**3.1. Управление состоянием приложения**

Управление

состоянием сложная задача. Необходимо управлять работой backend, webworkersи

компонентов UI, каждый и которых может

обновлять стостояние.

Какие данные хранить в

памяти, а что в URL? Что нужно для UI? Как синхронизировать состояние?

Типысостояния:

Состояние сервера

Постоянное

состояние

URL и состояние маршрутизации

Состояние клиента

Временное состояние

клиента

Локальное

состояние UI

Состояние сервера

хранится на сервере и предоставляется через REST.

Постоянное состояние

– данные состояний сервера которые хранятся локально на клиенте в памяти. Можно

сказать, что это кэш состояний сервера.

Состояние клиента не

хранится на сервере. Примером являются фильтры для создания списков отображающихся

пользователю. Элементы фильтрации хранятся на сервере, а значение фильтров нет.

И рекомендуется постоянное состояние и состояние клиента хранить в URL.

Приложения часто имеют

состояние, которое хранится на клиенте, но не представлено в URL. Как пример, YouTube запоминает

где остановлено видео. При следующем запуске видео начнется с остановленного момента.

Информация о остановке не храниться в URL

и пре передачи ссылки кому-либо видео

будет воспроизводится с начала. Это

временное состояние клиента.

Отдельные компоненты могут

иметь локальное состояние, управляющее аспектами поведения. Свернут список или

нет, какой цвет кнопки в данный момент. Это все локальный UI.

Как определить тип состояния – задать вопросы: можно

им поделится и где он живет.

**3.1.1 Чистые функции**

Возвращаемое значение чистой функции зависит только от ее входных данных (аргументов) и не влечет никаких побочных эффектов. С одним и тем же входящим аргументом результат всегда будет одинаковый. Пример:

function half(x) {  
 return x / 2;  
}

Функция half(x) принимает число x и возвращает значение половины x. Передадим этой функции аргумент 8 и она вернет 4. При вызове чистая функция всегда может быть заменена результатом своей работы. Например, можно бы заменить half(8) на 4: где бы эта функция не использовалась в коде, подмена никак не повлияла на конечный результат.

Чистые функции зависят только от того, что им передано. Например, чистая функция не может ссылаться на переменные из области видимости родителя, если они явно не передаются в нее в качестве аргументов. Но и даже тогда функция не может изменять что-либо в родительской области видимости.

var someNum = 8;  
  
// это НЕ чистая функция  
function impureHalf() {  
 return someNum / 2;  
}

* Чистые функции должны принимать аргументы.
* Одни и те же входные данные (аргументы) всегда произведут одинаковые выходные данные (вернут одинаковый результат).
* Чистые функции основываются только на внутреннем состоянии и не изменяют внешнее (примечание: console.log изменяет глобальное состояние).
* Чистые функции не производят побочных эффектов.
* Чистые функции не могут вызывать нечистые функции.

**3.1.2 Нечистые функции**

Нечистая функция изменяет состояние вне своей области видимости. Любые функции с *побочными эффектами* - нечистые, ровно как и процедурные функции без возвращаемого значения.

// нечистая функция производит побочный эффект

function showAlert() {

alert('This is a side effect!');

}

// нечистая функция изменяет внешнее состояние

var globalVal = 1;

function incrementGlobalVal(x) {

globalVal += x;

}

// нечистая функция процедурно вызывает чистую фунцию

function proceduralFn() {

const result1 = pureFnFirst(1);

const result2 = pureFnLast(2);

console.log(`Done with ${result1} and ${result2}!`);

}

// нечистая функция выглядит чистой,

// но возвращает разные значения

// при одинаковых входных данных

function getRandomRange(min, max) {

return Math.random() \* (max - min) + min;

}

#### 3.1.3 Побочные эффекты в JavaScript

Когда функция или выражение изменяет состояние вне своего контекста, результат является побочным эффектом. Примеры побочных эффектов: вызов API, манипулирование DOM, вывод alert, запись в базу данных и так далее. Если функция производит побочные эффекты, она считается *нечистой*. Функции, вызывающие побочные эффекты, менее предсказуемы и их труднее тестировать, поскольку они приводят к изменениям вне их локальной области видимости.

**3.1.4 Состояние**

Состояние — информация, к которой программа имеет доступ и с которой может работать в определенный момент времени. Сюда входят данные, хранящиеся в памяти, порты ввода/вывода, базы данных и так далее. Например, содержимое переменных в приложении в любой данный момент времени репрезентативно для состояния приложения.

**С состоянием**

Программы, приложения или компоненты с состоянием хранят в памяти данные о текущем состоянии. Они могут изменять состояние, а также имеют доступ к его истории. Следующий пример демонстрирует это:

var number = 1;

function increment() {  
 return number++;  
}

// глобальная переменная изменяется: number = 2  
increment();

#### Без состояния

Функции или компоненты без состояния выполняют задачи, словно каждый раз их запускают впервые. Они не ссылаются или не используют в своем исполнении раннее созданные данные. Функции зависят только от их аргументов и не имеют доступа, не нуждаются в знании чего-либо вне их области видимости. Чистые функции не имеют состояния. Пример:

var number = 1;

function increment(n) {  
 return n + 1;  
}

// глобальная переменная НЕ изменяется: возвращает 2  
increment(number);

Приложения без состояния все еще управляют состоянием. Однако они возвращают свое текущее состояние без изменения предыдущего состояния. Это принцип функционального программирования.

Управление состоянием важно для любого сложного приложения. Функции или компоненты с состоянием изменяют состояние и его историю, их труднее тестировать и отлаживать. Функции без состояния полагаются только на свои входные данные для создания данных выходных. Программа без состояния возвращает новое состояние, а не модифицирует существующее состояние.

**3.1.5 Неизменяемость и изменяемость**

Концепции неизменяемости и изменяемости более туманны в JavaScript, чем в некоторых других языках программирования. Важно знать, что эти термины означают и как они реализуются в JavaScript. Определения достаточно просты:

**Неизменяемый**

Если объект является неизменяемым, его значение не может быть изменено после создания.

**Изменяемый**

Если объект изменяем, его значение может быть изменено после создания.

**Реализация: неизменяемость и изменяемость в JavaScript**

В JavaScript строки и числовые литералы реализованы неизменяемыми. Это легко понять, если рассмотреть, как мы работаем с ними:

var str = 'Hello!';  
var anotherStr = str.substring(2);  
// результат: str = 'Hello!' (не изменена)  
// результат: anotherStr = 'llo!' (новая строка)

Используя метод substring() на нашем Hello!, строка не изменяет исходную строку. Вместо этого она создает новую строку. Мы могли бы переопределить значение переменной str на что-то другое, но, как только мы создали нашу строку Hello!, она навсегда останется Hello!.

Числовые литералы также неизменяемы. Следующий пример всегда будет иметь одинаковый результат:

var three = 1 + 2;  
// результат: three = 3

Ни при каких обстоятельствах 1 + 2 не может стать чем-либо, кроме 3.

Это демонстрирует, что в JavaScript присутствует реализация неизменяемости. Однако разработчики JS знают, что язык позволяет изменить многое. Например, объекты и массивы изменяемы. Рассмотрим следующий пример:

var arr = [1, 2, 3];  
arr.push(4);  
// результат: arr = [1, 2, 3, 4]  
  
var obj = { greeting: 'Hello' };  
obj.name = 'Jon';  
// результат: obj = { greeting: 'Hello', name: 'Jon' }

В этих примерах исходные объекты изменены. Новые объекты не возвращаются.