1 Как вы можете описать процесс распространения событий в

JavaScript (Event Propagation)? Расскажите о его основных фазах.

Какие есть способы применения знаний о фазах

рааспространения событий на практике?

Распространение событий предполагает вертикальное перемещение сначала сверху вниз по иерархии элементов страницы , срабатывание срабатывает на нужном элементе , а затем поднятие поднимается обратно вверх.

Цель этого для отслеживания кликов или действий на родительских элементах, даже если они произошли на дочерних, и управлять поведением страницы.

Например, можно повесить один обработчик на список, а не на каждый его пункт, или остановить всплытие, если нужно.

2 Расскажите, что такое Promise в JavaScript, как он работает, какие

основные состояния у него бывают? Каким еще способомможно

обработать асинхронный код? Какую роль в асинхронности играет

событийный цикл (Event Loop)?

Промис это элемент асинхронной работы (то есть когда нужно дожидаться выполнения определенной задачи, например запроса на сервер и затем делать что-то, обновлять поля или представление). Плюс такого подхода, что при его выполнении не блокируется выполнение остального кода. Промис может работать посредством колбэков или при помощи async и await, а также обрабатывать ошибки или успешное выполнение функции (через then и catch). Есть 3 статуса у промиса: Pending , Fulfilled , Rejected.Что касается Event Loop, то это скорее принцип работы javascript , по которому выполняются синхронные функции, хранится стек асинхронных задач, и обеспечивается их извлечение и выполнение из очереди.

3 Что такое ООП, и какие его ключевые принципы вы могли бы

описать? При объяснении приводите примеры. Максимально

подробно расскажите о том, как ООП реализовано в JavaScript.

В разработке применяются принципы SOLID, которые расшифровываются как

S — Single Responsibility Principle (Принцип единственной ответственности)

Каждый класс должен иметь только одну причину для изменения.

У класса должна быть только одна задача или ответственность.

Например один класс/модуль по хорошему не должны выполнять задачи рендеринга элемента и сохранение его в базу данных. Применительно для angular хорошей практикой считается использование компонента только для отображение содержания, с вынесением логики в отдельный сервис, доступный по всему приложению.

O — Open/Closed Principle (Принцип открытости/закрытости)

Классы должны быть открыты для расширения, но закрыты для изменения.

Новая функциональность добавляется через наследование или композицию, а не изменяя существующий код.

Например, лучше прописать отдельный класс геометрической фигуры и сделать наследование от нее отдельно для круга или квадрата (или другой формы) .

В ангуляре как пример можно привести создание директивы изменяющей цвет элемента, где цвет можно передать в нее как переменную (то есть сама директива не меняется в ходе использования).

L — Liskov Substitution Principle (Принцип подстановки Барбары Лисков)

Объекты подклассов должны заменять объекты базового класса без нарушения работы программы.

Наследуемые классы не должны "ломать" поведение родительского класса.

Так если есть класс квадрата и класс прямоугольника, который он наследует, то квадрат не должен ломать логику отрисовки прямоугольника, т.к. равенство сторон разрушит первоначальную логику родительского класса.

В ангулар в случае наследование компонент/класс должен не игнорировать /ломать родительскую логику а вызывать родительские методы.

I — Interface Segregation Principle (Принцип разделения интерфейсов)

Лучше много маленьких интерфейсов, чем один большой.

Интерфейсы должны быть специфичными для клиента, а не универсальными.

Например, лучше прописать отдельно интерфейсы для каждой из сущностей.

Например некорректно было бы использовать

interface UserActions {

create(user: User): void;

update(user: User): void;

delete(id: number): void;

getAll(): User[];

getById(id: number): User;

}

А правильно разбить сущность на несколько интерфейсов

interface UserReader {

getAll(): User[];

getById(id: number): User;

}

interface UserWriter {

create(user: User): void;

update(user: User): void;

delete(id: number): void;

}

// Сервис реализует только нужные интерфейсы

class UserService implements UserReader, UserWriter { }

D — Dependency Inversion Principle (Принцип инверсии зависимостей)

Зависимости должны строиться на абстракциях (интерфейсах), а не на конкретных реализациях.

Высокоуровневые модули не должны зависеть от низкоуровневых. Зависимости должны строиться на абстракциях (интерфейсах), а не на конкретных реализациях.

В ангуляр используются например сервисы

@Injectable({ providedIn: 'root' })

class HttpApiService implements ApiService {

constructor(private http: HttpClient) {}

get<T>(url: string) { return this.http.get<T>(url); }

}

@Injectable({ providedIn: 'root' })

class MockApiService implements ApiService {

get<T>(url: string) { return of({} as T); }

}

При этом компонент не знает, какой именно класс используется, т.к. в модуле (раздел провайдерс) указывается нужный нам сервис/интерфейс.

4 Опишите максимально детально, как браузер обрабатывает URL,

начиная с момента его ввода в адресную строку и заканчивая

отображением страницы. Какие технологии позволяют ускорить

процесс? Какие проблемы (безопастность, междоменное

взаимодействие) могут возникать в процессе?

1. Ввод и парсинг URL

Браузер анализирует URL (протокол, домен, путь, параметры).

Добавляет https://, если не указан протокол.

2. Проверка кэша

Ищет ресурс в HTTP-кэше или Service Worker.

Если актуален — использует кэш, иначе идёт дальше.

3. Разрешение DNS

Преобразует домен в IP-адрес через DNS-серверы (кэширует результат).

4. Установка соединения

TCP-handshake (3 шага: SYN → SYN-ACK → ACK).

Для HTTPS — TLS-handshake (шифрование, проверка сертификата).

5. HTTP-запрос и ответ

Браузер отправляет запрос (GET, POST и др.).

Сервер возвращает статус-код (200, 404, 301) + ресурс (HTML, JSON, изображения).

Проблемы: CORS (междоменные запросы), перенаправления, ошибки сервера.

6. Парсинг и рендеринг

DOM (из HTML) + CSSOM (из CSS) → Render Tree (что и как отображать).

JavaScript может модифицировать DOM/CSSOM (блокирует рендеринг, если не async/defer).

Layout (расчёт позиций элементов) → Paint (прорисовка пикселей).

7. Оптимизации ускорения

HTTP/2, HTTP/3 (мультиплексирование, меньшие задержки).

Кэширование (HTTP-кэш, Service Worker).

CDN (размещение контента ближе к пользователю).

Lazy Loading (отложенная загрузка изображений/скриптов).

SSR (рендеринг на сервере, например, Next.js).

8. Проблемы безопасности

XSS (внедрение скриптов) → CSP, экранирование.

CSRF (подделка запросов) → CSRF-токены.

MITM (перехват трафика) → HTTPS, HSTS.

CORS (блокировка междоменных запросов) → заголовки Access-Control-Allow-Origin.