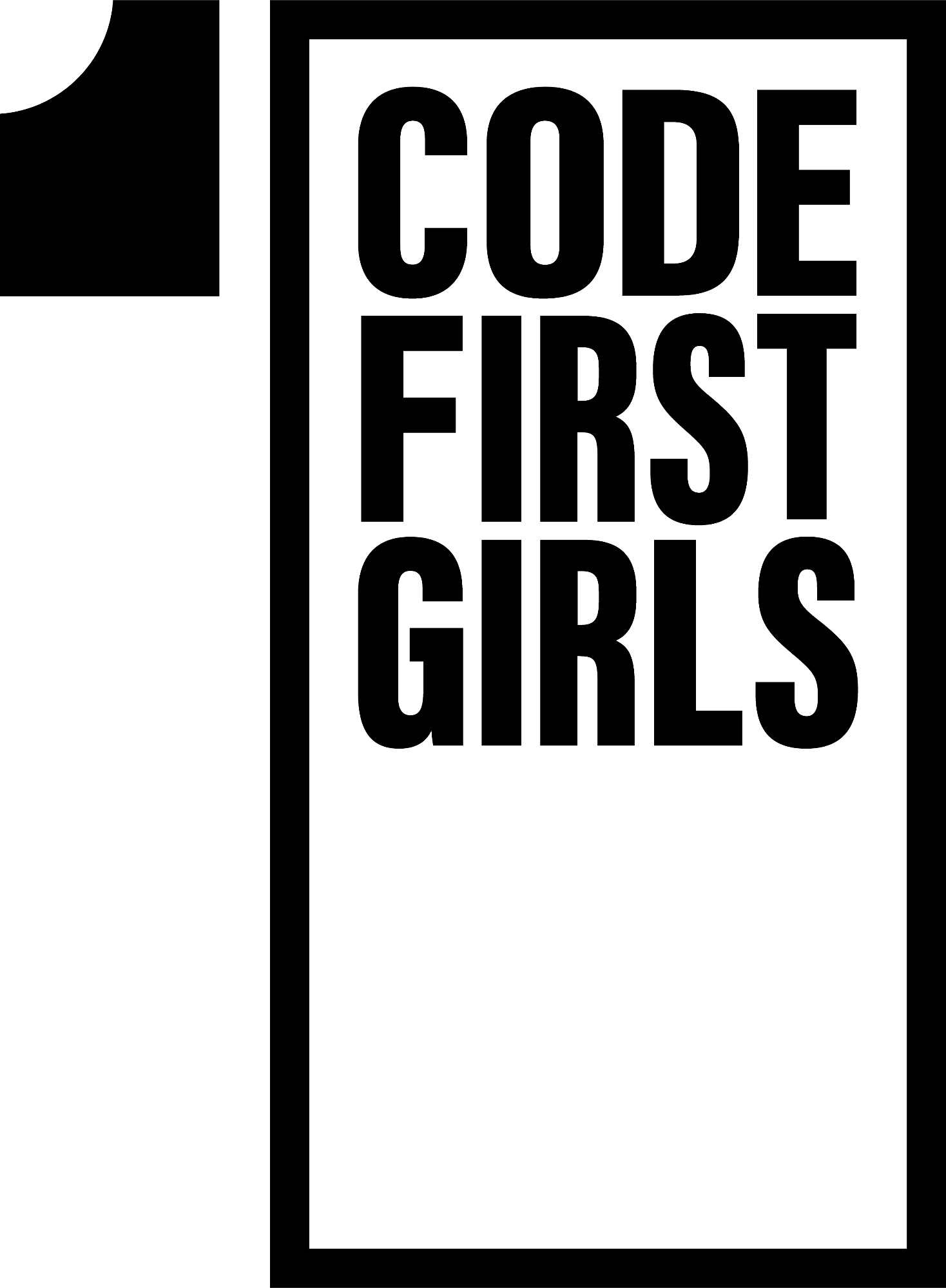
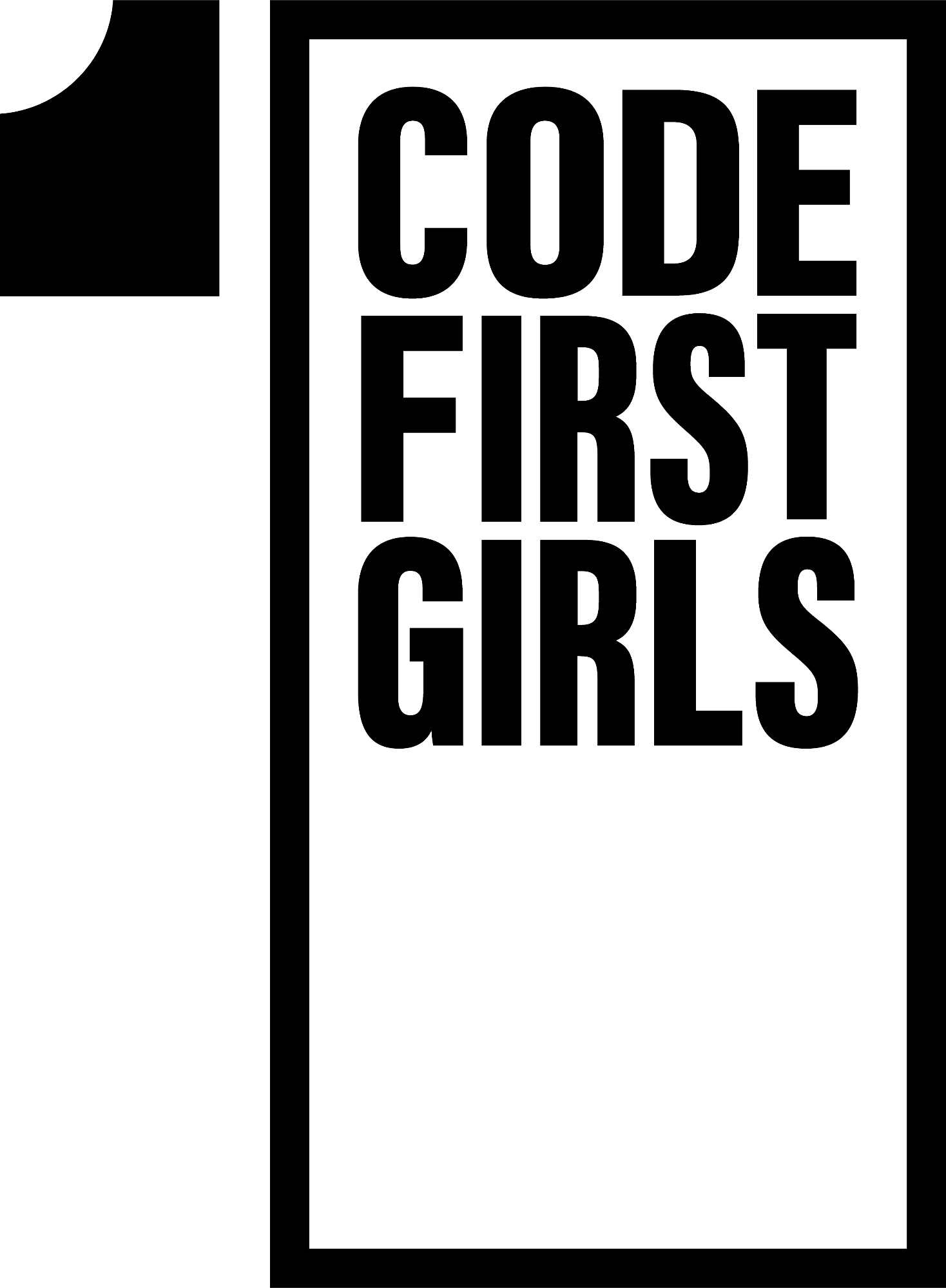


**FULLSTACK ASSESSMENT MATERIAL RELEASE**

THEORY QUESTIONS



|  |  |  |
| --- | --- | --- |
| **SECTION TYPE** | **TOTAL MARKS AVAILABLE** | **NOTES** |
| **Design heuristics** | 10 |  |
| **Redux** | 10 |  |
| **React** | 10 |  |
| **Agile** | 10 |  |
|  | **40 marks available total** |  |

**Important notes:**

* This document shares the first section of the FullStack Assessment which is composed of 4 FullStack Theory Questions
* You have 24 hours before the assessment to prepare.
* If any plagiarism is found in how you choose to answer a question you will receive a 0 and the instance will be recorded. Consequences will occur if this is a repeated offence. You can remind yourself of the plagiarism policy [here](https://drive.google.com/file/d/1k9UaGOR7hx54QRZ8jvp2jtC4P-8_Rs4F/view?usp=sharing).
* Answers need to be explained clearly and illustrated with relevant examples where necessary. Your examples can include code snippets, diagrams or any other evidence-based representation of your answer.

*`Questions begin on the next page*

1. In design Heuristics, what does the term “advantages of Matching between system and the real world” mean? What are the advantages?

In design heuristics, the term "advantages of Matching between system and the real world" is a principle that attempts to design systems or interfaces which closely resemble the real-world tasks, concepts, and interactions that users are familiar with. It is one of the 10 Nielsen’s usability Heuristics. The goal is to make the system as intuitive and easy to understand as possible for the intended users, by leveraging their already-existing knowledge and mental models to reduce the cognitive load required to memorize or learn in order to interact with unfamiliar design/concepts.

The advantages of matching between the system and the real world include 5 key areas:

1. familiarity (when system design matches users’ existing experiences and mental models it is familiar and therefore easier to use, enabling the transfer of skills and understanding of whatever new system you’re introducing, and decreasing the learning curve). This is often referred to as natural mapping, such as using user-focused language and design rather than system centered language and design (in programmer language this is what is best for customers and users not what is best/easiest for programmers);

2. ease-of-use (your system is intuitive as it is built to reflect real-world conventions and interactions also known as Skeuomorphism; again, users can rely on their existing knowledge to interact with the system more efficiently and effectively, it reduces the need for training or documentation) an example of this would be the ‘click and drag function’ to be able to copy files across in apple macs as opposed to windows being more system-orientated where you have to go through several right-click pathways to paste files across. Another example of good Skeuomorphism is how in the West green is associated with go and red with stop, utilising these colours to mean such on a website would intrinsically play to a user’s real world understandings, increasing the ease of use (if the context was western, in China red is a lucky colour so these are culturally specific phenomenons, which the designing team would need to be aware of depending on the target audience.);

3. reduction in cognitive load (no need to learn new concepts or terminology reduces cognitive load and allows users to focus on the tasks at hand rather than spending time and energy trying to figure out how new complex interfaces or interactions work) - – in its simplest terms, such as intuitively knowing a door is ‘push’ by design, rather than having to remember that even though it looks like a pull door it is in fact push.

4. preventing errors (building on the familiarity of the new system, it becomes easier for users to anticipate the outcomes of their actions, which reduces the likelihood of errors or accidental misuse);

5. user satisfaction (an easy-to-use and understand system leads to increased satisfaction; user feel more confident and in control, which enhances their overall user experience).

In conclusion, the aim and benefit of matching between the system and the real world in design heuristics is creating intuitive and user-friendly interfaces that leverage users' existing knowledge and mental models. By minimizing the learning curve and cognitive load, designers can enhance user satisfaction and usability while reducing the potential for errors.

|  |  |
| --- | --- |
| **Theory Questions** | 1. **otal** |

2. What do you understand by “Single source of truth”? and how does it relate to redux? What are the advantages ?

In order to answer this question, it is important to first define the phrase "Single source of truth", which refers to a principle in software development that promotes having a centralized and authoritative data source within an application. It means that there should be a single, well-defined, and consistent representation of data that serves as the sole reference for the entire system. This ensures that all parts of the application have access to the same, up-to-date information, eliminating data inconsistencies and conflicts.

In the context of Redux (a popular state management library for JavaScript), the concept of "Single source of truth" is closely related. Redux employs the principle by maintaining a single immutable state tree that represents the entire application's state. This state tree is stored in a central store and can only be modified by dispatching actions, which are processed by pure functions called reducers.

By adhering to the principle of a single source of truth, Redux simplifies state management and promotes a more organized, scalable, and predictable architecture for applications. It improves consistency, debugging capabilities, and separation of concerns, ultimately leading to more robust and maintainable codebases.

There are several key aadvantages of using a "Single source of truth" with Redux, such as:

Consistency (as when there is a single source of truth, all components within the application can access the same data, which eliminates inconsistencies that arise from multiple copies of data stored in different parts of the application, ensuring that everyone is working with the most up-to-date information).

Predictability (as Redux enforces a strict unidirectional data flow, where data changes occur through dispatched actions and are processed by reducers, this predictability simplifies debugging and understanding how state changes occur within the application, as all modifications are captured in a single location which protects the code from corruption although of course cannot ever eliminate all risk).

Scalability (as having a single source of truth makes it easier to manage and scale the application's state, where maintaining a centralized store allows for better organization and management of the state, especially as it increases in complexity, ensuring that it remains manageable and maintainable).

Debugging and time travel (as Redux provides powerful debugging capabilities, including the ability to record and replay actions, which facilitates easy time travel through the application's state history, which is immensely helpful in reproducing and debugging issues, as well as providing a mechanism for undo/redo functionality), and sseparation of concerns (as with Redux, components can focus solely on rendering the state and dispatching actions, while the state management logic resides in reducers; this separation of concerns enhances modularity and maintainability, as components become more reusable and easier to reason about).

1. What is the difference between a stateless component and a stateful component in React?

In order to answer this question, it is important to define what state is in the context of react: state is an object that belongs to the component that has declared it; therefore, its range of influence is limited to that current component. Said component that initialise its ‘state’ or update it within certain parameters (i.e., whether stateless or stateful). When the state extends out of its range of influence on the current component it becomes a prop or ‘input’ for other components. The state of the parent component usually becomes the prop (or optional input ) of the child component.

The stateless react component is the normal or expected react component, defined as a function that does not manage any state, hence ‘stateless’ and relies on props to have their data changed. It does not initialise classes, there are hooks or constructors needed. a stateless function component when given a set of props will always render the same exact JSX, therefore this code is easier to test as it will have fewer unexpected outputs. As it is simpler, more straightforward code, if offers better readability which makes it easier to add in additional components at a later date as needed when updating the code, it is often used for data that has been imported usually by the user, that will then be subject to change within the code. Stateless components do not keep track of the changes made to the data in the code. Therefore, you do not need an in-depth working knowledge of the render () facility to get your code to run. An example of a stateless component is a functional component. Some use cases for stateless components are when you want to run/write reusable code that is applicable elsewhere, when you don’t need any internal variables or state in your code, when you don’t need your code to be interactive.

Stateful components have a state, which gets initialised by the constructor and remembers changes made to the component’s state in the memory. It can get changed depending on the action of the original parent components or the children’s components. A stateless component can only render props but a stateful component can render props and states. Unlike stateless components, stateful components can use lifecycle methods such as componentDidMount() and ComponentWillMount(). The component frequently re-renders based on changes to its state and can pass down these changes from parent to child components as properties within the prop(s). An example of a stateful component is a class component often using a hook. Example of use cases for a stateful component are when creating an element that required user input or is within an interactive page.

1. List out the advantages and disadvantages of exploratory testing (used in Agile) and scripted testing?

Both exploratory testing and scripted testing have their strengths and weaknesses. In practice, a combination of both approaches can be beneficial, allowing testers to leverage the advantages of each method while mitigating their disadvantages. The breakdown of these is listed below.

The advantages of exploratory testing (used in Agile) include:

Flexibility (as it allows testers to adapt and explore the application based on their intuition and expertise, and for spontaneous test design and execution, enabling testers to uncover unexpected issues or scenarios).

Adaptability (as testers can quickly respond to changes in requirements or user feedback, because exploratory testing doesn't rely heavily on predefined test cases, which allows for on-the-fly adjustments and exploration of different test scenarios), and to ‘test as they go’.

Creativity (as exploratory testing encourages testers to think creatively and come up with new test ideas, which may uncover defects that might not be found through scripted testing alone).

Lastly, early defect detection (as exploratory testing can identify defects early in the development process since it can be performed as soon as there is a testable piece of functionality available, which addresses issues promptly, reducing the overall cost and impact of defects).

On the other hand , the disadvantages of exploratory testing are:

A lack of repeatability (since exploratory testing doesn't follow a predefined script or set of test cases, it can be challenging to reproduce the exact steps taken during testing, which makes it harder to track and document the exact sequence of actions and results for future reference).

Coverage concerns (as without predefined test cases, there is a risk that important functionality or edge cases may be overlooked; because it relies so heavily on tester expertise, exploratory testing may lead to biased coverage or overlook critical areas of the application).

And subjectivity (as the outcome of exploratory testing can be influenced by the tester's personal biases, assumptions, or familiarity with the system, this may introduce a level of subjectivity in the testing process, which can lead to inconsistent results across different testers).

For scripted testing, the advantages are largely the opposite and include:

Consistency (as scripted testing provides a structured and repeatable approach, ensuring that the same tests are executed consistently, which maintains consistent quality standards across different testing sessions or testers),

Coverage (as scripted testing allows for comprehensive coverage of predefined test cases, ensuring that all critical functionality and edge cases are tested, which provides a systematic approach to ensure that no important aspect of the system is missed).

Documentation (as scripted tests can serve as documentation for future reference, they provide a clear record of the test steps, expected results, and actual results, making it easier to reproduce and track issues encountered during testing),

And training and onboarding (as scripted testing can be useful for training new testers or team members, because predefined scripts provide a structured approach that helps newcomers understand the application's functionality and expected behaviour).

However, the disadvantages of this approach are:

Rigidity (as scripted testing may not adapt well to changing requirements or evolving functionality, and it can be time-consuming to update and maintain a large number of scripts when there are frequent changes or additions to the system).

Limited creativity (as scripted testing focuses on following predefined steps and expected results, leaving little room for testers to explore beyond those boundaries, which can lead to overlooking unexpected scenarios or edge cases that may be discovered through exploratory testing).

And it being time-consuming (as creating and maintaining scripted test cases can be time-consuming, especially for complex applications or when there are frequent updates, the effort required to keep test scripts up to date can be a challenge in fast-paced Agile environments).

So, in essence the 2 approaches are polar opposites from each other, offering countering pros and cons, so developers need to decide which approach better suits their projects, teams, and working styles.