THEORY QUESTIONS ASSIGNMENT

Full Stack Stream
(Maximum Score: 100)

KEY NOTES

- This assignment is to be completed at the student's own pace and submitted before the given deadline.
- There are **8** questions in total and each question is marked on a scale 1 to 20. The maximum possible grade for this assignment is 100 points.
- Students are welcome to use any online or written resources to answer these questions.
- The 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.

Theory questions	**Points allocated per Question**

1. What is React? (*E.g. Consider: what is it? What is the benefit of using it? What is its virtual DOM? Why would someone choose it over the standard HTML / CSS stack?)*(15 marks)

   React is an open-source, free front-end JavaScript library. It allows building interactive user interfaces for websites and mobile apps. It's been created by Jordan Walke, a software engineer at Facebook (now Meta), and first used for Facebook News Feed in 2011, to then build Instagram timeline in 2012.

   There are multiple benefits of using React, bringing it to the forefront of the app development. Some of them include:

   - in React one creates separate UI components, that become building blocks of an application. They are reusable, which saves a lot of development time;
   - React uses Virtual DOM (virtual representation of UI kept in a memory). It compares it to the Real DOM, and then updates only items that were changed. This improves performance of the site;

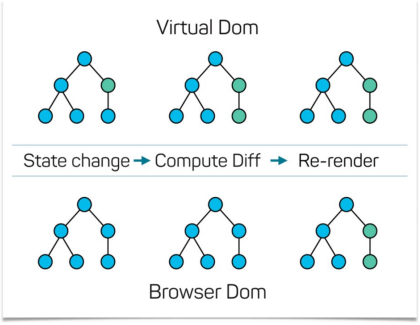

Figure 1: Virtual DOM, source: https://mfrachet.github.io/create-frontend-framework/vdom/intro.html

- for people coming from JavaScript background React is easy to learn, shortening the development process, hence being often chosen to build projects;
- in React the flow of data is unidirectional (from parent to children), which not only provides better code stability, but also makes debugging easier;
- it has strong community support and is constantly developed, there is a lot of free tutorials, as well as GitHub repositories related to React;
- due to its quick page loading time and faster rendering, it is SEO friendly, allowing the application to rank higher in search engine results page;
- thanks to JSX (JavaScript syntactic expression) feature, HTML structures can be written in the same file as JavaScript code;
- applications written in React are easy to test.

All that makes React a library that allows for creating dynamic, fast-loading, user friendly, easy to maintain applications, with a lot of libraries of ready-made components that can speed up app development, which outranks having HTML/CSS only applications.

2. What are Props? What is State? What is the difference between them? (10 marks)

Props (aka properties) are arguments passed into React components. It's important to remember that data flow in React is unidirectional, so that it flows from parent to child only. Data from props is read-only and cannot be modified by a component that is receiving it from the outside.

Example:

```
import React from 'react';

function Welcome(props){
 return <h1>Hello, {props.name}</h1>;
}

function App(){
 return(
  <div>
   <Welcome name="Anna" />
   <Welcome name="Joanna" />
   <Welcome name="Zuzanna" />
  </div>
 );
}

export default Welcome;
```

State is a special built-in object. It that allows components to create and manage their own data. Unlike props, components cannot pass data with state, but they can create and manage it internally. State should not be modified directly, but it can be modified with method called setState().

Example of state:

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

function Example() {
 const [count, setCount] = useState(0);

 return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={() => setCount(count + 1)}>
    Click me
   </button>
  </div>
 );
}

export default Example;
```

To recap previously noted differences between props and state:

- with props, components receive data from outside, with state, they can create and manage their own data;
- props pass data, state manages data;
- data from props is read-only and cannot be modified by a component that receives it, state data can be modified by its component, but it's private (not accessible from outside).

3. What are React Hooks? How do they differ from existing lifecycle methods? (10 marks)

There are three phases in a lifecycle of a component: mounting (when a new component is created and inserted into DOM), updating (when component updates or re-renders) and unmounting (when component is removed from DOM).

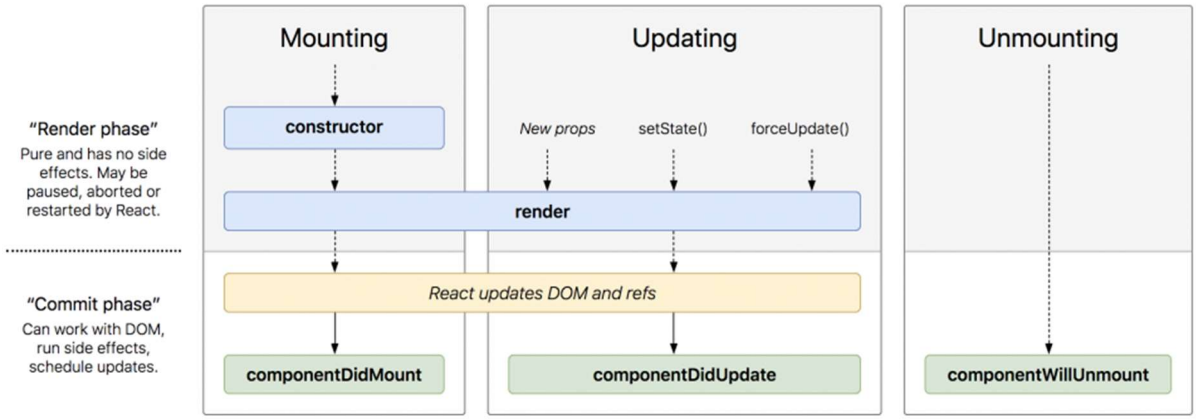

Figure 2: Lifecycle methods diagram, source: https://github.com/wojtekmaj/react-lifecycle-methods-diagram

To be able to handle various side effects (like fetching data on mount, sanitizing props when the component updates, cleaning up before the component unmounts) it is necessary to reach for lifecycle methods. Before React version 16.8, they could only be handled using class components and all functional components that wanted to access

them had to be rewritten as a class. Also state of the application could be accessed from class components only. This led to adding complexity to a code (making it harder to read and understand) and reduced code reusability (lifecycle methods must be a member of the class and so can't be extracted).

And so, the Hooks have been introduced. They allow to access lifecycle methods and state from within functional component. They allowed to:

- extract stateful logic from a component, so that it can be reused and tested independently;
- simplify complex components by breaking them into smaller functions based on their purpose;
- encourage using functional components that might be easier to understand for both people and machines.

Example of hooks in use:

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

function Example() {
 const [count, setCount] = useState(0);

 useEffect(() => {
  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 Example;
```

The same counter in a class component:

```
import React from 'react';

class Example extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   count: 0
  };
 }

 componentDidMount() {
  document.title = `You clicked ${this.state.count} times`;
 }
 componentDidUpdate() {
  document.title = `You clicked ${this.state.count} times`;
 }

 render() {
  return (
   <div>
    <p>You clicked {this.state.count} times</p>
    <button onClick={() => this.setState({ count: this.state.count + 1 })}>
     Click me
    </button>
   </div>
  );
 }
}

export default Example;
```

It's easy to see that the syntax differs and that useEffect hook combines componentDidMount and componentDidUpdate methods. It's also possible to pass a function within useEffect hook, and it will be invoked only when the component is removed from the DOM (making it equal to componentWillUnmount method).

Hooks and functional components are now a recommended way of writing React code, and some of the lifecycle methods previously used in class components have been deprecated and are currently to be prefixed with UNSAFE – for example UNSAFE_componentWillMount(). Despite all that there are no plans of removing class components and some lifecycle methods are still available only via class (like the ones connected with Error Boundaries).

4. Design the perfect door - what should it look like, what are the components for it? What design heuristics should it follow, and how does your design match? What made you choose this design? (20 marks).

    a. *Consider in particular (likely need to do independent learning): who are your stakeholders? What are their personas? What are the doors requirements and how will your stakeholders benefit from your solution?*

To better visualise this task, I have prepared two additional documents: Door_design.pdf and Stakeholder_personas.pdf. They have been created in draw.io. Following Norman's heuristic, I have marked them in bold in relevant to them sections.

My door will be wide enough for strollers and wheelchairs to pass through (900mm wide, which is a standard for wheelchairs [***when all else fails, standardise***]), which will make it easy to navigate for both Timothy and Gabriella. For the same reason it will not have a threshold. Transparent panels will help informing Timothy (and other customers) about availability of seats, without the need to open the door to check [***make thigs visible***].

Door will have a long lever that will make it easy to open and won't cause any pain for Rupert *[**use both the knowledge in the world and in the head***]. With the lever being on the right-hand side, when coming in and pushing the door, customers will be using their left hand (which, for majority of people, is the weaker one). When coming out and having to pull the door, the right, stronger hand will be engaged (that can help with swinging the door open for wheelchair/stroller users) [***getting the mappings right***].

Door will be constructed from composite materials, making it lightweight yet durable. The foam core will offer a great insulation, so it's going to be energy efficient. It will be semi-automated – it will open like a normal door would and can be closed as such [***simplify the structure of the task***], but if someone forgets to close it and it stays open for a specific period, it will close automatically [***design for error***] – that will help Diana to keep the bills lower.

From the outside, the protruded casing will immediately inform that the door should be pushed. Visible hinges and overlay of the door plate over the frame on the inside will show that the door has to be pulled [***exploit the power of constraints, both natural and artificial***].

5. What is Angular, and how does it differ from React? *You may need to conduct independent research and learning for this* (10 marks)

Angular is open-source, free web framework, written in Typescript, created and maintained by Google. It was first released in 2010 under the name AngularJS (Angular 1.0). Thanks to its features, such as two-way data binding and dependency injections, it became extremely popular. In September 2016 it got completely revamped and released as Angular (or Angular 2.0). Current version is 14 and was released in June 2022.

Differences between Angular and React include:

- Angular is a complete framework that doesn't usually need additional libraries, whilst React is a library that focuses on building UI components and usually requires installing extra modules or libraries to perform some functions;
- Angular uses two-way data binding method (also supporting one-way): if UI input is modified, the model state will change (also works the other way); React uses one-way data binding only, which means that UI elements can't change component's state;
- Angular is based on Typescript and React is based on JavaScript;

- Angular uses MVC (Model View Controller), and React uses Virtual DOM

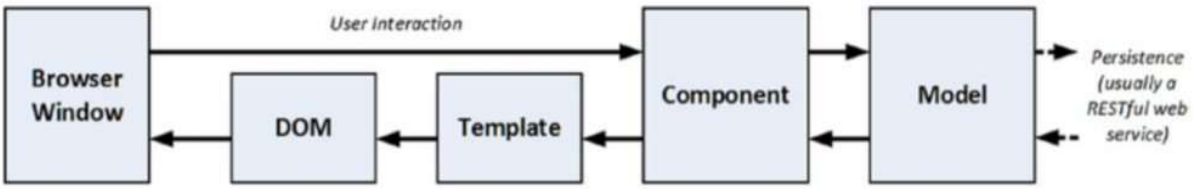

Figure 3: MVC Angular model, source: https://wuschools.com/what-is-mvc-and-understanding-the-mvc-pattern-in-angular/

- React will be quite simple to understand and learn for people with JavaScript background, whilst Angular is more complex, robust and requires more learning time;
- the structure of Angular is fixed, based on three layers (Model, Controller and View), and each component is written in four separate files (TypeScript to implement a component, HTML file to define the view, CSS for styling and a testing file); React offers View layer only (Model and Controller are added with other libraries) and doesn't have specific structure – it has to be designed for each application individually.

6. Please describe Redux in as much detail - especially consider: *why would someone use it? What is it? What's the benefit of using it? Are there any potential drawbacks to using it? How can it be added to a project? What is dispatch, provider, actions, etc?* (15 marks)

   Redux is a predictable state container, which means that the state of the application is kept in a store, and all components can access any state needed for them to operate. It was created by Dan Abramov and Andrew Clark, and initially released in June 2015. Currently main maintainers are Mark Ericson and Tim Dorr.

   Redux is mostly used as a state management tool with React, but it can be used with any JavaScript library or framework.

   In React, whenever one component needs to share state with another that it doesn't have parent-child relation to and can't pass it as props, the state has to be lifted up to the nearest parent component and to the next, until it reaches the right one. Redux helps to solve this issue.

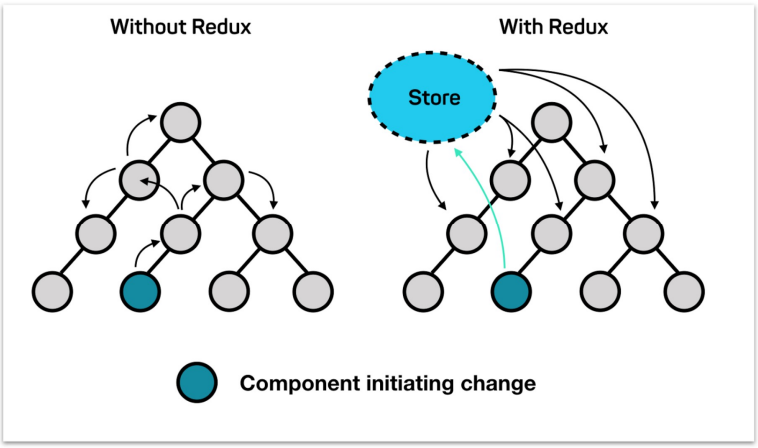

Figure 4: Redux, source: https://blog.codecentric.de/en/2017/12/developing-modern-offline-apps-reactjs-redux-electron-part-3-reactjs-redux-basics/

   There are three building parts of Redux:

- **Actions**: a way to send data from application to Redux store. Data can be collected from user interaction, form submissions, API calls etc. They are send using store.dispatch() method and are JavaScript objects that must have type property (indicating what type of action to perform) and payload (information that should be worked on by action). They are created by action creator (functions). For example:

```
function actionCreator(text){
return { type: 'SOME_ACTION', payload: text }
}

store.dispatch(actionCreator('sample payload'))
```

- **Store**: holds the application state. The recommendation is to keep only one store in any Redux application (this is its intended pattern) for increased reliability, speed and easier debugging. Any action performed on a state will always return a new state, making it predictable. In React, <Provider> component will make Redux store accessible to any nested component (most applications will render Provider at the top level, with the whole component tree of app inside it);
- **Reducers**: functions, which take current state of the application, perform an action, and return a new state. Reducers must always:
    - calculate the new state based on *state* and *action* arguments;
    - make immutable updates by copying the existing state and making changes to the copied values;
    - refrain from doing any asynchronous logic, calculating random values or causing other side effects

Example reducer:

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action){
 if (action.type === 'counter/incremented'){
  return {
   ...state,
   value: state.value + 1
  }
 }
 return state
}
```

Some benefits of using Redux include:

- ease of sharing state between components;
- predictability of state (the same state&action will always return the same result);
- ease of maintenance (the strict rules of code organisation make it easier to understand and maintain);
- ease of debugging (because of logging actions and state, it's easy to track the errors);
- it is a lightweight solution so doesn't add to the size of application;
- because of using functions, it is easy to test;

- it's possible to use it for server-side rendering (handle the initial render of the application by sending the state of the app to the server along with its response to the server request. Components are then rendered in HTML and sent to the clients).

Drawbacks can include:

- increased complexity: with additional layer of using manipulation logic actions or reducers, Redux increases complexity of code;
- lack of encapsulation: it's not possible to encapsulate data in Redux library and any component can access it, which for complex apps might cause potential security issues;
- restricted design: even though this helps with maintaining the app, it can also be an obstacle of reaching the desired outcome easily (code needs to be written in a non-mutable manner);
- Redux is an in-memory state store, so if application crashes, the entire app state is lost. Caching solution has to be used to create a backup, that can add extra overhead.

Therefore, this is a great solution for big scale, complex application, but might not be needed for small projects that follow simple logic and have one source of data.

Redux, its toolkit, core and complementary packages can be installed with package manager, such as npm or yarn. It is then recommended to start a new app with React and Redux by using the official template (JS or Typescript), so for example: npx create-react-app my-app --template redux.

7. Please describe Linux in as much detail as possible (feel free to use notes made during lessons or draw from the lesson directly!). Especially consider: *what is its history? Why would someone use it over other existing operating systems? How does Windows and Mac OSX differ to Linux? How does Linux function, what are some unique features to it? How can it be installed today?* (10 marks)

   Linux is a family of free open-source operating systems based on Linux kernel that was first released in 1991 by Linus Torvalds. While studying at University of Helsinki, he decided to start a project of creating a system like MINIX (a UNIX operating system). In a post to a newsgroup, he wrote:

   > *I'm doing a (free) operating system (**just a hobby, won't be big and professional like gnu**) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).*

   GNU that he mentioned in his post, was a project led by Richard Stallman, who wanted to create an operating system that is free and has accessible by source-users code. GNU stands for "GNU's not Unix!", meaning that it is Unix-like, but doesn't contain any Unix code. Because there were delays in producing a kernel (GNU Hurd), Linux kernel came into play.

   Currently this "small, hobby project" of Linus Torvalds is running on 96.3% of the world's top 1 million web servers, and all the 500 fastest supercomputers (source: https://truelist.co/blog/linux-statistics/).

The main difference between Linux and big operating systems like Microsoft Windows or maxOS is the fact that it is open source. It is also its most unique and prominent feature. The components can be customised and it doesn't come preloaded with bloatware. Users have easy access to the source code, making it a very flexible system. It is also a command-based one, meaning that operations are performed by executing specific programming instructions in a terminal (although this can be altered by installing a distribution with graphic user interface). Whilst Windows or Max OSX have maintained rather standard version structure, Linux, due to its open-source nature, has currently more than 300 actively maintained distributions (Linux-based OS).

There are multiple reasons someone could choose Linux over commercial OS:

- programs and files downloaded on Linux can't change settings and configuration if user is not logged in as root, making it safer and increases security. The fact that its source code is freely available for review also means that most of the weaknesses have already been discovered;
- Linux doesn't need to be rebooted after every update or patch, making it a first choice when it comes to choosing a system to run on the server;
- it is very stable and not prone to crashes, and its speed of operating doesn't decrease with time;
- users can update their chosen variant of Linux and all the installed software centrally;
- it can run on any hardware;
- it's free, so even businesses can use it, reducing the cost of IT operation (side note: original Linux kernel came with commercial use restrictions, but it was later changed);
- since the raise of Linux GUIs, it has become easy to use also for people not very verbose in command terminal.

It's important to note that Linux has still a way to go when it comes to software compatibility. Despite efforts of providing alternatives and possibility of running Windows emulator or Darling macOS compatibility layer, there is still a gap to breach.

The architecture of Linux system can be represented by a diagram:

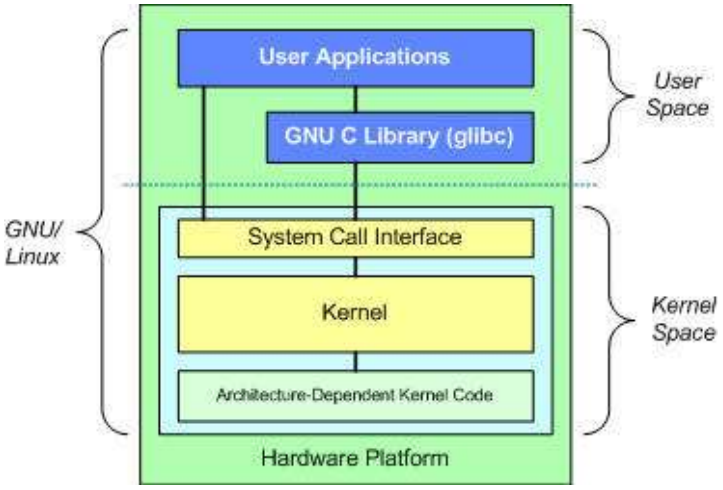

Figure 5: Linux architecture diagram, source: https://developer.ibm.com/articles/l-linux-kernel/

It contains a hardware platform on which it is installed, a kernel space that manages the resources, and user space where user applications are executed.

There are many ways of installing Linux – it's even possible to have both Linux and another operating system on one computer (dual-booting, only one system will run at a specific

time, but user can choose on start which one to use). There are also distributions that will boot from a USB (for example Linux Lite).

8. What are they, and which is better between Class components and Functional components? Provide a discussion. Consider: *Go deep - how does each one work? What are the unique properties or behaviours to each one? Why would someone use one over the other? What are the advantages and disadvantages of each one? Who benefits from these advantages and disadvantages, who is it suitable for?* (10 marks)

   Components in React are independent and reusable pieces of code. There are two types: class and functional components.

   Class components are JavaScript classes that extend React.Component (accessing all the component's properties) and return JSX inside a render method (which is essential to include). Other React.Component methods (like constructor(), componentDidMount() etc.) are optional.

```
import React from "react";

class ClassComponent extends React.Component {
 render() {
   return <h1>Hello, world</h1>;
 }
}
```

   Functional component are JavaScript functions that return JSX.

```
import React from "react";

function FunctionalComponent() {
return <h1>Hello, world</h1>;
}
```

   Both have to follow naming convention (starting with capital letter), but only class component will extend React.Component and have a render function.

   Beside the syntax, there are also other differences, that make each component variant unique:

   Passing props:

   In class component, props (described in Question 2) are accessed an expression: {this.props.<propertyName>}:

```
class Hello extends React.Component {
 render() {
  return <h1>Hello {this.props.name}!</h1>;
 }
}

const element = <Hello name="World"/>;
ReactDOM.render(
element,
document.getElementById('root')
);
```

In functional component, "this" is not used, so the expression is {props.<propertyName>}:

```
function Hello(props) {
 return <h1>Hello {props.name}!</h1>;
}

const element = <Hello name="World" />;
ReactDOM.render(
 element,
 document.getElementById('root')
);
```

Handling state:

To initialise the state values (state description also available in Question 2) , class component uses "constructor" method:

```
class Timenow extends React.Component {
constructor(props) {
super(props);
this.state = {date: new Date()};
}

render() {
return(
<div>
<h1>Hello {this.props.name}!</h1>
<h2>Time Now {this.state.date.toLocaleTimeString()}.</h2>
</div>
);
}
}

const element = <Timenow name="World"/>;
ReactDOM.render(
element,
document.getElementById('root')
);
```

In functional component, state is managed by useState Hook (hooks described in Question 3):

```
function Timenow(props) {
 const [ date, setDate ] = React.useState(new Date());
 return (
  <div>
   <h1>Hello {props.name}!</h1>
   <h2>Time Now {date.toLocaleTimeString()}.</h2>
  </div>
 );
}

const element = <Timenow name="World" />;
ReactDOM.render(
 element,
 document.getElementById('root')
);
```

Lifecycle Methods:

The difference in handling them by class and component function has been described in Question 3.

Error Boundaries:

Handling error boundaries is possible only in class components. Error boundaries are React components, which will catch JavaScript errors anywhere in their child component tree. They will then log those errors and then display a backup UI, instead of

the component tree that crashed. They don't catch errors for event handlers, asynchronous code, server side rendering and errors in the error boundary itself. They catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them. There is a great explanation with examples available here: https://www.digitalocean.com/community/tutorials/react-error-boundaries

Personally, I think it is important to know both ways of creating React components. A lot of legacy code is written in classes, so it is imperative to understand their syntax and how they work. But functional components have so many benefits (like less code, decreased complexity, ease of testing, increased readability for both humans and machines to name a few), and they are becoming a recommended standard, so I would use them over class components going forward.