Q. Write a report on your understanding of Rendering and Design Patterns. Mention and elaborate where a particular Rendering pattern is applicable and is well suited for which use case.

🡪 Ans.

1. Rendering Patttern

Following are the types of Rendering pattern

✨CSR : Client Side Rendering

✨SSR : Server Side Rendering

✨SSG : Static Site Generation

✨ISR : Incremental Static Regeneration

What is Build process ?

source code --> build --> server --> client (brower)

first the source code is passed to build process, that it the code is stored on to the server. and then sent on to the brower.

e.g. npm run build / dev etc.

1) CSR : client side Rendering

Build server client

| 🟥🟥🟥 |

| |

html | |

🟥🟥🟥 | |

|----> ⬜⬜⬜ | --> 🟥🟥 --> 🟥🟢 --> 🟥⬜🟢

javascript | |

🟢🟢🟢 | |

| |

css | |

◻️◻️◻️ | 🟢🟢🟢 |

| |

- the build phase is something where you write the code

- on the sever all your code is kept seperate and all the html, css and js are kept seperate.

- and now on client side first we send html, and then javascript is sent and as required the js is sent and css is also added.

- this is the core react approach, where everthing will happen through javascript and we sent javascript to the client.

- since empty html page is thrown on client side, it is difficult for search engines. There is no content there, but it is actually created when client visites the page.

- here the web page is rendered/created on client side

Where it is used?

* Single Page Applications (SPAs):

SPAs load a single HTML page and dynamically update the content as the user interacts with the application. The rendering is done on the client side using JavaScript frameworks like React, Angular, or Vue.js. This provides a smoother and more interactive user experience.

* Offline Capabilities:

With client-side rendering, it is possible to cache resources and enable offline access to certain parts of the application. This is beneficial for users in environments with limited or intermittent connectivity.

------------------------------------------------------------------------------------------------------------------------------------------------------

2) Server-side Rendering

Build Server Client

html | 🟥🟢 |

🟥🟥 | ◻️ ----> 🟥🟢

| <--- ◻️

JS | 🟥🟢 |

🟢🟢 | ◻️ ----> 🟥🟢

| <---- ◻️

CSS | |

◻️◻️ | 🟥🟢 |

| |

- here all the html, css and js are loaded on server itself.

- and makes request to the server and on each request new web page is loaded.

- server has more power than the normal browser cause user uses his mobile phone, or laptop which have very limited ram and processign power.

- so, sending everying on server is beneficial as it is expandable too.

Where it is used?

* Search Engine Optimization (SEO):

SSR is beneficial for SEO because search engine crawlers can easily index the content that is rendered on the server. This is important for websites that rely on search engine visibility, as client-side rendering may face challenges in terms of search engine indexing.

* Improved Initial Page Load Time:

For websites that prioritize fast initial page load times, server-side rendering can provide a better experience. The server sends a fully rendered HTML page to the client, reducing the time it takes for the user to see meaningful content, especially on the first visit.

-----------------------------------------------------------------------------------------------------------------------------------------------------

3) SSG : Static Site Generation

Build Server Client

| 🟥🟢 |

🟥🟢 | ◻️ ----> 🟥🟢

⬜ | <---- ◻️

| 🟥🟢 |

🟥🟢 | ◻️ ----> 🟥🟢

🔲 | <---- ◻️

| |

🟥🟢 | 🟥🟢 |

◻️ | ⬜ |

here, the html css and javascript are combined together at the time of build only.

hence it takes a lot of time but it is the most efficient way.

creating web pages at build time.

Where it is used?

* Content-Based Websites:

Static site generation is well-suited for content-based websites, such as blogs, documentation sites, and news portals, where the content doesn't change frequently

* more performance and speed

Static sites are fast because they don't require server-side processing for each request.

* hosting is also cost effective

------------------------------------------------------------------------------------------------------------------------------------

4) ISR : Incremental Static Regeneration

Build Server Client

| | 🟥🟢 |

| 🟥🟢 | ◻️ ----> 🟥🟢

source--> | ⬜ | <--- ◻️

code | | 🟥🟢 |

| 🟥🟢 | ◻️ ----> 🟥🟢

| 🔲 | <---- ◻️

| | |

| 🟥🟢 | 🟥🟢 |

| ◻️ | ⬜ |

- in SSR we were unable to get the updates quickly, whenever the next phase is build then the updates are pushed.

e.g. let say you want to change some content after every 2 days, and you don't know when the next build phase is, so it is not convinient.

- but, in ISR after every some time the build phase will happen, so you can get the updated content in real time.

Where it is used ?

* Large Sites with Frequently Changing Content:

For large websites with a considerable amount of content, regenerating the entire site can be time-consuming and resource-intensive. Incremental static site generation allows developers to update only the parts of the site that have changed, reducing build times.

* Frequent Content Updates:

Websites that require frequent content updates, such as news sites, blogs, or e-commerce platforms, can benefit from ISG. Instead of regenerating the entire site for every new piece of content, only the affected pages are regenerated, improving efficiency.

2. Desing Patterns

* Creational Patterns

- it deal with the process of object creation. They provide solutions to the problem of how to instantiate objects in a flexible and efficient way, without specifying their exact classes this focuses on 2 thigns.

- abstraction

- hiding

there are 5 Categories

👉 Abstract factory DP

- it provides an interface for creating related objects without specifying their concrete class.

- multiple look and feel

------------------------------------------------------------------------

👉 Builder DP

- Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.

- builder : defines the interface

concreteBuilder : implements the interface

director : constructs an object by using interface provided by builder

product : reprsents the object under constructions

code:

class Product

{

private:

string partA\_, partB\_, partC\_;

public:

// Methods to set different parts of the product

// only defining the interface

void setPartA(const string& partA) {

partA\_ = partA;

}

void setPartB(const string& partB) {

partB\_ = partB;

}

void setPartC(const std::string& partC) {

partC\_ = partC;

}

void show() {

std::cout << "Part A: " << partA\_ << "\n";

std::cout << "Part B: " << partB\_ << "\n";

std::cout << "Part C: " << partC\_ << "\n";

}

};

class Builder {

public:

virtual void buildPartA() = 0;

virtual void buildPartB() = 0;

virtual void buildPartC() = 0;

virtual Product\* getResult() = 0;

}

class ConcreteBuilder : public Builder {

private:

Product\* product;

public:

// construction of methods

ConcreteBuilder() : product(new Product()) {}

void buildPartA() override {

product->setPartA("A");

}

void buildPartB() override {

product->setPartB("B");

}

void buildPartC() override {

product->setPartC("C");

}

Product\* getResult() override {

return product;

}

};

--------------------------------------------------------------------------

3. Factory DP

- it defines an inerface for creating an object, but the subclasses decides which class to instantiate.

- all the implementation details are stored in one class, and whenever a particular class want's to access it, he can use it.

e.g. creating desktop application where two abstract classes are there

> application

> documentation

e.g. creating a drawing application

- drawingApplication

- drawingDocumentation

structure:

- again the same participants are there,

> product

> concreteProduct()

#️⃣ Code

class Product {

public:

virtual void create() = 0;

};

class ConcreteProductA : public Product {

public:

void create() override {

// Implementation for creating Product A

}

};

class ConcreteProductB : public Product {

public:

void create() override {

// Implementation for creating Product B

}

};

4. Prototpye dp

- prototype is a template for any object before the actual object is constructed.

- you will clone the interface and then can add the functionalities to it.

structure:

- again there will be participants

> client (calls prototype())

> prototype (defines the interface )

> concretePrototype1 (constructs the product)

> concretePrototype2 (constructs the product)

class Prototype {

public:

virtual Prototype\* clone() = 0;

virtual void use() = 0;

};

class ConcretePrototype : public Prototype {

public:

Prototype\* clone() override {

return new ConcretePrototype(\*this);

}

void use() override {

// Implementation for using the cloned object

}

};

----------------------------------------------------------------------

5. Singletone

- it ensures that a class has single instance only and provides a global point of access to it meaning that, can be accessed from outside of the class as well.

- sometimes we want only one window manager or just one factory for a product, any other functionalities you don't want

strecture:

Singletone

-----------------

static instance() <---------- return uniqueinstance

SingletoneOperation()

code:

class Singleton {

private:

static Singleton\* instance;

Singleton() {} // Private constructor to prevent instantiation.

public:

static Singleton\* getInstance() {

if (!instance) {

instance = new Singleton();

}

return instance;

}

};

// Usage:

Singleton\* obj = Singleton::getInstance();