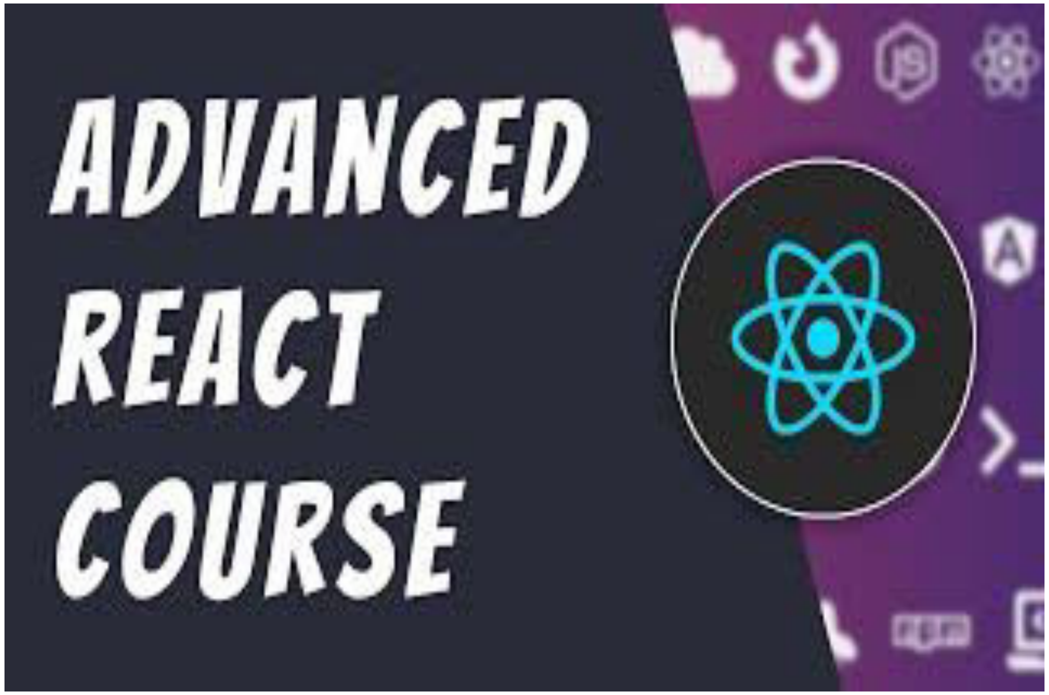
Advanced React



React Refs

- Refs is the shorthand used for **references** in React.
- It is an **attribute** which makes it possible to store a reference to particular DOM nodes or React elements.
- It provides a way to access React DOM nodes or React elements and how to interact with it.
- It is used when we want to change the value of a child component, without making the use of props.

When to Use Refs

- When we need DOM measurements such as managing focus, text selection, or media playback.
- It is used in triggering imperative animations.
- When integrating with third-party DOM libraries.
- It can also use as in callbacks.

When to not use Refs

- Its use should be avoided for anything that can be done **declaratively**.
- For example, instead of using **open()** and **close()** methods on a Dialog component, you need to pass an **isOpen** prop to it.
- You should have to avoid overuse of the Refs.

How to create Refs

- In React, Refs can be created by using **React.createRef()**.
- It can be assigned to React elements via the **ref** attribute.
- It is commonly assigned to an instance property when a component is created, and then can be referenced throughout the component.

```
class MyComponent extends React.Component {
 constructor(props) {
  super(props);
  this.callRef = React.createRef();
 }
 render() {
```

How to access Refs

- In React, when a ref is passed to an element inside render method, a reference to the node can be accessed via the current attribute of the ref.
- **const** node = **this**.callRef.current;

Refs current Properties

- The ref value differs depending on the type of the node:
- When the ref attribute is used in HTML element, the ref created with **React.createRef()** receives the underlying DOM element as its **current** property.
- If the ref attribute is used on a custom class component, then ref object receives the **mounted** instance of the component as its current property.
- The ref attribute cannot be used on **function components** because they don't have instances.

Add Ref to DOM elements

In the below example, we are adding a ref to store the reference to a DOM node or element.

```
import React, { Component } from 'react';
import { render } from 'react-dom';
 class App extends React.Component {
 constructor(props) {
  super(props);
  this.callRef = React.createRef();
  this.addingRefInput = this.addingRefInput.bind(this);
 }
  addingRefInput() {
  this.callRef.current.focus();
 }
  render() {
```

```
return (
     <div>
       <h2>Adding Ref to DOM element</h2>
       <input  type="text"   ref={this.callRef} />
       <input  type="button“  value="Add text input"

         onClick={this.addingRefInput}
       />
     </div>
   );
  }
}
export default App;
```

Add Ref to Class components

In the below example, we are adding a ref to store the reference to a class component.

Example

```
import React, { Component } from 'react';
import { render } from 'react-dom';
  function CustomInput(props) {
 let callRefInput = React.createRef();
   function handleClick() {
  callRefInput.current.focus();
 }
   return (
  <div>
    <h2>Adding Ref to Class Component</h2>
    <input
      type="text"   ref={callRefInput} />
<input
       type="button" value="Focus input"   onClick={handleClick}  />
   </div>
 );
}
class App extends React.Component {
 constructor(props) {
   super(props);
   this.callRefInput = React.createRef();
 }
   focusRefInput() {
   this.callRefInput.current.focus();
 }
```

```
render() {
    return (
      <CustomInput ref={this.callRefInput} />
    );
  }
}
export default App;
```

Callback refs

- In react, there is another way to use refs that is called "callback refs" and it gives more control when the refs are set and unset.
- Instead of creating refs by createRef() method, React allows a way to create refs by passing a callback function to the ref attribute of a component.
- It looks like the below code.

- **<input type="text" ref={element => this.callRefInput = element} />**
- The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere.
- It can be accessed as below:
- this.callRefInput.value

Example

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class App extends React.Component {
    constructor(props) {
    super(props);
      this.callRefInput = null;
      this.setInputRef = element => {
      this.callRefInput = element;
    };
```

```
this.focusRefInput = () => {
   //Focus the input using the raw DOM API
    if (this.callRefInput) this.callRefInput.focus();
  };
 }
  componentDidMount() {
  //autofocus of the input on mount
  this.focusRefInput();
 }
  render() {
  return (
    <div>
  <h2>Callback Refs Example</h2>
      <input
        type="text"
        ref={this.setInputRef}
      />
```

```
<input
          type="button"
          value="Focus input text"
          onClick={this.focusRefInput}
        />
      </div>
    );
  }
}
export default App;
```

Note

In the above example, React will call the "ref" callback to store the reference to the input DOM element when the component mounts, and when the component unmounts, call it with null.

Refs are always up-to-date before the componentDidMount or componentDidUpdate fires.

The callback refs pass between components is the same as you can work with object refs, which is created with React.createRef().

React with useRef()

- It is introduced in React 16.7 and above version.
- It helps to get access the DOM node or element, and then we can interact with that DOM node or element such as focussing the input element or accessing the input element value.
- It returns the ref object whose .current property initialized to the passed argument.
- The returned object persist for the lifetime of the component.
- Syntax
- **const refContainer = useRef(initialValue);**
- Example
- In the below code, useRef is a function that gets assigned to a variable, inputRef, and then attached to an attribute called ref inside the HTML element in which you want to reference.

Example

```
function useRefExample() {
  const inputRef= useRef(null);
  const onButtonClick = () => {
    inputRef.current.focus();
  };
  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={onButtonClick}>Submit</button>
    </>
  );
}
```

Hooks

- Hooks are a new addition in React 16.8.
- They let developers use state and other React features without writing a class For example- State of a component
- It is important to note that hooks are not used inside the classes.

- When to use a Hooks

- If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class.
- But, now you can do it by using a Hook inside the existing function component.

Rules of Hooks

- Hooks are similar to JavaScript functions,
- Hooks rule ensures that all the stateful logic in a component is visible in its source code.
- These rules are:
- Only call Hooks at the top level
- Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions. This rule ensures that Hooks are called in the same order each time a components renders.
- Only call Hooks from React functions
- Cannot call Hooks from regular JavaScript functions. Instead, call Hooks from React function components. Hooks can also be called from custom Hooks.

React Hooks Installation

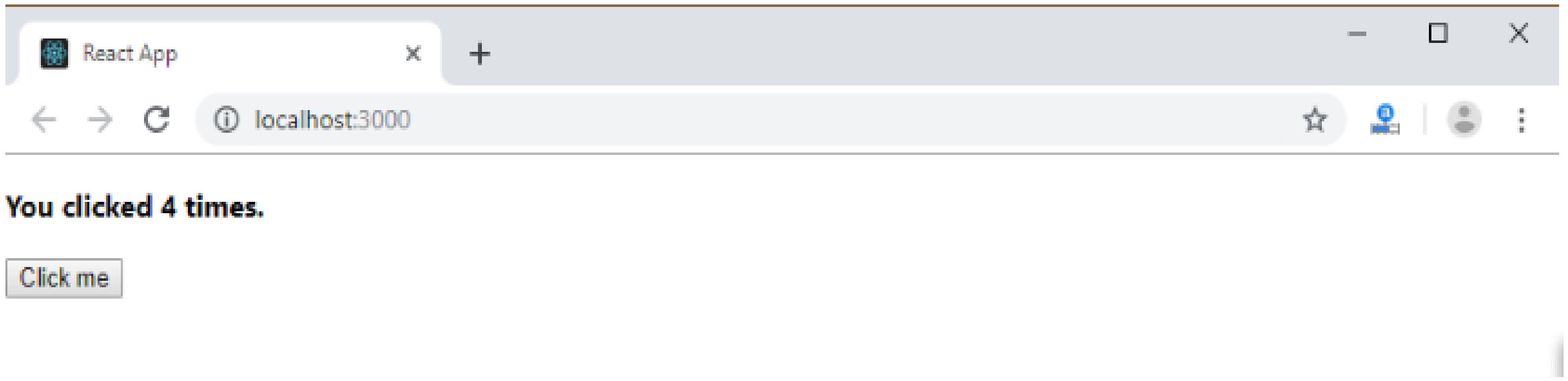
- To use React Hooks, we need to run the following commands:
- $ npm install react@16.8.0-alpha.1 --save
- $ npm install react-dom@16.8.0-alpha.1 --save
- The above command will install the latest React and React-DOM alpha versions which support React Hooks.
- *Make sure the **package.json** file lists the React and React-DOM dependencies as given below.*
- "react": "^16.8.0-alpha.1",
- "react-dom": "^16.8.0-alpha.1",

Hooks State

- Hook state is the new way of declaring a state in React app.
- Hook uses useState() functional component for setting and retrieving state.
- Let us understand Hook state with the following example.

App.js

```
import React, { useState } from 'react';
 function CountApp() {
 // Declare a new state variable, which we'll call "count"
 const [count, setCount] = useState(0);
   return (
   <div>
     <p>You clicked {count} times</p>
     <button onClick={() => setCount(count + 1)}>
       Click me
     </button>
   </div>
 );
}
export default CountApp;
```



You clicked 4 times.

- In the above example, useState is the Hook which needs to call inside a function component to add some local state to it.
- The useState returns a pair where the first element is the current state value/initial value, and the second one is a function which allows us to update it.
- After that, we will call this function from an event handler or somewhere else.
- The useState is similar to this.setState in class.

Hooks Effect

- The Effect Hook allows us to perform side effects (an action) in the function components.
- It does not use components lifecycle methods which are available in class components.
- In other words, Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.
- Side effects have common features which the most web applications need to perform, such as:
- Updating the DOM,
- Fetching and consuming data from a server API,
- Setting up a subscription, etc.

```
import React, { useState, useEffect } from 'react';
 function CounterExample() {
 const [count, setCount] = useState(0);
  // Similar to componentDidMount and componentDidUpdate:
 useEffect(() => {
  // Update the document title using the browser API
  document.title = `You clicked ${count} times`;
 });
  return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={() => setCount(count + 1)}>
    Click me
   </button>
  </div>
 );
}
export default CounterExample;
```

- In React component, there are two types of side effects:
- Effects Without Cleanup
- Effects With Cleanup

- Effects without Cleanup
- It is used in useEffect which does not block the browser from updating the screen.
- It makes the app more responsive.
- The most common example of effects which don't require a cleanup are manual DOM mutations, Network requests, Logging, etc.

Effects with Cleanup

- Some effects require cleanup after DOM updation.
- For example, if we want to set up a subscription to some external data source, it is important to clean up memory so that we don't introduce a memory leak.
- React performs the cleanup of memory when the component unmounts.
- However, as we know that, effects run for every render method and not just once.
- Therefore, React also cleans up effects from the previous render before running the effects next time.

Custom Hooks

- A custom Hook is a JavaScript function.
- The name of custom Hook starts with "use" which can call other Hooks.
- A custom Hook is just like a regular function, and the word "use" in the beginning tells that this function follows the rules of Hooks.
- Building custom Hooks allows you to extract component logic into reusable functions.

App.js

```
import React, { useState, useEffect } from 'react';

const useDocumentTitle = title => {
  useEffect(() => {
    document.title = title;
  }, [title])
}

function CustomCounter() {
  const [count, setCount] = useState(0);
  const incrementCount = () => setCount(count + 1);
  useDocumentTitle(`You clicked ${count} times`);
  // useEffect(() => {
  //   document.title = `You clicked ${count} times`
  // });
```

```
return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={incrementCount}>Click m
e</button>
    </div>
  )
}
export default CustomCounter;
```

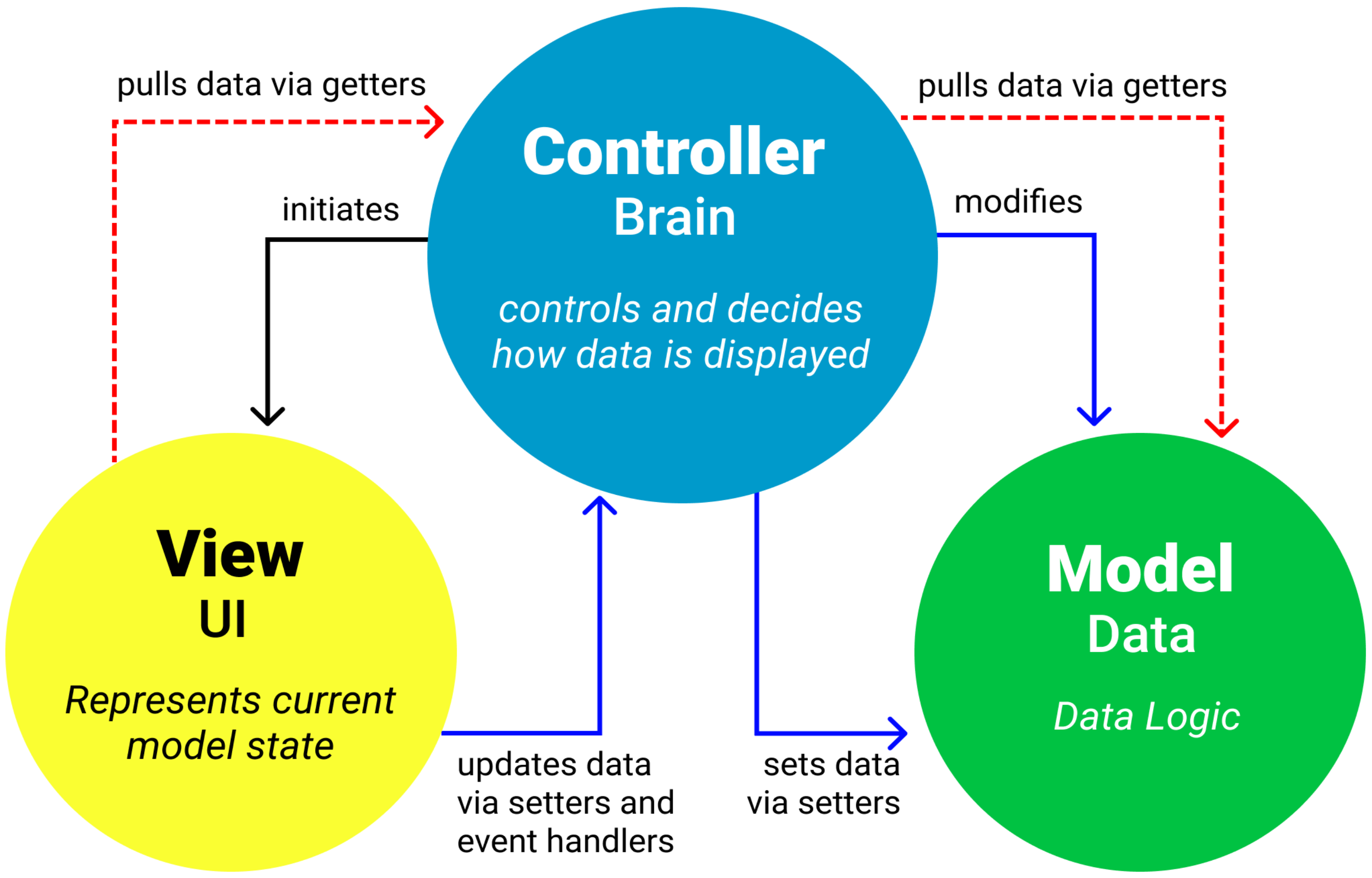
Model-View-Controller (MVC)

- The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the view, and the controller.
- Each of these components are built to handle specific development aspects of an application.
- MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.
- MVC architecture in web technology has become popular for designing web applications as well as mobile apps.

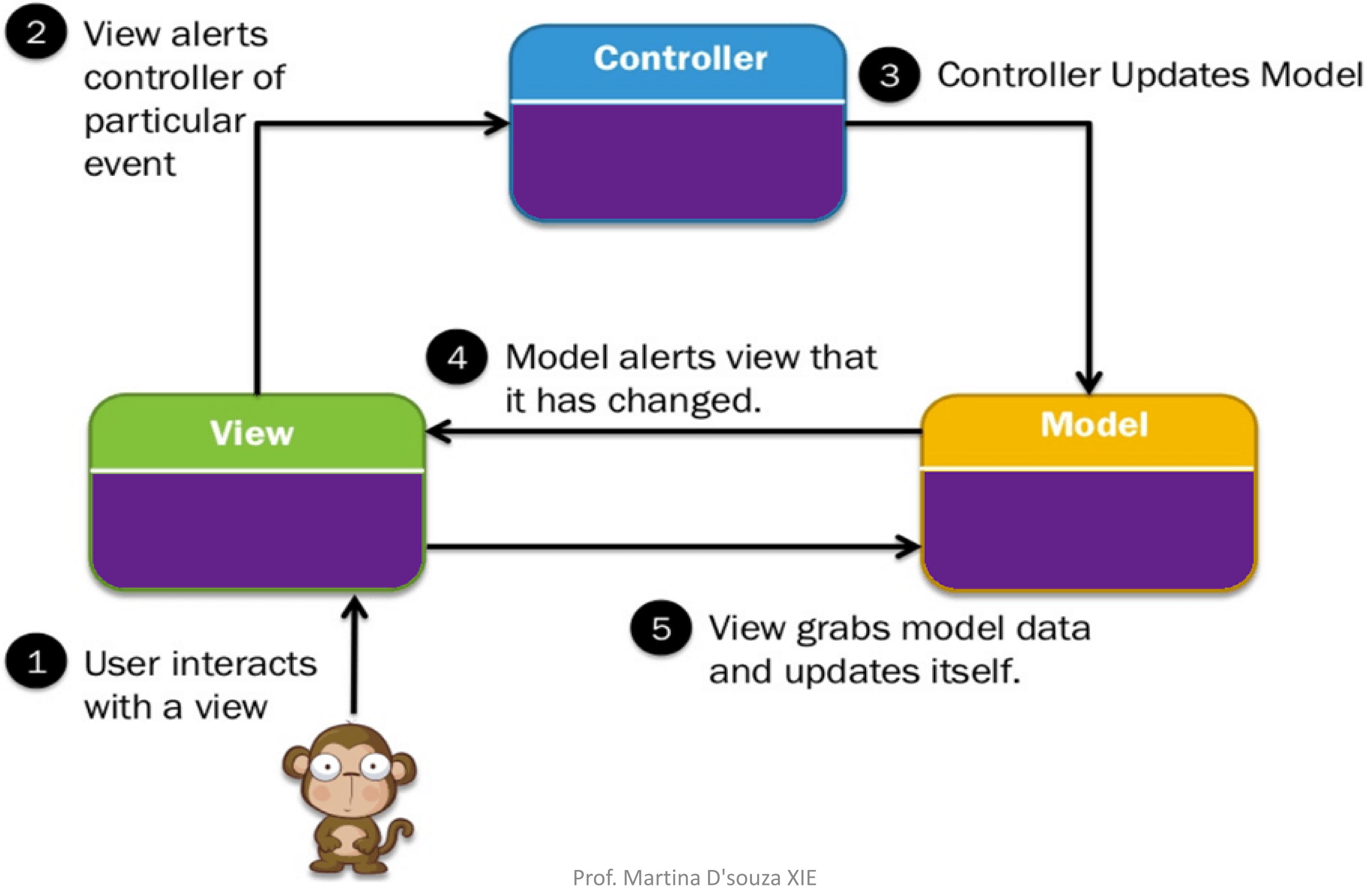
MVC

- MVC stands for model-view-controller. Here's what each of those components mean:
- **Model**: The backend that contains all the data logic
- **View**: The frontend or graphical user interface (GUI)
- **Controller**: The brains of the application that controls how data is displayed

MVC Architecture Pattern



MVC


2 View alerts controller of particular event
Controller
3 Controller Updates Model
4 Model alerts view that it has changed.
View
Model
5 View grabs model data and updates itself.
1 User interacts with a view

MVC Components

Model

- The model component stores data and its related logic.
- It represents data that is being transferred between controller components or any other related business logic.
- For example, a Controller object will retrieve the customer info from the database.
- It manipulates data and sends back to the database or uses it to render the same data.
- It responds to the request from the views and also responds to instructions from the controller to update itself.
- It is also the lowest level of the pattern which is responsible for maintaining data.

View

- A View is that part of the application that represents the presentation of data.
- Views are created by the data collected from the model data.
- A view requests the model to give information so that it resents the output presentation to the user.
- The view also represents the data from charts, diagrams, and tables.
- For example, any customer view will include all the UI components like text boxes, drop downs, etc.

Controller

- The Controller is that part of the application that handles the user interaction.
- The controller interprets the mouse and keyboard inputs from the user, informing model and the view to change as appropriate.
- A Controller send’s commands to the model to update its state(E.g., Saving a specific document).
- The controller also sends commands to its associated view to change the view’s presentation (For example scrolling a particular document).

Features of MVC

- Easy and frictionless testability. Highly testable, extensible and pluggable framework
- To design a web application architecture using the MVC pattern, it offers full control over your HTML as well as your URLs
- Leverage existing features provided by ASP.NET, JSP, Django, etc.
- Clear separation of logic: Model, View, Controller. Separation of application tasks viz. business logic, Ul logic, and input logic
- URL Routing for SEO Friendly URLs. Powerful URL- mapping for comprehensible and searchable URLs
- Supports for Test Driven Development (TDD)

Popular MVC web frameworks

Here, is a list of some popular MVC frameworks:
Ruby on Rails
Django
CakePHP
Yii
CherryPy
Spring MVC
Catalyst
Rails
Zend Framework
CodeIgniter
Laravel
Fuel PHP
Symphony

MVC Example

Controller
View
Model

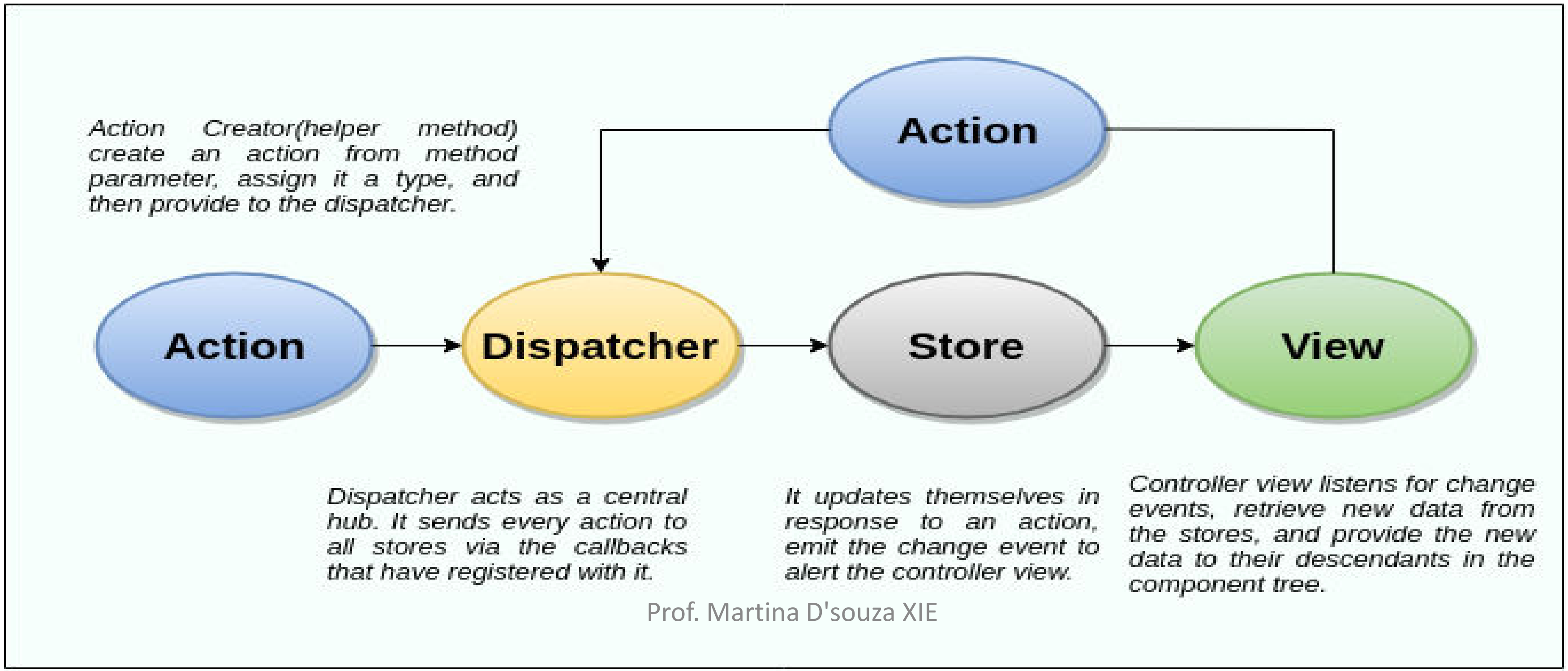
MVC Example
View: User Interface
Controller: Engine
Model: Fuel

Flux

- Flux is an architecture that Facebook uses internally when working with React.
- It is *not* a framework or a library.
- It is simply a new kind of architecture that complements React and the concept of Unidirectional Data Flow.
- This data enters the app and flows through it in one direction until it is rendered on the screen.
- It is useful when the project has dynamic data, and we need to keep the data updated in an effective manner.
- It reduces the runtime errors.

FLUX

- Flux applications have three major roles in dealing with data:
- Dispatcher
- Stores
- Views (React components)



Action Creator(helper method) create an action from method parameter, assign it a type, and then provide to the dispatcher.

Dispatcher acts as a central hub. It sends every action to all stores via the callbacks that have registered with it.

It updates themselves in response to an action, emit the change event to alert the controller view.

Controller view listens for change events, retrieve new data from the stores, and provide the new data to their descendants in the component tree.

FLUX

- In Flux application, data flows in a single direction(unidirectional).
- This data flow is central to the flux pattern.
- The dispatcher, stores, and views are independent nodes with inputs and outputs.
- The actions are simple objects that contain new data and type property.

Dispatcher

- It is a central hub for the React Flux application and manages all data flow of your Flux application.
- It is a registry of callbacks into the stores.
- It has no real intelligence of its own, and simply acts as a mechanism for distributing the actions to the stores.
- All stores register itself and provide a callback.
- It is a place which handled all events that modify the store.
- When an action creator provides a new action to the dispatcher, all stores receive that action via the callbacks in the registry.

Dispatcher's API has five methods

- Methods

1. **register():** It is used to register a store's action handler callback.
2. **unregister():**It is used to unregisters a store's callback.
3. **waitFor():**It is used to wait for the specified callback to run first.
4. **dispatch():**It is used to dispatches an action.
5. **isDispatching():**It is used to checks if the dispatcher is currently dispatching an action.

Stores

- It primarily contains the application state and logic.
- It is similar to the model in a traditional MVC.
- It is used for maintaining a particular state within the application, updates themselves in response to an action, and emit the change event to alert the controller view.

Views

- It is also called as controller-views.
- It is located at the top of the chain to store the logic to generate actions and receive new data from the store.
- It is a React component listen to change events and receives the data from the stores and re-render the application.

Actions

- The dispatcher method allows us to trigger a dispatch to the store and include a payload of data, which we call an action.
- It is an action creator or helper methods that pass the data to the dispatcher.

Advantage of Flux

- It is a unidirectional data flow model which is easy to understand.
- It is open source and more of a design pattern than a formal framework like MVC architecture.
- The flux application is easier to maintain.
- The flux application parts are decoupled.

Flux Vs MVC

SN	MVC	FLUX
1.	It was introduced in 1976.	It was introduced just a few years ago.
2.	It supports Bi-directional data Flow model.	It supports Uni-directional data flow model.
3.	In this, data binding is the key.	In this, events or actions are the keys.
4.	It is synchronous.	It is asynchronous.
5.	Here, controllers handle everything(logic).	Here, stores handle all logic.
6.	It is hard to debug.	It is easy to debug because it has common initiating point: Dispatcher.
7.	It is difficult to understand as the project size increases.	It is easy to understand.
8.	Its maintainability is difficult as the project scope goes huge.	Its maintainability is easy and reduces runtime errors.