Chapter 6: Putting It All Together: The Budget App Project:

3. Project Setup & Details:

Download the starter project from GitHub. Its name is 6-Budgety.

4. Project Planning & Architecture: Step 1:

The first thing on our to do list is to add event handler. Actually, we input description and value and hit OK button and data will appear on UI. So it starts with the click of a button. We have to add an event handler for that

→ Then we have to get the data out of our input fields.

→ Then we have to add the new item to our data structure.

→ Then we will add the new item to the UI.

→ Then the budget calculation is done.

→ Update the UI.

TO-DO LIST:

→ Add event handler

→ Get input values

→ Add new item to our data structure.

→ Add the new item to the UI.

→ Calculate budget

→ Update the UI.

Structuring our code with modules:

Modules:

→ Important aspect of any robust application's architecture.

→ Keep the units of code for a project both cleanly separated and organized.

→ Encapsulate some data into privacy and expose other data publicly.

Modules allow us to break up our code into logical parts and then make them interact with one another.

Let's think about the tasks that we already have in our to-do list

UI Module:

→ Get input values
→ Add new item to the UI.
→ Update the UI.

Data module:

→ Add the new item to our data structure.
→ Calculate budget.

Controller module:

→ Add event handler.

5: Implementing the Module Patteon:

What you will learn in this lecture:

- → How to use the module pattern
- → More about private and public data, encapsulation and separation of concerns.

app.js:

```
var budgetController = (function() {

    var x = 23;

    var add = function(a) {
        return x + a;
    }

    return {
        publicTest: function(b) {
            return add(b);
        }
    }
})();

var UIController = (function() {

    // Some code

})();

var controller = (function(budgetCtrl, UICtrl) {

    var z = budgetCtrl.publicTest(5);
```

```
    return {
        anotherPublic: function() {
            console.log(z);
        }
    }

})(budgetController, UIController);
```

Output (In console):

```
controller.anotherPublic() ↵
28
undefined
```

Talking about the budgetController function:

We pass '5' into the function with 'b' and this '5' value is passed into the add function and the result will be 5 + x ie, 5 + 23 which is 28. Using the publicTest method, which is the one that we exposed to the public, actually works

while using the add methods, or the 'x' variable doesnot bcoz they're private.

When the JavaScript ountime hits this line:

```
var budgetController = (function() {
```

it gets executed and this anonymous function is declared and immediately invoked bcoz of this operator in the last line:

```
})();
```

Then these variables and functions are declared and we return an object:

```
return {
    publicTest: function(b) {
        console.log(add(b));
    }
}
```

This object contains the method publicTest. This object gets assigned to the budget variable after this function returns. Finally, the budgetController variable is simply an object.

containing the method called publicTest. Bcoz that's what we returned from this function :

```
var budgetController = (function(){
    .
    .
    .
})
```

Here, publicTest method uses 'add' and 'x' variable even after this function has executed. This works bcoz of Closures.

The IIFE returns immediately and it's gone. But the publicTest function that we return here will always have access to the 'x' variable and the add function b.coz a closure was created here.

That's why we say that the publicTest method is public because it was returned and now we can use it. But the 'x' and 'add' variables are private bcoz they

are in closure and only the publicTest method can access them.

Talking about the UIController function:

There's nothing written in this for now. Until now, the two modules that we have created: budgetController and UIController are completely independent modules. There will not be any interaction between these two.

Separation of conceons: This means that each part of the application should only be interested in doing one thing independently. We're using this here.

These two controllers don't know about each other. But we have to connect them in some way. For eg: we need to read data from UI and then add the data as a new expense in the budgetControllers. That's why we create a third module which is appControllers.

Talking about the controller function:

This controller has two parameters: budgetCtrl and UICtrl. We are storing the publicTest method in a variable called 'z'. As we don't have access from the outside to this variable, we have to create another method simply to print this variable to the console. This is the way we do it:

```
return {
    anotherPublic: function () {
        console.log (z);
    }
}
```

Doing this is the only way that from the outside we can have access to 'z'.

All of this is just to know how it works.

6. Setting Up the First Event Listeners:

What you will learn in this lecture:

→ How to set up event listeners for keypress events.

→ How to use event object.

app.js:

```
// BUDGET CONTROLLER
var budgetController = (function() {

    // Some code

})();

// UI CONTROLLER
var UIController = (function() {

    // Some code

})();

// GLOBAL APP CONTROLLER.
```

```
var controller = (function (budgetCtrl, UICtrl){

    var ctrlAddItem = function() {

        // 1. Get the field input data.

        // 2. Add the item to the budget
           controller

        // 3. Add the item to the UI.

        // 4. Calculate the budget.

        // 5. Display the budget on the UI.
    }        console.log('It works.');

document.querySelector('.add__btn').addEventLis-
    -tener('click', ctrlAddItem);

document.addEventListener('keypress', function(event){
    if (event.keyCode === 13 || event.which===13) {
        ctrlAddItem();
    }
});
})(budgetController, UIController);
```

7. Reading Input Data:

What you will learn in this lecture:

→ How to read data from different HTML input types.

app.js:

```
.
.
.
// UI Controller.
var UIController = (function () {

    var DOMStrings = {
        inputType: '.add__type',
        inputDescription: '.add__description',
        inputValue: '.add__value',
        inputBtn: '.add__btn'
    };

    return {
        getInput: function () {
```

This is the method to get all three inputs from the UI.

```
    return {
        type: document.querySelector(DOMStrings.inputType).
                value, // will be either inc or exp.
        description: document.querySelector(DOMStrings.
                inputDescription).value,
        value: document.querySelector(DOMStrings.
                inputValue).value
    };
},

getDOMStrings: function() {
    return DOMStrings;
}
};
})();

// GLOBAL APP CONTROLLER
var controller = (function(budgetCtrl, UICtrl){
    var DOM = UICtrl.getDOMStrings(); // we are getting DOMstrings.
    var ctrlAddItem = function(){
```

We are exposing DOMstrings to the public using this method.

```
//1. Get the field input data.
var input = UICtrl.getInput();
console.log(input);

//2. Add the item ....
```

8. Creating an Initialization Function:

What you will learn in this lecture:

→ How and why to create an initialization function.

We are creating an init function bcoz we want to have a place where we can put all the code that we want to be executed right at the beginning when our application starts.

We should organize our code so that we only have functions in the controller.

app.js:

```
// GLOBAL APP CONTROLLER.

var controller = (function (budgetCtrl, UICtrl){

    var setupEventListeners = function() {

        var DOM = UICtrl.getDOMStrings();

        document.querySelector(DOM.inputBtn).
            addEventListener('click', ctrlAddItem);

        document.addEventListener('keypress', function(
            event) {
            if ( event.keycode === 13 || event.which
                                    === 13) {

                ctrlAddItem();
            }

    });

});

var ctrlAddItem = function() {
```

```
    // 1. Get the field input data
    var input = UICtrl.getInput();

    // 2. Add the.....

    .
    .
    .

    // 5. Display the budget on the UI.
};

return {
    init: function(){
        console.log('Application has started.');
        setupEventListeners();
    }
};

})(budgetController, UIController);

controller.init();
```

9. Creating Income & Expense Function Constructors:

What you will learn in this lecture:

→ How to choose function constructors that meet our application's needs.

→ How to set up proper data structure for our budget controller.

We will discuss about how we will store our income and expense data in a budget controller.

app.js:

```
// BUDGET CONTROLLER
var budgetController = (function() {
    // we use capital letters to distinguish constructors from other functions.
    var Expense = function(id, description, value){
        this.id = id; // this.id should be equal to the ID that we pass into our function constructor.
        this.description = description;
        this.value = value;
    };

    var Income = function(id, description, value){
```

```
    this.id = id;
    this.description = description;
    this.value = value;
};

var data = {
    allItems: {
        exp: [],
        inc: []
    },
    totals: {
        exp: 0,
        inc: 0
    }
};
})();
```

Summary of the code:

→ We need a data model for expenses and incomes.

→ We know that each new item will have a description and a value. We should have some way to distinguish between different incomes or expenses. We want them to have a unique ID number as well. How would you store this kind of data? We would choose an object. So an object that simply has a description, a value and an ID. What do we do when we want to create lots of objects? We create function constructors. which we can then use to instantiate lots of income and expense objects. Like that, we basically create a custom data type for incomes and expenses.

→ The budget controller keeps track of all the

incomes and expenses and also of the budget itself and later also the percentages. So we need a good data structure for that. Imagine that the user would input 10 incomes so we would create 10 income objects. So where would we store all of these 10 incomes. The best solution is to store these into an array.

10. Adding a New Item to our Budget Controller:

What you will learn in this lecture:

→ How to avoid conflicts in our data structures.

→ How and why to pass data from one module to another.

app.js:

```
//BUDGET CONTROLLER

var budget.....

:

:

var data = {

    allItems : {
```

```
    ⋮
};

return {
    addItem: function (type, des, val) {
        var newItem, ID;

        // [1 2 3 4 5], next ID = 6
        // [1 2 4 6 8], next ID = 9
        // ID = last ID + 1

        // Create new ID
        if ( data.allItems [type].length > 0){
            ID = data.allItems[type][data.
                allItems [type].length - 1].id + 1;
        } else {
            ID = 0;
        }

        // Create new item based on 'inc'
        or 'exp' type
```

```
if (type === 'exp') {
    newItem = new Expense(ID, des, val);
} else if (type === 'inc') {
    newItem = new Income(ID, des, val);
}

// Push it into our data structure
data.allItems[type].push(newItem);

//Return the new element
return newItem;
}
,
testing: function() {
    console.log(data);
}
};
})();
```

This is just to test the code.

```
var CtrlAddItem = function(){
    var input, newItem;

    // 1. Get the field input data.
```

```
input = UICtrl.getInput();

//2. Add the item to the budget controller
newItem = budgetCtrl.addItem(input.type,
    input.description, input.value);

//3. Add...
:
```

Summary of the code:

→ We've created a public method in the budget controller that can allow other modules to add a new item into our data structure. So we return an object once again which contains all of our public methods which for now is: addItem.

→ If someone calls this addItem method, what do they have to tell us in order that we can create a new item? First, we would want to know the type: if it's an income or an expense. We need the description and

the value of the income or expense. We have used different names here (type, des, val) so that we have less confusion. So in this method, they have their own names.

→ Now, how would we add a new expense? or income

We do it like this:

```
return {
    addItem: function(type, des, val) {
        var newItem, ID;

        ID = 0;

        if (type === 'exp'){
            newItem = new Expense (ID, des, val);
        } else if ((type === 'inc') {
            newItem = new Income (ID, des, val);
        }

    }
};
```

If we look at the controllers, the 'input' that we receive will actually contain the type:

inc or exp.

→ In getInput method, we are returning 'type' and we're going to later pass into our addItem method. We are using the 'type' in addItem method.

→ And the ID, we can think about it later.

→ Now we need to push the inc or exp data into the respective arrays. For that, we did this:

```
data.addItems[type].push(newItem);
```

→ Then we returned the newItem. Because only if we return, the other module or the other function that's going to call this one can have direct access to the item that we just create.

→ For IDs, we have to use the length of the already existing array and add one. If we don't have any item in our data structure,

the arrays are empty and the length is '0'. So we have added this condition: when the array is empty, the new ID should be zero. We wrote this for ID:

```
if (data.allItems[type].length > 0){
    ID = data.allItems[type][data.allItems[type].
    length - 1].id + 1;
} else {
    ID = 0;
}
```

→ To add the new item to the budget controller:

```
budgetCtrl.addItem(input.type, input.description,
                   input.value);
```

→ We wrote the 'testing' method just to test it in the console.

11. Adding a New Item to the UI:

What you will learn in this lecture:

→ A technique for adding big chunks of HTML into the DOM.

→ How to replace parts of strings.

→ How to do DOM manipulation using the innerAdjacentHTML method.

app.js:

```
.
.
.

// UI CONTROLLER
var UIController = (function() {

    var DOMstrings = {
        inputType: '.add__type',
        inputDescription: '.add__description',
        inputValue: '.add__value',
        inputBtn: '.add__btn',
        incomeContainer: '.income__list',
        expensesContainer: '.expenses__list'
    };

    return {
        getInput: function() {
            .
            .
            .
```

```
    };
},
addListItem: function (obj, type) {
    var html, newHtml, element;
    // Create HTML string with placeholder
    text.
    if (type === 'inc') {
        element = DOMstrings.incomeContainer;
        html = '<div class="item clearfix"
        id="income-%id%"> <div class="
        item__description">%description%</div>
        <div class="right clearfix"> <div
        class="item__value">%value%. </div>
        <div class="item__delete"> <button
        class="item__delete--btn"> <i class=
        "ion-ios-close-outline"> </i> </button>
        </div> </div> </div>';
    } else if (type === 'exp') {
```

```
element = DOMstrings.expensesContainer;

html = '<div class="item clearfix" id=
"expense-%id%"><div class="item__
description">%description%</div><div
class="right clearfix"><div class="
item__value">%value%</div><div class=
"item__percentage">21%</div><div
class="item__delete"><button class="
item__delete--btn"><i class="ion-ios-
close-outline"></i></button></div></div>
</div>';
}

// Replace the placeholder text with some
actual data

newHtml = html.replace('%id%', obj.id);
newHtml = newHtml.replace('%description%',
    obj.description);
newHtml = newHtml.replace('%value%', obj.
                          value);
```

```
        // Insert the HTML into the DOM.
        document.querySelector(element).insertAdjacentHTML
            ('beforeend', newHtml);
    },

    getDOMstrings: function() {
        return DOMstrings;
    }
};
})();

var ctrlAddItem = function() {
    var input, newItem;

    // 1. Get the field input data
    :

    // 3. Add the item to the UI.
    UIctrl.addListItem(newItem, input.type);

    // 4. ...
    :
```

Summary of the code:

→ We wrote a new public method called addListItem in UI controller. We need an object and type (inc or exp) to add this new item to the list. This object is the same object that we created using a function constructor and then passed to our app controller in the last lecture.

→ In the ctrlAddItem function (this is the function that's called when someone hits input button / enter key), we first read the input out of the fields, store it into the input variable, then using this input variable, and addItem method, we create a new item and we return it and store it in newItem variable. So in the newItem variable, we have the new object.

And that's the object that we're gonna pass to our addListItem method.

→ In addListItem method, we have three things to do. First thing is:

1. Create HTML string with placeholder text:

```
addListItem: function (obj, type){
    var html;

    if (type === 'inc'){
        html = '<div. ... </div';
    } else if (type === 'exp'){
        html = '<div .... </div>';
    }
```

We have used "%.id.%." kind of things in html bcoz the placeholder text is easier to find & we don't override something that we don't want.

2. Replace the placeholder text with actual data.

We use a never before used method called 'replace' method.

```
newHtml = html.replace('%id%', obj.id);
```

What replace method does is this: it searches for a string and then replaces that string with the data that we put into the method. Here we are replacing '%id%' with 'obj.id'.

3. Insert the HTML into the DOM.

The insertAdjacentHTML method first accepts the position and then the text, ~~to the~~ ie, the string, the HTML that we want to insert. There are four different positions in which we can insert the HTML.

Visualization of position names:

```
<!-- beforebegin -->
<p>
    <!-- afterbegin -->
      foo
    <!-- beforeend -->
</p>
<!--afterend-->
```

All of the elements that we add to the ".expense__list" will be the childs of ".expense__list" container. For income list, they are the childs of ".income__list".

We use 'beforeend' to append a new item.

```
document.querySelector(element).insertAdjacentHTML(
    'beforeend', newHtml);
```

(element: it comes from DOMStrings.)

→ To add the item to the UI, we wrote this:

```
UICtrl.addListItem(newItem, input.type);
```

12. Clearing Our Input Fields:

What you will learn in this lecture:

→ How to clear HTML fields.

→ How to use querySelectorAll

→ How to convert list into an array.

→ A better way to loop over an array:

for each loops.

app.js:

```
    .
    .
    .

//Insert the HTML into the DOM
document.querySelector(element).insertAdjacentHTML
    ('beforeend', newHtml);

},

clearFields: function () {

    var fields, fieldsArr;

    fields = document.querySelectorAll(DOMstrings.
        inputDescription + ', ' + DOMstrings.
                                    inputValue);
```

```
fieldsArr = Array.prototype.slice.call(fields);

fieldsArr.forEach(function(current, index,
                                    array) {

    current.value = "";

});

fieldsArr[0].focus();

};

.
.
.

var ctrlAddItem = function(){

    .
    .
    .

    // 4. Clear the fields

    UICtrl.clearFields();

    .
    .
    .

};

.
.
.
```

Summary of the code:

→ We wrote a public method called clearFields. We don't need any parameters here bcoz we know which fields we're gonna clear.

The syntax of querySelectorAll is like CSS selecting. We separate different selectors with a comma

→ But the querySelectorAll returns a list. We need to convert this list into an array.

→ There's an array method called "slice". What slice does is to return a copy of the array that it's called on. Usually, we call this method on an array and it returns another array. But we can trick this method and pass a list into it

and it will still return an array. This is the way to do it:

```
fieldsArr = Array.prototype.slice.call(fields);
```

We called the slice method using the call method and then passed the 'fields' variable into it so that it becomes the this variable.

The slice method is stored in the Array prototype. This Array is the function constructor for all arrays. All the methods that the arrays inherit from the Array function constructors are in the Array's prototype property. So the slice method is also there.

→ Then we've used a for-each method. All we did is to pass a callback function into this method and then this callback function is applied to each of the elements

in the array. Here we have access to three things: current, index, array.

- current → current value: value of the array that is currently being processed
- index → this goes from zero to length of array minus one.
- array → entire array.

We want to see them empty.

→ Finally, we have used this method in the controller.

→ The final thing we did is to set the focus back to the first field. We did it using this:

```
fieldsArr[0].focus();
```

13. Updating the Budget Controller:

- What you will learn in this lecture:
  - → How to convert input fields to numbers.
  - → How to prevent false inputs.

In ctrlAddItem function, we have the spots where we have to fill "calculate the budget" and "Display the budget on the UI" methods. These methods will handle the budget update. We have to move this to a separate function. Bcoz, these steps belong together and we will do all of this again when we want to delete an item. So instead of repeating this code, we can simply write a new function to keep the DRY ~~function~~ principle. For that, we wrote updateBudget.

app.js:

```
:
:

return {
    getInput: function() {
        return {
            type: docu....
                :
```

```
        value: parseFloat(document.querySelector(DOMstrings.inputValue).value)
    };
},
.....

document.addEventListener('keypress', function(event) {
    if (event.keyCode === 13 || event.which === 13) {
        ctrlAddItem();
    }
});

};

var updateBudget = function() {
    // 1. Calculate the budget
    // 2. Return the budget
    // 3. Display the budget on the UI.
};
```

```
var ctrlAddItem = function () {

    var input, newItem;

    // 1. Get the field input data
    input = UICtrl.getInput();

    if (input.description !== "" && !isNaN(input.value)
            && input.value > 0) {

        // 2. Add the item to the budget controller
        newItem = budgetCtrl.addItem(input.type,
            input.description, input.value);

        // 3. Add the item to the UI
        UICtrl.addListItem(newItem, input.type);

        // 4. Clear the fields
        UICtrl.clearFields();

        // 5. Calculate and update budget
        updateBudget();
    }

};
```

We wrote the parseFloat method to convert the string into an integers. We added an if condition in ctrlAddItem to avoid getting null values when we click ok without entering anything.

14. Updating the Budget: Budget Controller:

What you will learn in this lecture:

→ How and why to create simple, reusable functions with only one purpose.

→ How to sum all elements of an array using forEach method.

app.js:

```
// BUDGET CONTROLLER
var budgetController = ...:

    :

var Income = fun...
    :
};
```

```
var calculateTotal = function (type) {
    var sum = 0;
    data.allItems[type].forEach(function(cur){
        sum += cur.value;
    });
    /*
    0
    [200, 400, 100]
    sum = 0 + 200
    sum = 200 + 400
    sum = 600 + 100 = 700
    */
    data.totals[type] = sum;
};

var data = {
    allItems: {
        exp: [],
        inc: []
    },
```

This is how the forEach loop works.

```
totals: {
    exp: 0,
    inc: 0
},

budget: 0,
percentage: -1

};

.
.
.

//Push it into our data structure
data.allItems[type].push(newItem);

//Return the new element
return newItem;

},

calculateBudget: function() {
    //Calculate total income & expenses
    calculateTotal('exp');
```

```
        calculateTotal('inc');

        // Calculate the budget: income - expenses
        data.budget = data.totals.inc - data.totals.exp;

        // Calculate the percentage of income that
           we spent
        if (data.totals.inc > 0) {
            data.percentage = Math.round((data.
                totals.exp / data.totals.inc) * 100);
        } else {
            data.percentage = -1;
        }

    },

    getBudget: function() {
        return {
            budget: data.budget,
            totalInc: data.totals.inc,
            totalExp: data.totals.exp,
            percentage: data.percentage
        };
    },
```

```
var updateBudget = function() {

    // 1. Calculate the budget
    budgetCtrl.calculateBudget();

    // 2. Return the budget
    var budget = budgetCtrl.getBudget();

    // 3. Display the budget on the UI
    console.log(budget);

};
```

Summary of the code:

→ We wrote a calculateBudget public method in the budgetController.

→ Since we don't want anyone to use the sum of incomes & expenses function, we made calculateTotal a private function.

→ We have all expenses in allItems and

all incomes in inc array of allItems. We just need to loop over these and sum all the values. For that, we created a sum variable which was initially '0'.

→ To store the sums inside exp and inc of totals, we did this:

```
data.totals[type] = sum;
```

→ In calculateBudget method, we called the private function like this:

```
calculateTotal('exp');
calculateTotal('inc');
```

} with these two lines, the totals: exp & inc will be set to total exp & total inc.

→ Finally, we have used this method in global app control:

```
budgetCtrl.calculateBudget();
```

To return the budget to us, we wrote this:

```
getBudget: function(){
    :
}
```

To store what we have returned in a variable, we wrote this:

```
var budget = budgetCtrl.getBudget();
```

15. Updating the Budget UI Controller:

What you will learn in this lecture:

→ Practice DOM manipulation by updating the budget and total values.

app.js:

```
...

//UI CONTROLLER

var UIController = ....

    var DOMstrings = {
    ...

    budgetLabel: '.budget__value',
    incomeLabel: '.budget__income--value',
    expensesLabel: '.budget__expenses--value',
    percentageLabel: '.budget__expenses--percentage'
    };
```

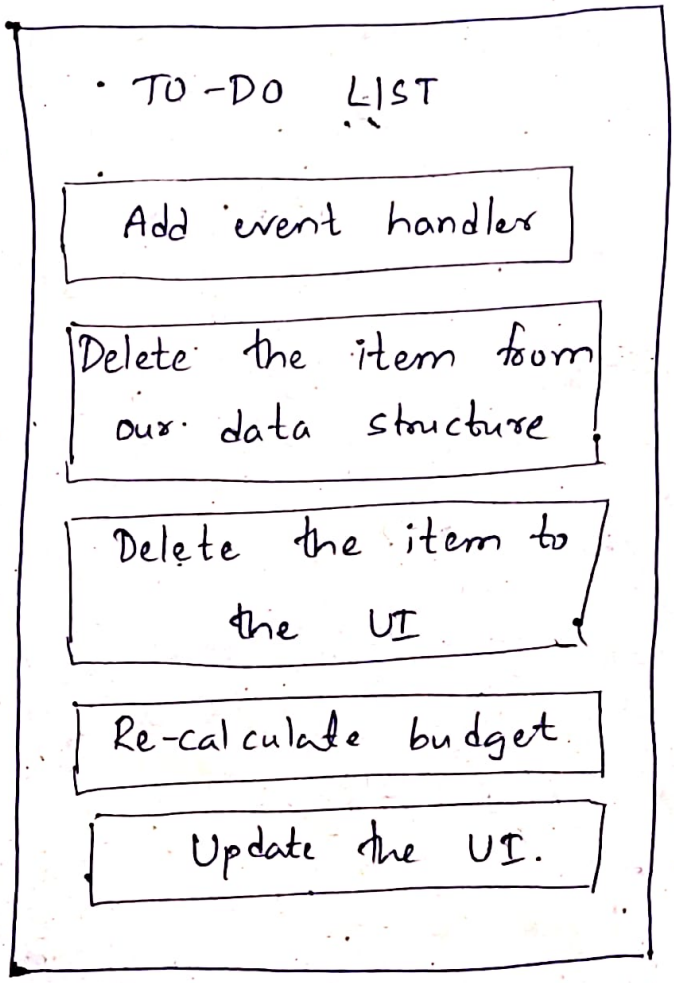
```
        fieldsArr[0].focus();
    },

    displayBudget: function(obj) {
        document.querySelector(DOMstrings.budgetLabel).
            textContent = obj.budget;
        document.querySelector(DOMstrings.incomeLabel).
            textContent = obj.totalInc;
        document.querySelector(DOMstrings.expensesLabel).
            textContent = obj.totalExp;

        if (obj.percentage > 0) {
            document.querySelector(DOMstrings.percen-
            -tageLabel).textContent = obj.percentage
                                    + '%';
        } else {
            document.querySelector(DOMstrings.
                percentageLabel).textContent = '---';
        }
    },
```

```
var updateBudget = function () {

    // 1. Calculate the budget

    budgetCtrl.....

    // 3. Display the budget on the UI

    UICtrl.displayBudget(budget);

};

return {
    init: function () {
        console.log('App. has started.');
        UICtrl.displayBudget({
            budget: 0,
            totalInc: 0,
            totalExp: 0,
            percentage: -1
        });
        setupEventListeners();
    }
};

})(budgetController, UIController);

controller.init();
```

16. Project Planning & Architecture: Step 2:

Refer the screenshot of After step 1...

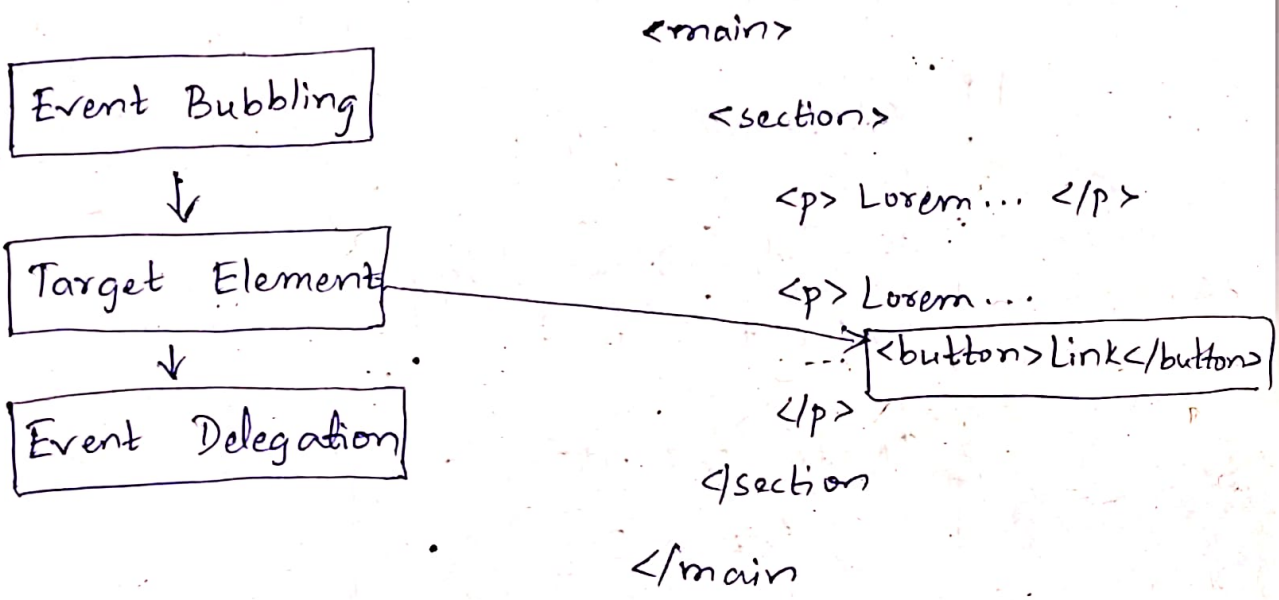
To-do List:



17: Event Delegation:

If an event bubbles up in the DOM tree and if we know where the event was fired, then we can simply attach an event handler to a parent element and

wait for the event to bubble up and we can then do whatever we intended to do with our target element. This is called Event Delegation.



Use cases for Event Delegation:

→ When we have an element, with lots of child elements that we're interested in;

→ When we want an event handler attached to an element that is not yet in the DOM when our page is loaded.

18. Setting Up the Delete Event Listener Using Event Delegation:

What you will learn in this lecture:

→ How to use event delegation in practice.

→ How to use IDs in HTML to connect the UI with the data model.

→ How to use the "parentNode" property for DOM traversing.

app.js:

```
...

// UI CONTROLLER
var UIController = (function() {

    var DOMstrings = {

        ...

        container: '.container'

    };

    ...
```

```
// GLOBAL APP CONTROLLER.

var controller = (function (budgetCtrl, UICtrl){

    var setupEventListeners = function( ){

        .
        .
        .

        document.querySelector (DOM.container).
            addEventListener ('click', ctrlDeleteItem);

    };

    .
    .
    .

    //3. Add the item to the UI

    UICtrl. addListItem (newItem, input.type);

    //4. Clear the fields

    UICtrl. clearFields ( );

    //5. Calculate and update budget

    updateBudget ( );

}

};
```

```
var ctrlDeleteItem = function(event) {

    var itemID, splitID, type, ID;

    itemID = event.target.parentNode.
        parentNode.parentNode.parentNode.id;

    if (itemID) {

        //inc-1
        splitID = itemID.split('-');
        type = splitID[0];
        ID = parseInt(splitID[1]);

        //1. Delete the item from the data
                                    structure

        //2. Delete the item from the UI

        //3. Update and show the new
                                    budget.

    }

};
```

This is how 'split' method works:

console:

```
var s = 'inc-1' ↵
s.split('-') ↵
["inc", "1"]

var s = 'inc-1-type-3' ↵
s.split('-') ↵
["inc", "1", "type", "3"]
```

19. Deleting an Item from our Budget Controller:

What you will learn in this lecture:

→ Yet another method to loop over an array: map.

→ How to remove elements from an array using the "splice" method.

app.js:

```
    // Push it into our data structure
    data.allItems[type].push(newItem);

    // Return the new element
    return newItem;
},

deleteItem: function(type, id) {
    var ids, index;

    //id = 6
    // data.allItems[type][id];
    // ids = [1 2 4 6 8]
    // inddex = 3

    ids = data.allItems[type].map(function(current) {
        return current.id;
    });
```

```
        index = ids.indexOf(id);

        if (index !== -1) {

            data.allItems[type].splice(index, 1);

        }

    },

    .....

var ctrlDeleteItem = function(event){

    var ....

    .....

    if (...) {

        .....

        // 1. Delete the item from the data structure
        budgetCtrl.deleteItem(type, id);

        .....

    }

};

    .....
```

Map method receives a callback function, which has access to the current element, the current index and the entire array.

The difference bn map and for-each is that map returns a brand new array.

```
var ids = data.allItems[type].map(function(current){
    return current.id;
});
```

With the above code, we would, end up with a new array with the same length as "allItems[type]" array but with "current.id" in all of the elements.

```
index = ids.indexOf('id');
```

The above code returns the index number of the element of the array that we input ('id' in this case).

Splice element is used to remove elements.

```
if (index !== -1){
    data.allItems [type].splice(index, 1);
}
```

The first argument in the splice method is the position number at which we want to start deleting. The second argument is the number of elements that we want to delete.

20. Deleting an Item from the UI:

app.js:

```
.....

    //Insert the HTML into the DOM.
    document.querySelector(element).insertAdja-
        -centHTML('beforeend', newHtml);
},
```

```
deleteListItem: function(selectorID) {
    var el = document.getElementById(seletorID);
    el.parentNode.removeChild(el);
},
:
:

var ctrlDeleteItem = function(event){
    :
    :
    //2. Delete the item from the UI
    UICtrl.deleteListItem(itemID);

    //3. Update and show the new budget
    updateBudget();
}
};
:
:
```

Open this link:

blog.garstasio.com/you-dont-need-jquery/dom-manipulation/

21. Project Planning & Architecture: Step 3:

TO-DO LIST

- Calculate Percentages
- Update Percentages in UI
- Display the current month and year
- Number formatting
- Improve input field UX

22. Updating the Percentages Controller:

What you will learn in this lecture:

→ Reinforcing the concepts and techniques we have learned so far.

When will the income percentages actually be updated? These percentages are calculated and updated each time we add or delete an item. These percentages are the percentage

of the income that each expense represents.

So we need to create a new function and call it in ctrlAddItem and ctrlDeleteItem.

app.js:

```
.
.
.
var updateBudget = function() {
.
.
.
};

var updatePercentages = function () {
    //1. Calculate percentages

    //2. Read percentages from the budget
       controller

    //3. Update the UI with the new
    percentages.
};
```

```
var ctrlAddItem = function () {
    var input, newItem;

    ...

    // 6. Calculate and update percentages
    updatePercentages();
}
};

var ctrlDeleteItem = function (event) {
    var ...

    ...

    // 4. Calculate and update percentages.
    updatePercentages();
}
};

...
```

23. Updating the Percentages Budget Controller:

What you will learn in this lecture:

→ How to make our budget controller interact with the Expense prototype.

app.js:

```
// BUDGET CONTROLLER.
var budgetController = (function () {
    var Expense = function (id, description,
        value) {
        this.id = id;
        this.description = description;
        this.value = value;
        this.percentage = -1;
    };

    Expense.prototype.calcPercentage = function (
        totalIncome) {
        if (totalIncome > 0) {
```

```
        this.percentage = Math.round((this.value /
            totalIncome) * 100);
    } else {
        this.percentage = -1;
    }
};

Expense.prototype.getPercentage = function(){
    return this.percentage;
};

.
.
.

calculateBudget: function() {

    .
    .
    .

},

calculatePercentages: function() {
    data.allItems.exp.forEach(function(cur){
        cur.calcPercentage(data.tools.inc);
```

```
    });
},

getPercentages: function(){
    var allPerc = data.allItems.exp.map(
        function(cur) {
            return cur.getPercentage();
    });
    return allPerc;
},
:

var updatePercentages = function(){
    // 1. Calculate percentages
    budgetCtrl.calculatePercentages(.);

    // 2. Read percentages from the budget
                                    controller
    var percentages = budgetCtrl.getPercentages(.);
```

```
    //3. Update the UI with the new
        percentages.
    console.log(percentages);
};
```

Summary of the code:

→ We need to calculate the expense percentage for each of the expense objects that are stored in the expenses array. We're always gonna need the total income to calculate the percentages.

→ We need to do this for each object individually. So there should be a method on each of these expense objects that calculates this percentage

→ So we added a method in Expenses function constructor. We'll add it in its

prototype so that all of the objects that are created through the expense prototype will inherit this method bcoz of the prototype chain.

→ We have created a getPercentage for this function constructor. This just retrieves the percentage from the object and returns it.

→ In calculatePercentages method, we have retrieved all expenses and calculated the percentages.

→ In getPercentages method, we want to loop over and store the data somewhere. That's why we used map. allPerc is an array with all of the percentages.

→ Finally, we have used these methods in our Controller: updatePercentages.

24. Updating the Percentages UI Controller:

What you will learn in this lecture:

→ How to create our own forEach function but for nodeLists instead of arrays.

app.js:

```
.....

// UI CONTROLLER.
var UIController = (function () {
    var DOMstrings = {
        .....
        expensesPercLabel: '.item__percentage'
    };
    .....
    displayBudget: function (obj) {
        document.....
        .....
    },
```

```
displayPercentages: function (percentages) {
    var fields = document.querySelectorAll(
        DOMstrings.expensesPercLabel);

    var nodeListForEach = function(list, callback){
        for (var i=0; i< list.length; i++){
            callback(list[i], i);
        }
    };

    nodeListForEach(fields, function(current,
            index) {
        if (percentages[index] > 0) {
            current.textContent = percentages
                [index] + '%';
        } else {
            current.textContent = '---';
        }
    });
},
```

```
var updatePercentages = function(){

    //1.....

    //3. Update the UI with the new percentages
    UICtrl.displayPercentages(percentages);
};
```

Summary of the code:

→ When we call our nodeListForEach function, we pass a callback function into it. This is the callback function:

```
function(current, index){

});
```

This function is assigned to this callback parameter: callback. In there, we loop over

our list and in each iteration, the callback function gets called with these arguments: list[i], i which are exactly these ones: current, index. So this code will be executed a number of times based on the length. Then we'll have access to the current element and to the current index because we passed them into the callback with these: list[i], i.

→ current.textContent = percentages[index];

This means, we want the current element and we used textContent on the current element. We want the percentages at the position 'index'. So imagine at the first element, we want the first percentage, at the second element, we want the second percentage in the array and so on.

→ Finally, with UIController, we are pushing it to the UI.

25. Formatting Our Budget Numbers: String Manipulation:

What you will learn in this lecture:

→ How to use different String methods to manipulate strings.

app.js:

```
:

//UI CONTROLLER.

var UIController = (function(){

    var DOMstrings = {

        :

    };

    var formatNumber = function(num, type){

        var numSplit, int, dec, type;

        /*

        + or - before number

        exactly 2 decimal points

        comma separating the thousands

        2310.4567 → 2,310.46.

        2000 → 2,000.00

        */
```

```
num = Math.abs(num);
num = num.toFixed(2);
numSplit = num.split('.');
int = numSplit[0];
if (int.length > 3) {
    int = int.substr(0, int.length - 3) +
        ',' + int.substr(int.length - 3, 3);
    // Above line means: if input: 23510, output:
                                        23,510
}
dec = numSplit[1];
return (type === 'exp' ? '-' : '+') + ' ' +
    int + '.' + dec;
};
```

abs which means absolute removes the sign of the number.

console:
(2.4567).toFixed(2)↵

"2.46" ← This returns a string.

(2).toFixed(2)↵
"2.00".

```
displayBudget: function(obj) {
    var type;
    obj.budget > 0 ? type = 'inc' : type = 'exp';

    document.querySelector(DOMstrings.budgetLabel).
      textContent = formatNumber(obj.budget, type);

    document.querySelector(DOMstrings.incomeLabel).
      textContent = formatNumber(obj.totalInc, 'inc');

    document.querySelector(DOMstrings.expensesLabel).
      textContent = formatNumber(obj.totalExp, 'exp');

    .
    .
    .
};
```

26. Displaying the Current Month & Year:

What you will learn in this lecture:

→ How to get the current date by using the Date object constructor.

app.js:

```
displayPercentages: function(percentages){
    .....
    .....
},

displayMonth: function(){
    var now, months, month, year;

    now = new Date();

    //var christmas = new Date(2016, 11, 25);
```

This means December coz 'index(11)' is December.

```
    months = ['January', 'February', 'March', 'April',
    'May', 'June', 'July', 'August', 'September',
    'October', 'November', 'December'];

    month = now.getMonth();

    year = now.getFullYear();
```

```
document.querySelector(DOMstrings.dateLabel).
    textContent = months[month] + ' ' + year;
};
.
.
.
return {
    init: function(){
        console.log(...);
        UICtrl.displayMonth();
        UICtrl.displayBudget({
            .
            .
            .
        });
        .
        .
        .
```

27. Finishing Touches: Improving the UX:

What you will learn in this lecture:

→ How and when to use 'change' events.

app.js:

First of all, cut nodeListForEach function and paste it after the formatNumber function so that we can reuse it.

```
displayMonth: function(){
    ...
changedType: function(){
    var fields = document.querySelectorAll(
        DOMstrings.inputType + ',' +
        DOMstrings.inputDescription + ',' +
        DOMstrings.inputValue);

    nodeListForEach(fields, function(cur) {
        cur.classList.toggle('red-focus');
    });

    document.querySelector(DOMstrings.inputBtn).
        classList.toggle('red');

},
getDOMstrings: function(){
    return DOMstrings;
}

};
```

```
// GLOBAL APP CONTROLLER

var controller = ....

    var setupEventListeners = function(){

        var DOM = .....

        :

        document.querySelector(DOM.inputType).
            addEventListener('change', UICtrl.
                changedtype);

    };

    :
```

28: We've Made It: Final Considerations:

Refer the screenshot to have a look at the final architecture of our app.