Enzyme:

Enzyme is a JavaScript Testing utility for React that makes it easier to test your React Components' output. You can also manipulate, traverse, and in some ways simulate runtime given the output. Enzyme's API is meant to be intuitive and flexible by mimicking jQuery's API for DOM manipulation and traversal.

Along with Enzyme there are other ways through which you can test web applications like cypress, react testing framework created by Facebook, and many others. Enzyme has various advantages that help developers to test their work easily.

Enzyme is a tool that creates a virtual DOM which will eliminate the need of using a browser to test your work. You can also simulate different events on your UI components. Enzyme has two concepts when it comes to rendering a virtual DOM, **shallow rendering** and **mount**.

Enzyme has this concept of **shallow rendering** which allows you to render components only one level deep, i.e. if you have child components in a parent component, it will put placeholders for child components in order to provide quicker tests. However **mount** is a contrary concept to this where Enzyme renders all parents and children in DOM.

Enzyme also provides access to props and states so you can see and examine them as a part of your testing.

To use enzyme in your project you need to install following packages:

- **enzyme** - **jest-enzyme** - **enzyme-adapter-react-16**

Following are the types of tests that you can do as a developer in you project:

1. **Unit tests**: they are about testing one piece of code.
2. **Integration tests**: they tell how multiple units works
3. **Acceptance or E2E tests**: the purpose of these tests are to demonstrate how a user would interact with the app. Often you use selenium/cypress to run on an actual browser and see interactions work from e2e, i.e. from ui to database operations.

Testing goals:

There are many intentions or goals behind testing a project:

- **Easy to maintain tests**: Test behavior, not implementation
- **Easy Diagnosis of failing tests**: Test state (implementation) after each user interaction.

We always need to keep these goals in mind so that we don't make our job complex. Sometimes we apply too many tests for checking the state or implementation of our application which makes it complicated for us and other developers in the team. So we can always try to use a balance of both and write tests accordingly.

Snapshot testing: The concept behind snapshot testing is that we provide a structure or frozen code of component to test, the testing framework renders the component and then we compare the rendered structure with the given structure or frozen code to see if they match. Snapshot testing is usually avoided because it does not support test driven development approaches and breaks frequently.

Testing using Enzyme:

1. Simple UI Tests:

If you are working on a very simple web application then you apply selectors on the html tags to see if they render. E.g. I have a div in my component I can apply a selector on it in this way:

```
<div data-test="component-app" className="App">
```

Afterwards in my code, I will shallow render the component and test if it renders successfully like this:

```
test('renders without error', () => {
   const wrapper = setup();
   const appComponent = findByTestAttr(wrapper, 'component-app');
   expect(appComponent.length).toBe(1);
  });
```

In this way you can write your tests and simulate different events like inserting texts, clicking etc.

2. Test Driven development:

You can also follow test driven development on your UI projects, the way to do it is by writing your test first in your test file with steps, e.g. first render the component, find a heading for counter in wrapper, expect its value is 0, then find the add button, click the add button, expect the value of counter heading to be 1 now. Once you are done with your test case you can start writing functionality for it in your project. You will start by creating a header and a button in your component with the text and properties that you have already asserted or are expecting in your test case. Once the UI part is done you are gonna have to write a click function for the button to increment the counter.

In this way you have implemented the desired functionality on the basis of your test case. You can follow this approach for other complex functionalities like a text field which would have some validations and various behaviors on the input provided to it.

3. **Redux Testing**:

Redux is a useful tool in many applications where state management becomes complicated due to involvement of multiple components. Some people prefer to skip unit testing on redux actions and reducers and just do the integration testing on API to see how the redux store reacts to that. You can write tests for reducers and actions and it is a good approach as you test the implementation of reducers and actions. Adding the tests for those will help ease diagnosing test failures when functionality updates.

Another thing that we test in these cases are the components which are connected to the state. We can test the behavior of components when a certain action type is passed in their functionalities. In those cases you are gonna need to create a mock store configuration that mimics your actual redux store and you can pass that to the component which you will render. In this case Enzyme recommends you to use the `dive()` method to get to the child component in the wrapper. In case of non connected components, they are further from the root component so you can pass mock action creators as props.

4. **Async Actions testing in Axios**:

A server is necessary for an actual app and from there we are getting the data for our app. But we don't want to test the server, we want to test asynchronous calls made by our application. So that when our tests fail we will know that it is because of our implementation and not due to server response or connection. So in this case we use moxios without running the server or testing it. Axios sends requests to moxios instead of sending them to http, the test then specifies what moxios should respond with. Action creator calls axios which then calls moxios instead of http and returns a response to axios, which is received by the action creator which started the operation. You can install moxios in your test using `moxios.install()`. An example of this would be:

```
moxios.wait(( ) => {
      const request = moxios.requests.mostRecent();
      request.respondWith({
            status: 200
            response: <Any response object or data>
      });
});
```

In some test cases we may need to test both the api calls and action creators like if we are testing a component which contains an api call and calls the action creator afterwards.