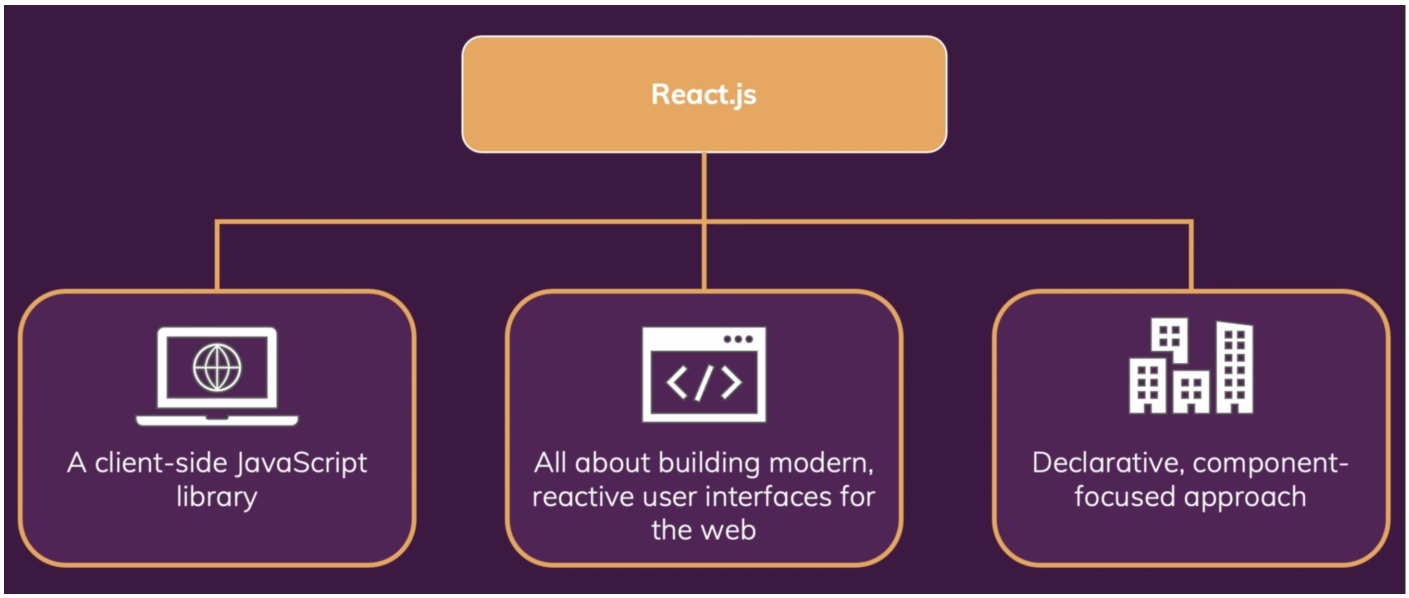
React

React is a modern Javascript library for building user interfaces.

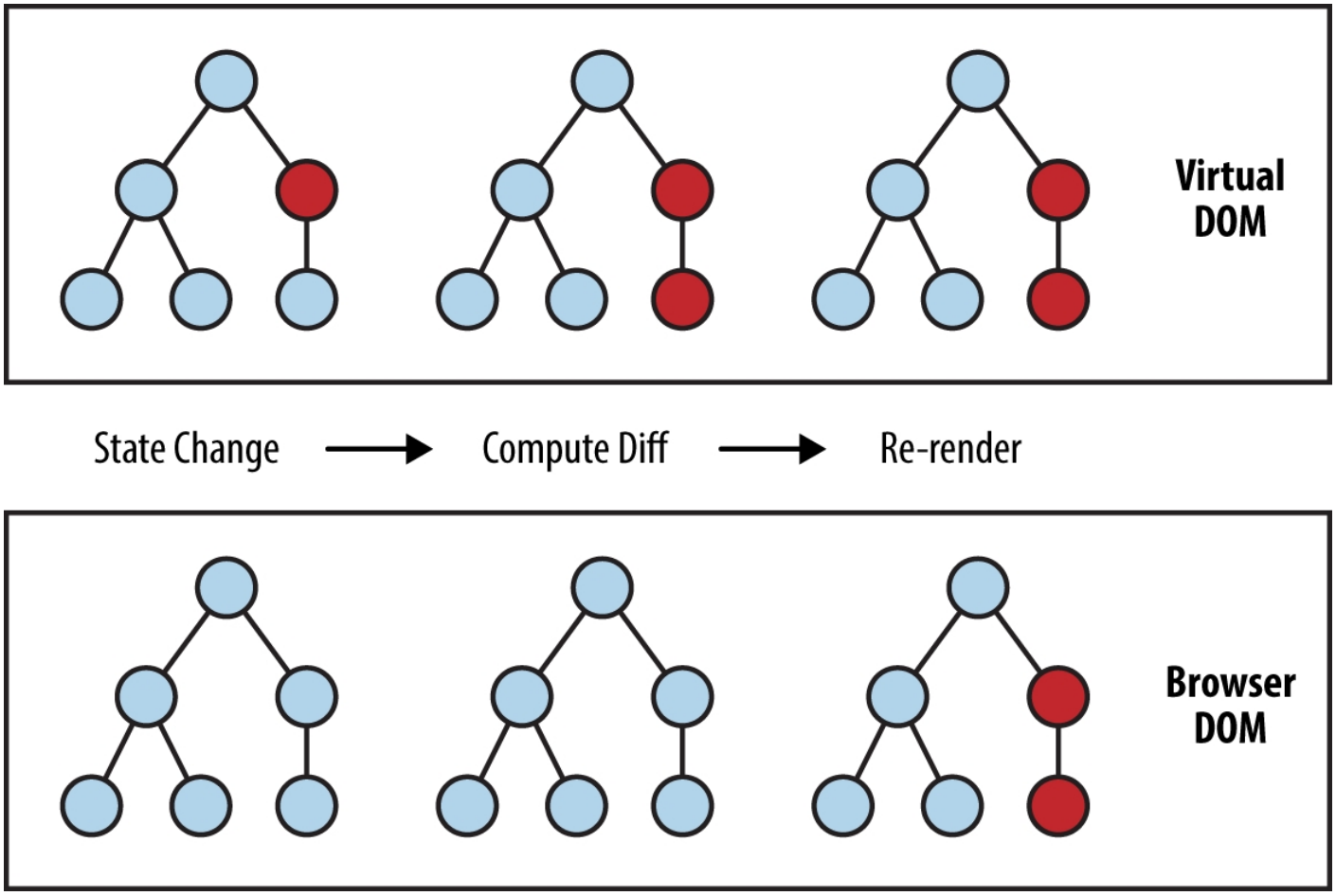
- React makes it easier to create modern interactive UIs. React adheres to the declarative programming paradigm, you can design views for each state of an application and React will update and render the components when the data changes. This is different than imperative programming (Javascript).
- Component-Based: Components are the building blocks of React, they allow you to split the UI into independent, reusable pieces that work in isolation. Data can be passed through components using "props".



React Virtual DOM

When new elements are added to the UI, a virtual DOM, which is represented as a tree, is created. Each element is a node on this tree. If the state of any of these elements changes, a new virtual DOM tree is created. This tree is then compared or "diffed" with the previous virtual DOM tree.

Once this is done, the virtual DOM calculates the best possible method to make these changes to the real DOM. This ensures that there are minimal operations on the real DOM. Hence, reducing the performance cost of updating the real DOM.



The red circles represent the nodes that have changed. These nodes represent the UI elements that have had their state changed. The difference between the previous version of the virtual DOM tree and the current virtual DOM tree is then calculated. The whole parent subtree then gets re-rendered to give the updated UI. This updated tree is then batch updated to the real DOM.

JSX

JSX is an XML-like syntax extension to ECMAScript without any defined semantics.

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called “components” that contain both.

```
const element = <h1>Hello, world!</h1>;
```

In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces:

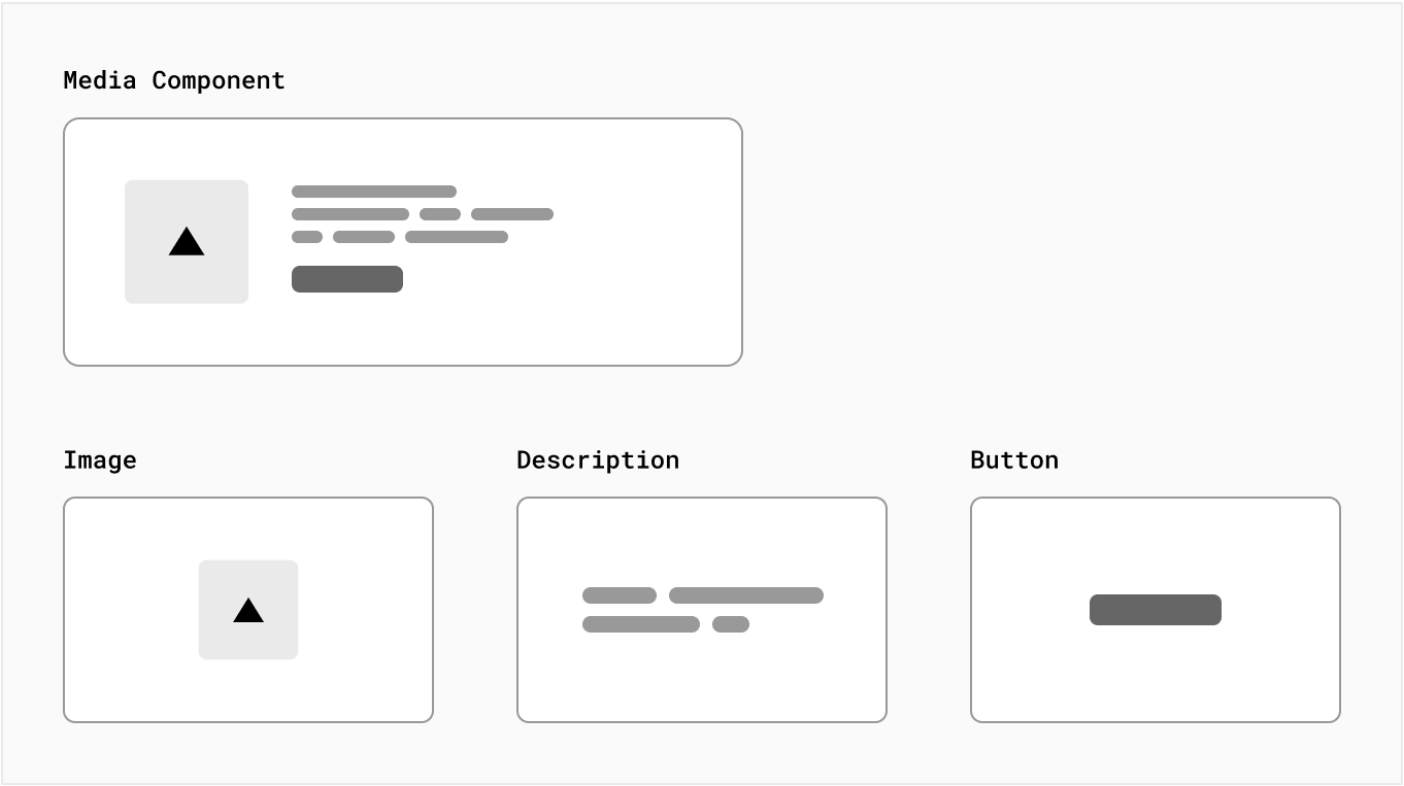
```
const name = 'Andrei';
const element = <h1>Hello, {name}</h1>;
```

Components

User interfaces can be broken down into smaller building blocks called components.

Components allow you to build self-contained, reusable snippets of code. If you think of components as LEGO bricks, you can take these individual bricks and combine them together to form larger structures. If you need

to update a piece of the UI, you can update the specific component or brick.



This modularity allows your code to be more maintainable as it grows because you can easily add, update, and delete components without touching the rest of our application. The nice thing about React components is that they are just JavaScript.

In React, components are functions or classes, we are only going to use functions. Inside your script tag, write a function called header:

```
export default function StartupEngineering() {
  return <div>StartupEngineering</div>;
}
```

Nesting Components

Applications usually include more content than a single component. You can nest React components inside each other like you would do with regular HTML elements.

The <Header> component is nested inside the <HomePage> component:

```
function Header() {
  return <h1>Startup Engineering</h1>;
}

function HomePage() {
  return (
    <div>
      {/* Nesting the Header component */}
      <Header />
    </div>
  );
}
```

Props

Similar to a JavaScript function, you can design components that accept custom arguments (or props) that change the component's behavior or what is visibly shown when it's rendered to the screen. Then, you can pass down these props from parent components to child components. Props are **read only**.

In your HomePage component, you can pass a custom title prop to the Header component, just like you'd pass HTML attributes. Here, titleSE is a custom argument (prop).

```
function HomePage({titleSE}) {
  return (
```

```
    <div>
      <Header title={titleSE} />
    </div>
  );
}
```

Conditional Rendering

We'll create a Greeting component that displays either of these components depending on whether a user is logged in:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please sign up.</h1>;
}
```

We can also use inline if-else operators.

When sending props from parent to child component, we can use **object destructuring** . Destructuring really shines in React apps because it can greatly simplify how you write props.

```
function Header({ title }) {
  return <h1>{title ? title : 'Default title'}</h1>;
}
```

Iterating through lists

It's common to have data that you need to show as a list. You can use array methods to manipulate your data and generate UI elements that are identical in style but hold different pieces of information.

```
function HomePage() {
  const names = ['POLI', 'ASE', 'UNIBUC'];

  return (
    <div>
      <Header title="Universities" />
      <ul>
        {names.map((name) => (
          <li key={name}>{name}</li>
        ))}
      </ul>
    </div>
  );
}
```

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.

The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys.

```
const universitiesItems = universities.map((uni) =>
  <li key={uni.id}>
    {uni.name}
  </li>
);
```

Adding Interactivity with State

React has a set of functions called hooks. Hooks allow you to add additional logic such as state to your components. You can think of state as any information in your UI that changes over time, usually triggered by user interaction.

You can use state to store and increment the number of times a user has clicked the like button. In fact, this is what the React hook to manage state is called: useState()

useState() returns an array. The first item in the array is the state value, which you can name anything. It’s recommended to name it something descriptive. The second item in the array is a function to update the value. You can name the update function anything, but it's common to prefix it with set followed by the name of the state variable you’re updating. The only argument to useState() is the initial state.

```
function HomePage() {
  // ...
  const [likes, setLikes] = React.useState(0);

  function handleClick() {
    setLikes(likes => likes + 1);
  }

  return (
    <div>
      {/* ... */}
      <button onClick={handleClick}>Likes ({likes})</button>
    </div>
  );
}
```

Data Binding

Data Binding is the process of connecting the view element or user interface, with the data which populates it.

In ReactJS, components are rendered to the user interface and the component’s logic contains the data to be displayed in the view(UI). The connection between the data to be displayed in the view and the component’s logic is called data binding in ReactJS.

```
export default function HomePage() {
  const [inputField, setInputField] = useState('');

  function handleChange(e) {
      setInputField(e.target.value);
  }

  return (
      <div>
          <input value={inputField} onChange={handleChange}/>
          <h1>{inputField}</h1>
      </div>
  );
}
```

Passing data from child to parent component

In react data flows only one way, from a parent component to a child component, it is also known as one way data binding. While there is no direct way to pass data from the child to the parent component, there are workarounds. The most common one is to pass a handler function from the parent to the child component that accepts an argument which is the data from the child component. This can be better illustrated with an example.

```
const Parent = () => {
  const [message, setMessage] = React.useState("Hello World");
  const chooseMessage = (message) => {
    setMessage(message);
  };
  return (
    <div>
      <h1>{message}</h1>
      <Child chooseMessage={chooseMessage} />
    </div>
  );
};
const Child = ({ chooseMessage }) => {
  let msg = 'Goodbye';
  return (
    <div>
      <button onClick={() => chooseMessage(msg)}>Change    Message</button>
    </div>
  );
};
```

The initial state of the message variable in the Parent component is set to ‘Hello World’ and it is displayed within the h1 tags in the Parent component as shown. We write a chooseMessage function in the Parent component and pass this function as a prop to the Child component. This function takes an argument message. But data for this message argument is passed from within the Child component as shown. So, on click of the Change Message button, msg = ‘Goodbye’ will now be passed to the chooseMessage handler function in the Parent component and the new value of message in the Parent component will now be ‘Goodbye’. This way, data from the child component(data in the variable msg in Child component) is passed to the parent component.