Chapter 1 :Learn React

Describing the UI

React is a JavaScript library for rendering user interfaces (UI). UI is built from small units like buttons, text, and images.
React lets you combine them into reusable, nestable components. From web sites to phone apps, everything on the screen can be broken down into components.

create, customize, and conditionally display React components.

1. How to write your first React component
2. When and how to create multi-component files
3. How to add markup to JavaScript with JSX
4. How to use curly braces with JSX to access JavaScript functionality from your components
5. How to configure components with props
6. How to conditionally render components
7. How to render multiple components at a time
8. How to avoid confusing bugs by keeping components pure
9. Why understanding your UI as trees is useful

1.Your first component

React applications are built from isolated pieces of UI called components. A React component is a JavaScript function that you can sprinkle with markup. Components can be as small as a button, or as large as an entire page. Here is a Gallery component rendering three Profile components:

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
```

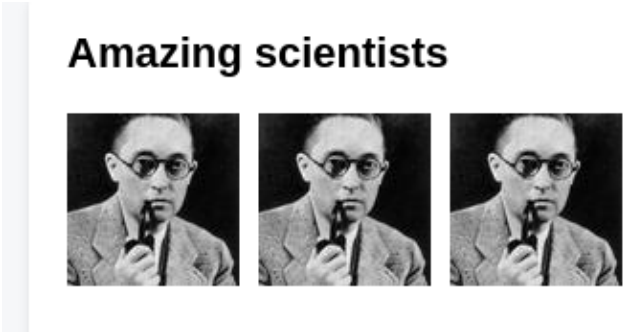
```
    </section>
  );
}
```

2.Importing and exporting components

You can declare many components in one file, but large files can get difficult to navigate. To solve this, you can export a component into its own file, and then import that component from another file:

```
import Profile from './Profile.js';

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```



Gallery.js

```
export default function Profile() {
  return (
    <img
      src="https://i.imgur.com/QIrZWGIs.jpg"
      alt="Alan L. Hart"
    />
  );
}
```

Profile,.js

3.Writing markup with JSX

Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information.

If we paste existing HTML markup into a React component, it won’t always work:

App.js

```
export default function TodoList() {
  return (
    // This doesn't quite work!
    <h1>Hedy Lamarr's Todos</h1>
    <img
      src="https://i.imgur.com/yXOvdOSs.jpg"
      alt="Hedy Lamarr"
      class="photo"
    >
    <ul>
      <li>Invent new traffic lights
      <li>Rehearse a movie scene
      <li>Improve spectrum technology
    </ul>
  );
}
```

Error
/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (5:4)

```
  3 |     // This doesn't quite work!
  4 |     <h1>Hedy Lamarr's Todos</h1>
> 5 |     <img
    |     ^
  6 |       src="https://i.imgur.com/yXOvdOSs.jpg"
  7 |       alt="Hedy Lamarr"
  8 |       class="photo"
```

If you have existing HTML like this, you can fix it using a converter:

```
export default function TodoList() {
  return (
    <>
      <h1>Hedy Lamarr's Todos</h1>
      <img
        src="https://i.imgur.com/yXOvdOSs.jpg"
        alt="Hedy Lamarr"
        className="photo"
      />
      <ul>
        <li>Invent new traffic lights</li>
```

```
      <li>Rehearse a movie scene</li>
      <li>Improve spectrum technology</li>
    </ul>
  </>
 );
}
```

Read Writing Markup with JSX to learn how to write valid JSX.

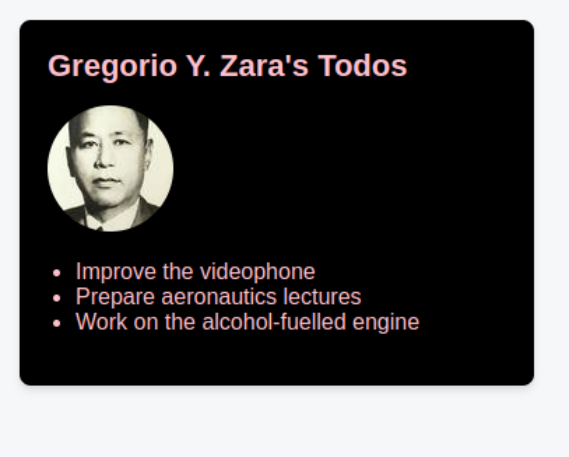
4. JavaScript in JSX with curly braces

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to “open a window” to JavaScript:

App.js

```
const person = {
 name: 'Gregorio Y. Zara',
 theme: {
  backgroundColor: 'black',
  color: 'pink'
 }
};

export default function TodoList() {
 return (
  <div style={person.theme}>
   <h1>{person.name}'s Todos</h1>
   <img
    className="avatar"
    src="https://i.imgur.com/7vQD0fPs.jpg"
    alt="Gregorio Y. Zara"
   />
   <ul>
    <li>Improve the videophone</li>
    <li>Prepare aeronautics lectures</li>
    <li>Work on the alcohol-fuelled engine</li>
   </ul>
  </div>
 );
}
```



Read JavaScript in JSX with Curly Braces to learn how to access JavaScript data from JSX.

5. Passing props to a component

React components use props to communicate with each other.
Every parent component can pass some information to its child components by giving them props.
Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, functions, and even JSX!

Util.js

```
export function getImageUrl(person, size = 's') {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    size +
    '.jpg'
  );
}
```

App.js

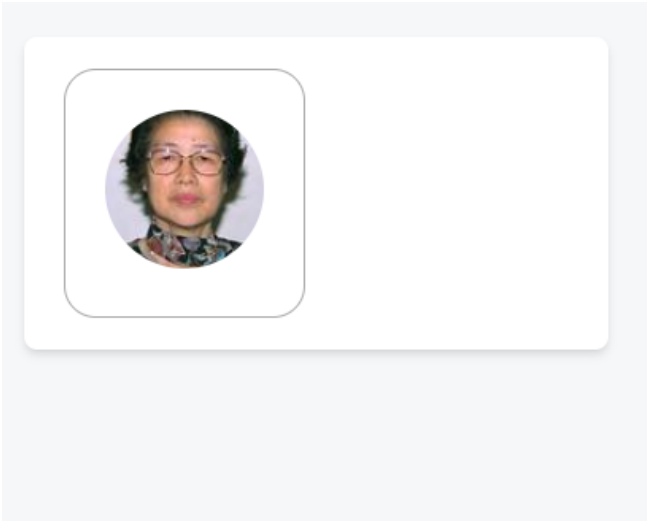
```
import { getImageUrl } from './utils.js'

export default function Profile() {
  return (
    <Card>
      <Avatar
        size={100}
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'YfeOqp2'
        }}
      />
    </Card>
  );
}

function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}

function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
```

```
  );
}
```



Read Passing Props to a Component to learn how to pass and read props.

6.Conditional rendering

Your components will often need to display different things depending on different conditions.

In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.

```
function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && '     '}
    </li>
  );
}
```

```
export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Read Conditional Rendering to learn the different ways to render content conditionally.

7. Rendering lists

You will often want to display multiple similar components from a collection of data. You can use JavaScript's filter() and map() with React to filter and transform your array of data into an array of components.

For each array item, you will need to specify a key. Usually, you will want to use an ID from the database as a key. Keys let React keep track of each item's place in the list even if the list changes.

```
mport { people } from './data.js';
import { getImageUrl } from
'./utils.js';

export default function List() {
  const listItems =
people.map(person =>
    <li key={person.id}>
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}:</b>
        {' ' + person.profession + '
'}
        known for
{person.accomplishment}
      </p>
    </li>
  );
  return (
    <article>
      <h1>Scientists</h1>
      <ul>{listItems}</ul>
    </article>
  );
}
```

App.js

```
export const people = [{
  id: 0,
  name: 'Creola Katherine
Johnson',
  profession: 'mathematician',
  accomplishment: 'spaceflight
calculations',
  imageId: 'MK3eW3A'
}, {
  id: 1,
  name: 'Mario José Molina-
Pasquel Henríquez',
  profession: 'chemist',
  accomplishment: 'discovery of
Arctic ozone hole',
  imageId: 'mynHUSa'
}, {
  id: 2,
  name: 'Mohammad Abdus
Salam',
  profession: 'physicist',
  accomplishment:
'electromagnetism theory',
  imageId: 'bE7W1ji'
}, {
  id: 3,
  name: 'Percy Lavon Julian',
  profession: 'chemist',
  accomplishment: 'pioneering
cortisone drugs, steroids and
birth control pills',
  imageId: 'IOjWm71'
}, {
  id: 4,
  name: 'Subrahmanyan
Chandrasekhar',
  profession: 'astrophysicist',
  accomplishment: 'white dwarf
star mass calculations',
  imageId: 'lrWQx8l'
}];
```

Data.js

```
export function
getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    's.jpg'
  );
}
```

Util.js

Read Rendering Lists to learn how to render a list of components, and how to choose a key.

Keeping components pure
Some JavaScript functions are pure. A pure function:

- ➔ **Minds its own business. It does not change any objects or variables that existed before it was called.**
- ➔ **Same inputs, same output. Given the same inputs, a pure function should always return the same result.**
- ➔
- ➔ **By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. Here is an example of an impure component:**

```
function Cup() {
  // Bad: changing a preexisting variable!
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
      <Cup />
    </>
  );
}
```

```
function Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup guest={1} />
      <Cup guest={2} />
      <Cup guest={3} />
    </>
  );
}
```

Read Keeping Components Pure to learn how to write components as pure, predictable functions.

8. Your UI as a tree

React uses trees to model the relationships between components and modules.

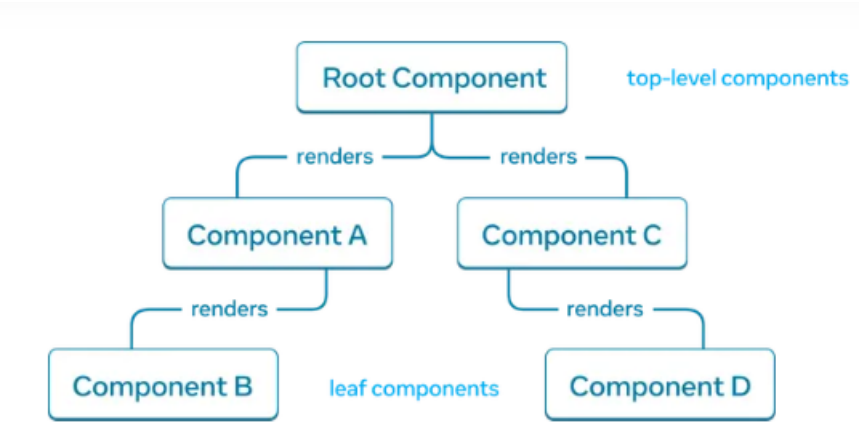
A React render tree is a representation of the parent and child relationship between components.

A tree graph with five nodes, with each node representing a component. The root node is located at the top the tree graph and is labelled 'Root Component'. It has two arrows extending down to two nodes labelled 'Component A' and 'Component C'. Each of the arrows is labelled with 'renders'.

'Component A' has a single 'renders' arrow to a node labelled 'Component B'. 'Component C' has a single 'renders' arrow to a node labelled 'Component D'.
An example React render tree.

Components near the top of the tree, near the root component, are considered top-level components. Components with no child components are leaf components. This categorization of components is useful for understanding data flow and rendering performance.

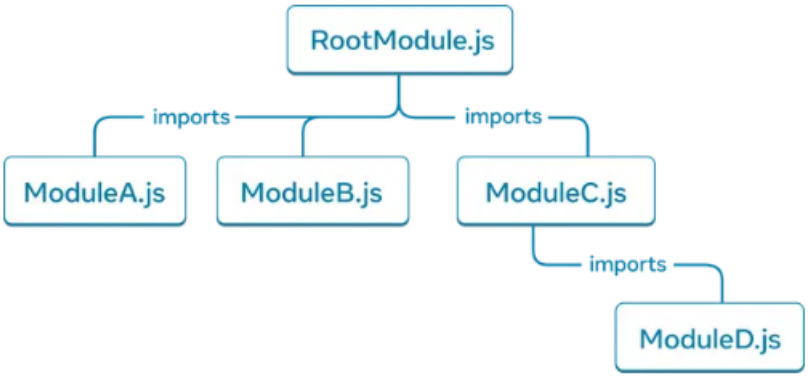
Modelling the relationship between JavaScript modules is another useful way to understand your app. We refer to it as a module dependency tree.



An example React render tree.

A tree graph with five nodes. Each node represents a JavaScript module. The top-most node is labelled 'RootModule.js'. It has three arrows extending to the nodes: 'ModuleA.js', 'ModuleB.js', and 'ModuleC.js'. Each arrow is labelled as 'imports'. 'ModuleC.js' node has a single 'imports' arrow that points to a node labelled 'ModuleD.js'.
An example module dependency tree.

A dependency tree is often used by build tools to bundle all the relevant JavaScript code for the client to download and render. A large bundle size regresses user experience for React apps. Understanding the module dependency tree is helpful to debug such issues.



An example module dependency tree.

CHAPTER 2

Importing and Exporting Components

The magic of components lies in their reusability: you can create components that are composed of other components. But as you nest more and more components, it often makes sense to start splitting them into different files. This lets you keep your files easy to scan and reuse components in more places.

You will learn

- What a root component file is
- How to import and export a component
- When to use default and named imports and exports
- How to import and export multiple components from one file
- How to split components into multiple files

2.1 The root component file

In Your First Component, you made a `Profile` component and a `Gallery` component that renders it:

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
```

```
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

These currently live in a **root component file,** named `App.js` in this example. Depending on your setup, your root component could be in another file, though. If you use a framework with file-based routing, such as Next.js, your root component will be different for every page.

2.2 Exporting and importing a component

What if you want to change the landing screen in the future and put a list of science books there? Or place all the profiles somewhere else? It makes sense to move `Gallery` and `Profile` out of the root component file. This will make them more modular and reusable in other files. You can move a component in three steps:

1. **Make** a new JS file to put the components in.
2. **Export** your function component from that file (using either default or named exports).
3. **Import** it in the file where you'll use the component (using the corresponding technique for importing default or named exports).

Here both `Profile` and `Gallery` have been moved out of `App.js` into a new file called `Gallery.js`. Now you can change `App.js` to import `Gallery` from `Gallery.js`:

```
import Gallery from './Gallery.js';

export default function App() {
  return (
    <Gallery />
  );
}
```

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/QIrZWGIs.jpg"
      alt="Alan L. Hart"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Notice how this example is broken down into two component files now:

1. `Gallery.js`:
    - Defines the `Profile` component which is only used within the same file and is not exported.
    - Exports the `Gallery` component as a **default export.**
2. `App.js`:
    - Imports `Gallery` as a **default import** from `Gallery.js`.
    - Exports the root `App` component as a **default export.**

Note

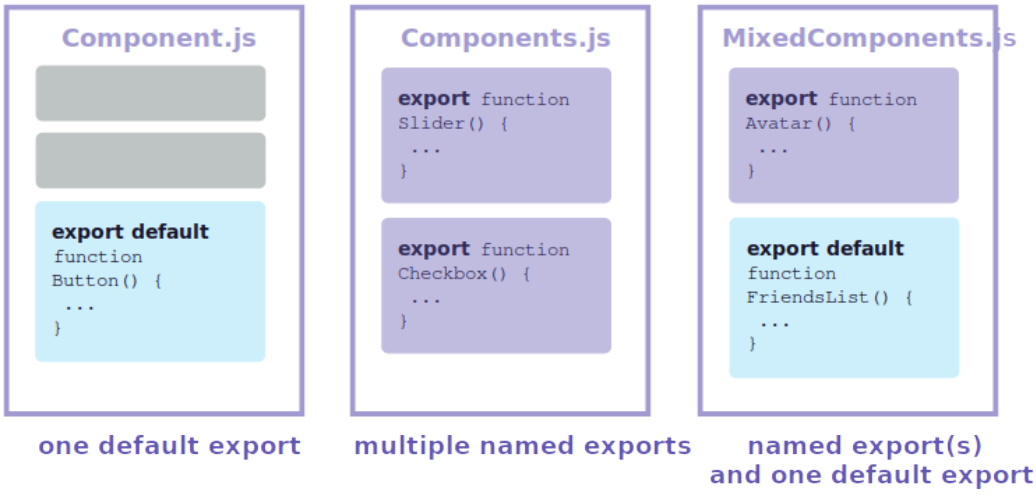
You may encounter files that leave off the .js file extension like so:

```
import Gallery from './Gallery';
```

Either './Gallery.js' or './Gallery' will work with React, though the former is closer to how native ES Modules work.

2.3. Default vs named exports

There are two primary ways to export values with JavaScript: default exports and named exports. So far, our examples have only used default exports. But you can use one or both of them in the same file. **A file can have no more than one** *default* **export, but it can have as many** *named* **exports as you like**



How you export your component dictates how you must import it. You will get an error if you try to import a default export the same way you would a named export! This chart can help you keep track:

Syntax	Export statement	Import statement

Default	export default function Button() {}	import Button from './Button.js';
Named	export function Button() {}	import { Button } from './Button.js';

When you write a default import, you can put any name you want after import. For example, you could write import Banana from './Button.js' instead and it would still provide you with the same default export. In contrast, with named imports, the name has to match on both sides. That's why they are called named imports!

People often use default exports if the file exports only one component, and use named exports if it exports multiple components and values. Regardless of which coding style you prefer, always give meaningful names to your component functions and the files that contain them. Components without names, like export default () => {}, are discouraged because they make debugging harder.

2.4 Exporting and importing multiple components from the same file

What if you want to show just one `Profile` instead of a gallery? You can export the `Profile` component, too. But `Gallery.js` already has a *default* export, and you can't have *two* default exports. You could create a new file with a default export, or you could add a *named* export for `Profile`. **A file can only have one default export, but it can have numerous named exports!**

2.5 How to split components into multiple file

sTo reduce the potential confusion between default and named exports, some teams choose to only stick to one style (default or named), or avoid mixing them in a single file. Do what works best for you!

First, **export** `Profile` from `Gallery.js` using a named export (no `default` keyword):

```
export function Profile() {
  // ...
}
```

Then, **import** `Profile` from `Gallery.js` to `App.js` using a named import (with the curly braces):

```
import { Profile } from './Gallery.js';
```

Finally, **render** `<Profile />` from the `App` component:

```
export default function App() {
  return <Profile />;
}
```

Now `Gallery.js` contains two exports: a default `Gallery` export, and a named `Profile` export. `App.js` imports both of them. Try editing `<Profile />` to `<Gallery />` and back in this example:

```
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

export default function App() {
return (
<Profile />
);
}
```

App.js

```
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

export default function App() {
  return (
    <Profile />
  );
}
```

Now you're using a mix of default and named exports:

- `Gallery.js`:
  - Exports the `Profile` component as a **named export called `Profile`.**
  - Exports the `Gallery` component as a **default export.**
- `App.js`:
  - Imports `Profile` as a **named import called `Profile`** from `Gallery.js`.
  - Imports `Gallery` as a **default import** from `Gallery.js`.
  - Exports the root `App` component as a **default export.**

Try out some challenges

Challenge 1 of 1:

Split the components further

Currently, `Gallery.js` exports both `Profile` and `Gallery`, which is a bit confusing.

Move the `Profile` component to its own `Profile.js`, and then change the `App` component to render both `<Profile />` and `<Gallery />` one after another.

You may use either a default or a named export for `Profile`, but make sure that you use the corresponding import syntax in both `App.js` and `Gallery.js`! You can refer to the table from the deep dive above:

Syntax	Export statement	Import statement
Default	`export default function Button() {}`	`import Button from './Button.js';`
Named	`export function Button() {}`	`import { Button } from './Button.js';`

```
// Move me to Profile.js!
export function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```