set of files, linked to each other by **URLs** that can tell you, or your program, a lot about a subject. You could use this imaginary **JSON API** to make an app that showed you what matches were coming up this weekend and how those teams had fared when they played in the past. Or one that ranked all the teams in the league by things like wins/losses, score, number of players, etc. Notice that you're getting a lot more information than you're using in those apps, but that's normal with **JSON** that some other service is providing: They don't know exactly what kind of website or app you're building, so they give you *lots* of info!

3 Now it's time to take a look at the **JSON** PokeAPI is sending you. Check out dojo.soy/bulbasaur. I've pasted the **JSON** for Bulbasaur (Pokémon #1) into this online **JSON** viewer to make things a bit more readable for you. On the left, you can see the 9,081 lines that make up the Bulbasaur **JSON**. On the right, you can see that as a tree you can click into, which a human being has a better chance of understanding! What's important to learn here, really, is that you don't need to understand all of the **JSON** to use some of it. I have not read all of Bulbasaur's file and I never will, I've written a program that uses it, and you're in the process of doing the same! Since you'll be storing this info in your program, technically, if you get 150 pokemon in there, you'll have over a million lines of code in your program!

4 On the next card, you're going to create a simpler **JSON** object for a Pokémon—a handful of lines that you can use to store and retrieve the information you really need.

1 Now you're going to make your own **JSON** object. It's going to be a simple version of a Pokémon. What you're going to need to build a basic Pokédex is listed below. I figured out this list by looking at the **JSON** of one Pokémon using the tool you saw on the last card.

- The Pokémon's id number
- Name of the Pokémon
- A picture of it
- Its “types” (fire, grass, water, etc.)

The Pokémon will be a kind of **object**, just like the ToDo was in the Intermediate JavaScript Sushi Cards. You can use a `function` to create and `return` an **object**, which lets you keep all the code related to that object neatly organised together. Add this function to your code to do so:

```
function Pokemon(pokemonIndex) {
 var info = pokemonInfo[pokemonIndex]


 this.id = info.id
 this.name = info.name
 this.image = pokePics[pokemonIndex]


}
```

The code takes in the index (number in an array) of a Pokémon which it looks up and stores in the `info` **variable**. The `this` **keyword** is used to refer to **variables** attached to the specific Pokémon **object** you're creating. Every Pokémon will have an `id`, `name`, and `image`, so you need to refer to the specific ones attached to *this* Pokémon. What the code does here is take the `name` and `id` from the **JSON** about the Pokémon and the `image` from the **array** (`PokePics`) of images.

2 The next part is a bit trickier: A Pokémon can have more than one type, so you need to create a `types` **array** to store them, and then add them using a `for` **loop** that looks through the types in `info.types` and adds their names into `this.types`. You need to update the `function` you just wrote to add the code for types, like this:

```
function Pokemon(pokemonIndex) {
 var info = pokemonInfo[pokemonIndex]


 this.id = info.id
 this.name = info.name
 this.image = pokePics[pokemonIndex]


 this.types = []


 for(var i = 0; i < info.types.length; i++){
   var type = info.types[i].type.name
   this.types.push(type)
 }


}
```

3 Now that you've got a `function` to make a Pokémon, you need another one that will make all the Pokémon, by calling this `function` as many times as you have Pokémon. As you can probably guess, that's another `for` loop! Aren't `for` loops great?

```
function makePokemonList(pokemonCount) {
 for(var i = 0; i < pokemonCount; i++){
   pokemon.push(new Pokemon(i))
 }
}
```

Notice how the `new` **keyword** is used to create a `Pokemon` object, using your `Pokemon` `function`. That `new` is telling **JavaScript** that the `Pokemon` function is making an **object**, rather than just doing series of instructions, so it will treat it a little differently. This is why the `Pokemon` **object** gets stored in the `pokemon` **array**, rather than a reference to the `Pokemon` `function`.

4 Add a call to `makePokemonList` to `getPokemon`, passing the number of Pokémon you've fetched as the value of `pokemonCount`, like this:

```
async function getPokemon(pokemonCount){
    pokemon = []
    await fetchManyPokemon(pokemonCount)
    await fetchPokemonImages()
    await makePokemonList(pokemonCount)
}
```

1 You can now make your own Pokémon objects, containing all the information you think is important about a Pokémon. However, you might have noticed that it takes a few seconds for the page to load every time it refreshes. To really get an idea of this, change the call to `buildDex` so it's got a larger value for `pokemonCount` passed to it. Try `buildDex(10)` and see how long that takes. It's because the code has to go off and fetch the info from PokeAPI every time the page loads. Wouldn't it be nice to store that information so you didn't have to wait? You can do that! You can use **JavaScript** to store the info and pictures on the user's computer so that once they've downloaded them the first time they don't have to wait for them again! You'll do this using a part of **JavaScript** called `localStorage`.

2 What you want the code to do here is:

- Check if there are already saved Pokémon and if there are enough of them to match `pokemonCount` (more is fine!)
- If yes:
  - Great! Use those Pokémon!
- If no:
  - Fetch the Pokémon info
  - Fetch the Pokémon pictures
  - Make the Pokémon **objects**
  - Save them, so you can use them next time

3 A lot of that looks like the code in `getPokemon` right now. In fact, this is basically what you want `getPokemon` to do: give you Pokemon. It should figure out how to get them and send them back, no matter where it found them. Let's start by wrapping the existing code in an `if` statement that tests for saved Pokémon (we'll worry about the count and saving parts in the next few steps):

```
async function getPokemon(pokemonCount){
 if(localStorage.getItem("pokemon") === null) {
   pokemon = []
   await fetchManyPokemon(pokemonCount)
   await fetchPokemonImages()
   await makePokemonList(pokemonCount)
   // save the pokemon list
 }
 else {
   // get the pokemon list
 }
}
```

The only part of this code that should look new to you is `localStorage.getItem("pokemon") === null`. This is **calling** the `getItem` `function` of `localStorage`, which comes built into **JavaScript**, and asking if it has an item called "pokemon". If it does, it will return that item. If it doesn't, it will return `null`, a special value that basically means "nothing". The three equals signs (`===`), you may remember, are how we check that two things are the same. So, if there is no value for "pokemon" stored in `localStorage` then **JavaScript** will **return** `null` and the test will pass, running the code to `fetch` and create your Pokémon. Otherwise, the `else` block runs… we'd better fill that in!

4 `localStorage` isn't as clever as the rest of **JavaScript**; it can only store text **strings**. Since your Pokemon are complex **objects**, and you have a whole **array** of them, that's a problem. However, you've already seen the solution! **JSON** is JavaScript Object Notation, specifically made to store complex **JavaScript objects** as **strings**. Thankfully, it's really easy to turn an **object** into **JSON** and back. You just need to add one line into the `if` statement to save the Pokémon list, like so:

```
if(localStorage.getItem("pokemon") === null) {
 pokemon = []
 await fetchManyPokemon(pokemonCount)
 await fetchPokemonImages()
 await makePokemonList(pokemonCount)
 localStorage.setItem("pokemon", JSON.stringify(pokemon))
}
```

This line **sets** the value **returned** by `JSON.stringify(pokemon)` as associated with the **key** `"pokemon"` in `localStorage`. As you might guess, `JSON.stringify()` turns the object passed to it into a **JSON string**!

5 Now you need to get those values back, if they already exist, and store them in the `pokemon` **array**, which is where the rest of your code will be looking for them. Notice how filling the same **array** in either situation means that only the `getPokemon` `function` needs to care about where the Pokémon are coming from! As you may have guessed, this is another quick call to `localStorage` and `JSON`. This time, it's `localStorage.getItem()` to **get** what you **set** previously and, once you've got it, `JSON.parse()` to turn it back into an **object**.

```
else {
 pokemon = JSON.parse(localStorage.getItem("pokemon"))
}
```

6 Finally, you need to add the check for the situation where you have three Pokémon stored, change the call to `getDex(15)`, and still just get three Pokémon back! Right now, the `if` is just checking that there are *any* stored and, if so, jumping to the `else` of just using them. You need to add a check where first it checks for *any* and then for *at least* the right number. What that actually means is checking that the length of the stored **array** is not less than the value of `pokemonCount`. Update the `if` statement to do that like so:

```
if(localStorage.getItem("pokemon") === null || JSON.parse(localStorage.getItem("pokemon")).length < pokemonCount) {
```

Remember that `||` (or) means that if the condition on either side is `true` then the whole statement counts as `true`. Likewise, `&&` (and) means that both conditions need to be `true` for the statement to count as `true`.

7 Now watch how fast the page loads, once it's had a chance to save the Pokémon!

1 Now that you have a list of Pokémon and they're saved and quick to reload, it's time to start working on displaying them! This is mostly about creating **HTML** elements with the content from your Pokémon **objects** in them. It would be helpful to have completed some of the HTML/CSS Sushi Cards, but if you haven't don't worry, this isn't really the same as writing **HTML**. This is telling **JavaScript** to do it for you! I've already provided some **CSS** that will handle laying things out for you, as well as the bits of a HTML **form** that you'll need later. For now, create a `function` that will let you create, instead of the single image of the first Pokémon you've got right now, a more detailed display of any Pokémon you **pass** it. To do this you're going to be creating a **HTML element** called a `figure` with an `img` (image) **element** and a `figcaption` (figure caption) **element** inside it. Your image will, naturally, be the picture of the Pokémon. The caption will be the name. I've included comments here (the lines starting with `//`) but you don't need to copy them into your code. They're just there to explain what's happening, as this is more complex than **HTML** creation you may have seen before. It's long, but it's just doing very similar things—creating **HTML elements**, setting some **properties** on them and placing them inside each other—a few times.

```
function makePokemonHTML(pokemon) {
  // Create a figure element
  var figure = document.createElement("figure")

  // Give the figure an id based on the Pokémon's id
  // you can use this to select the figure later
  // notice that .toString() is used to convert the number to a string
  figure.id = "pokemon-"+pokemon.id.toString()

  // Create an img (image) element
  var image = document.createElement("img")

  // Set the src (source) of that element to be the image of the Pokémon
  image.src = pokemon.image

  // Create a figcaption element
  var caption = document.createElement("figcaption")

  // Create a TextNode (piece of text) that includes the name of the Pokémon
  var pokeName = document.createTextNode("#"+pokemon.id+": "+pokemon.name)

  // Put the "pokeName" TextNode inside the "caption" figcaption tag
  caption.appendChild(pokeName)

  // Put the "image" img tag inside the "figure" figure tag
  figure.appendChild(image)

  // Put the "caption" figcaption tag inside the "figure" figure tag
  figure.appendChild(caption)

  // Hand back the completed piece of HTML
  return figure
}
```

2 Now you need a `function` that will loop through the Pokémon you've got and make **HTML** for each of them, `append`-ing it to the displayed list of Pokémon each time. This one's a bit simpler, but I've still included some comments to help. Again, you don't need to include them in your own code!

```
function displayPokemonList() {
  // Select the HTML tag with the id "display"
  // This is a tag I made for you and included in index.html
  var display =  document.getElementById("display")

  // Set all the HTML inside that tag to the empty string
  // that is, delete it.
  display.innerHTML = ""

  // For each Pokémon in the pokemon array,
  // make HTML using makePokemonHTML and then
  // put that HTML inside the display tag.
  // Note that the append tag will put the latest tag
  // in last. So they'll appear in the order they are in
  // in the pokemon array.
  for(var i = 0; i < pokemon.length; i++) {
    display.appendChild(makePokemonHTML(pokemon[i]))
  }
}
```

3 Finally, replace the code you've got for displaying a single Pokémon in `buildDex` with code that calls `displayPokemonList` like so:

```
async function buildDex(pokemonCount){
  await getPokemon(pokemonCount)
  displayPokemonList()
}
```

1

Now you've got a pretty cool looking list of Pokémon displaying for you, it's time to add some tools to help users find the Pokémon they're looking for. You're going to create two of them: A filter based on the type of the Pokémon and a search based on its name. To be able to filter by type, you need to have a list of all they types that Pokémon in your list have. The easiest way to do this is to make **JavaScript** create the list for you, as an **array**. You do this by:

- Creating an empty **array** to store your types (`typesList`)
- Looping over each of the Pokémon
- Inside that loop, looping over each of the current Pokémon's types
- Then check if the type is *not* already in your list. The code for that is `typesList.indexOf(type) === -1` . If it was in the **array** this would return the **index** (position number) of the type.
- If it is not in the **array**, adding it

Also, you may want to sort it A–Z to make finding a type easier for users. That's simple, as A–Z (or smallest to largest for numbers) is the default sort for **JavaScript arrays**, so you can just use `typesList.sort()` . Here's the new `function` you should add to your code:

```
function getTypesList() {
  var typesList = []
  for(var i = 0; i < pokemon.length; i++){
    for(var j = 0; j < pokemon[i].types.length; j++){
      var type = pokemon[i].types[j]
      if(typesList.indexOf(type) === -1){
        typesList.push(type)
      }
    }
  }
  typesList.sort() // alphabetize the list, to find things easily
  return typesList
}
```

2 Right, now you have that list, you need to update the HTML `<select>` tag, with the `type-picker` id, that users will use to pick types. Right now, there's a single `<option>` tag inside it and that just says “Loading” so users know not to use it until it's ready. You need to insert `<option>` tags for all of the types, as well as an “all” option, so they can see all the Pokémon. Notice that `<option>` tags don't just contain the type as text to display (the `TextNode`), but also as a `value` **property**. You then need to add a listener, like the ones used to detect clicks in the Intermediate JavaScript Sushi project, which will detect changes and run another `function` that you'll make in the next step, called `filterDexByType`. Here's the code for the `createTypePicker` function you'll use to do this:

```
function createTypePicker() {
 var typePicker = document.getElementById("type-picker")
 var typesList = getTypesList()

 typePicker.innerHTML = "<option value=\"all\">all</option>"

 for(var i = 0; i < typesList.length; i++){
   var type = document.createElement("option")
   type.value = typesList[i]
   var typeName = document.createTextNode(typesList[i])
   type.appendChild(typeName)
   typePicker.appendChild(type)
 }

 typePicker.addEventListener("change", filterDexByType)
}
```

Try testing the selection by adding an `alert` with the type. Don't forget to remove it once the filtering is working though!

3 Now you've got everything set up for your user to be able to tell the program what type they want to filter to. Time to actually filter things! The easiest way to do this is to just hide the Pokémon that don't match the filter, rather than adding or removing them. You can do that by changing the `style.display` property of the `<figure>` **tags** you created. So what you'll need is a loop that takes the type You can get that from the `this.value` property, since `this` will be referring to the **HTML** tag that triggered the `function`. That tag is the `<select>` tag, whose value is the value of the selected `<option>` tag. The `function` to do this (and handle the special case of showing everything if “all” is selected) is below, and you can go ahead and add it, then test your filters!

```
function filterDexByType(){
 var selectedType = this.value

 for(var i=0; i < pokemon.length; i++){
   var fig = document.getElementById("pokemon-"+pokemon[i].id.toString())

   if(pokemon[i].types.indexOf(selectedType) === -1){
     if(selectedType === "all"){
       fig.style.display = "initial"
     }
     else{
       fig.style.display = "none"
     }
   }
   else {
     fig.style.display = "initial"
   }

 }

}
```

1 Now you can filter to all the fire types, or whatever types you like (but fire types are the coolest, right? Charizard is awesome). Time to let your users search for their favourite Pokémon! This means connecting up the text box and the button I've included on the page for you. Their `id`s are `pokemon-name` and `pokemon-name-button`. The code just needs to listen for clicks on the button, take the `value` in `pokemon-name` and compare it to the `name` property of every Pokémon, to see if that name *contains* it, rather than perfectly matches it. So searching for “saur” should find bulbasaur, ivysaur and venusaur. You can test that using `.includes("saur")` on the **string** you're checking. That search function looks like this:

```
function searchDex(event) {
 event.preventDefault()
 var searchText = document.getElementById("pokemon-name").value
 for(var i=0; i < pokemon.length; i++){
   var fig = document.getElementById("pokemon-"+pokemon[i].id.toString())
   if(pokemon[i].name.includes(searchText)){
     fig.style.display = "initial"
   }
   else {
     fig.style.display = "none"
   }
 }
}
```

2 The last thing you need to do is connect it up to the click listener. There's not really a great place to do that in the code, so just stick it in the end of `buildDex`, since it's part of setting up the page. Like this:

```
async function buildDex(pokemonCount){
 await getPokemon(pokemonCount)
 displayPokemonList()
 createTypePicker()
 document.getElementById("pokemon-name-button").addEventListener("click",searchDex)
}
```

That's it! You've got a fully working Pokédex! Check out the next card for some ideas on how you can improve the project on your own.

1 This card is just a few ideas on where you could take the project next, if you want to keep working on it. Pick one that interests you and give it a shot with help from your Mentors and research on the internet.

- Start somewhere else in the Pokédex, say from number 100.
  - How do you need to change the `localStorage` code to handle this?
- Speed up increasing the list size—right now every Pokémon is re-loaded to `localStorage` if you add one. How can you be smarter about that?
- Right now, the two filters don't work together. So if you filter to “flying” and search for “char” you find all three “char” Pokémon instead of just Charizard, the only one with the “flying” type. Fix that!
- Add two buttons to sort the Pokémon A–Z and Z–A by name. Maybe add another to re-order them by id number again. You'll need to look up information on writing custom **array** sorting for this.
- Let me mark my six favourite (and no more!) Pokémon as my team. Save them, so they're still marked when I re-load the page. You'll need `locaStorage` for the saving. Look at the Intermediate Sushi Cards for ideas on handling the marking.
- Try adding more filters, using the **JSON** viewer from the “What is JSON?” card to pick some interesting values and creating some filters based on them like on the “Filtering Pokémon” card.