Теоретическая часть

1. Почему современные веб-приложения часто реализовывают с разделением на backend и frontend части? За что отвечает backend, а за что frontend?

Современные веб-приложения часто используют архитектурное разделение на frontend и backend для повышения гибкости, масштабируемости и надежности системы.

- Разделение ответственности
  Интерфейс и пользовательское взаимодействие обрабатываются на стороне клиента (frontend), а обработка данных, бизнес-логика и безопасность — на стороне сервера (backend). Это упрощает поддержку кода и позволяет разделить зоны ответственности между разработчиками;
- Параллельная разработка
  Команды могут работать независимо: одна разрабатывает пользовательский интерфейс, другая — серверную логику и API;
- Масштабируемость и переиспользуемость
  Backend можно использовать для разных клиентских платформ (веб, мобильные приложения и т.д.). Компоненты системы масштабируются отдельно, в зависимости от нагрузки;
- Безопасность
  Критичная логика, доступ к базе данных и проверка прав пользователей выполняются на стороне сервера. Это предотвращает возможность обхода или вмешательства со стороны клиента.

В чём основные отличия? Бэкенд занимается разработкой серверной части веб-приложений и сайтов. Он отвечает за работу баз данных, серверов и логику, которая происходит на серверной стороне. Frontend, напротив, занимается созданием клиентской части веб-приложений, сайтов, которая взаимодействует с пользователем.

Что делает frontend?

- Отображение данных пользователю.
- Реакцию на действия пользователя.
- Получение и отправку данных через API.
- Поддержание состояния интерфейса, маршрутизацию и валидацию данных.

Он работает в браузере пользователя и представляет собой «витрину» приложения.

Что делает backend?

- Обработку запросов от клиента.
- Выполнение бизнес-логики.
- Работа с базами данных и хранение информации.
- Управление пользователями, правами доступа и безопасностью.

- Интеграцию с внешними сервисами и системами.

Он работает на сервере и служит основой всей функциональности, которая не видна напрямую пользователю.

2. Почему современный frontend часто реализовывают с помощью фреймворков (angular, react, vue.js)? Почему часто на фронтенде используют стрейт менеджеры (redux, mobx)?

- Упрощение разработки сложных интерфейсов
  Современные интерфейсы — это динамичные, интерактивные приложения с множеством состояний. Фреймворки помогают управлять этим сложным поведением предсказуемо и эффективно.
- Компонентный подход
  Фреймворки позволяют строить интерфейс из переиспользуемых и изолированных компонентов, что облегчает сопровождение и масштабирование кода.
- Реактивность и управление состоянием
  Фреймворки обеспечивают автоматическое обновление UI при изменении состояния (например, данные обновились — интерфейс перерисовался без ручного вмешательства).
- Богатая экосистема и инструменты
  Фреймворки предоставляют поддержку маршрутизации, работы с формами, анимаций, сборки и тестирования «из коробки» или через экосистему.
- Поддержка сообщества и корпоративная надежность
  Большие сообщества, документация и поддержка делают фреймворки устойчивым выбором для коммерческих проектов.

Почему часто используют state-менеджеры?

- Централизованное управление состоянием
  Когда приложение растёт, передача состояния между множеством компонентов становится сложной. State-менеджер предоставляет единый источник истины — глобальное хранилище состояния.
- Предсказуемость
  В Redux, например, состояние изменяется только через «actions» и «reducers». Это делает поведение приложения предсказуемым и проще отлаживаемым.
- Упрощение отладки и тестирования
  История изменений состояния можно логировать, откатывать и тестировать независимо от UI.
- Снижение сложности при масштабировании
  State-менеджер помогает избегать проблем с «пробросом пропсов» и разрастанием локальных состояний в глубоко вложенных компонентах.

3. Зачем в веб-приложениях использовать дизайн систему и компоненты? Почему бы не использовать просто CSS?

Зачем использовать дизайн-систему и компонентный подход, вместо просто CSS?

1. Единый внешний вид и поведение (Consistency)

Дизайн-система задает правила оформления: цвета, отступы, шрифты, поведение компонентов. Это гарантирует, что интерфейс будет выглядеть и вести себя одинаково на всех страницах.

> Без дизайн-системы два разработчика могут оформить одинаковую кнопку по-разному, что создает хаос в UI.

2. Повторное использование компонентов

Компонент — это готовый UI-элемент с логикой, стилями и поведением. Вместо того чтобы писать одно и то же снова и снова, ты просто подключаешь готовый компонент.

> Пример: кнопка, карточка, модальное окно — всё это может быть переиспользовано десятки раз.

3. Упрощение поддержки и масштабирования

Если изменить стиль в компоненте (например, цвет у кнопки), все места, где он используется, обновятся автоматически. Это экономит время и снижает риск ошибок.

4. Снижение количества дублирующего CSS

Когда нет компонентного подхода, часто возникает дублирование: похожие стили пишутся заново. Это раздувает кодовую базу и мешает поддержке.

5. Совместная работа дизайнеров и разработчиков

Дизайн-система — это «единый язык» между дизайном и фронтендом. Дизайнер рисует компоненты согласно системе, а разработчик реализует их один раз и переиспользует.

Почему одного CSS может не хватать?

- CSS без структуры не масштабируется;
- В больших проектах быстро возникает рассинхрон в стилях;
- Нет гарантии, что интерфейс будет единообразным;
- Труднее отлаживать, изменять и поддерживать код.

4. Зачем придумали разные виды тестирования: юнит-тестирование, интеграционное тестирование, e2e тестирование? В чем разница и какое когда использовать?

- Юнит-тесты проверяют логику на "молекулярном" уровне (отдельные функции/компоненты);
- Интеграционные тесты проверяют, как модули работают вместе;
- E2E-тесты проверяют поведение системы в целом, как пользователь её видит.

1. Юнит-тестирование (Unit Testing)

Модульное (компонентное) тестирование (unit testing, module testing, component testing) направлено на проверку отдельных небольших частей приложения, которые (как правило) можно исследовать изолированно от других подобных частей. При выполнении данного тестирования могут проверяться отдельные функции или методы классов, сами классы, взаимодействие классов, небольшие библиотеки, отдельные части приложения. Часто данный вид тестирования реализуется с использованием специальных технологий и инструментальных средств автоматизации тестирования, значительно упрощающих и ускоряющих разработку соответствующих тест-кейсов.

Что тестирует:
Один маленький блок кода — функцию, метод или UI-компонент — в изоляции от всего остального.

Цель:
Убедиться, что каждая единица логики работает корректно при разных входных данных.

Когда использовать:

- При тестировании утилит, форматирования, вычислений
- При разработке компонентов с логикой

2. Интеграционное тестирование (Integration Testing)

Интеграционное тестирование направлено на проверку взаимодействия между несколькими частями приложения (каждая из которых, в свою очередь, проверена отдельно на стадии модульного тестирования). К сожалению, даже если мы работаем с очень качественными отдельными компонентами, «на стыке» их взаимодействия часто возникают проблемы. Именно эти проблемы и выявляет интеграционное тестирование.

Что тестирует:
Как работают несколько модулей вместе: компоненты + API, формы + валидация и т.п.

Цель:
Проверить, что связка частей приложения правильно взаимодействует между собой.

Когда использовать:

- При тестировании логики между UI и API
- При проверке взаимодействий между компонентами

3. E2E-тестирование (End-to-End Testing)

E2E (сквозное) тестирование подразумевает проверку всего приложения целиком, от начала до конца. Этот тип тестов обычно имитирует реальный пользовательский сценарий и проверяет, что все юниты, или части системы, работают вместе корректно и выполняют бизнес-требования.

Что тестирует:
Полное поведение приложения с точки зрения пользователя — от открытия страницы до результата действий.

Цель:
Проверить, что всё работает в реальной среде: UI + роутинг + API + база данных и т.д.

Когда использовать:

- Для критических пользовательских сценариев (регистрация, заказ, оплата);
- В CI перед деплоем;
- Для регрессионного тестирования.

Практическая часть

Экшены Redux:

```
ACTION_TYPES = { //Определяем набор констант - “типы действий”
  FETCH_DATA_REQUEST, //Начала загрузки
  FETCH_DATA_SUCCESS, //Данные успешно загружены
  FETCH_DATA_FAILURE, //Во время загрузки произошла ошибка
  CLEAR_DATA, //Очистить таблицу
  SAVE_CSV, //Сохранить данные в csv
  SELECT_ROW, //Выбрать строку
}

function fetchData() { // асинхронный экшн для возвращения функции
  return async dispatch => {
    dispatch({ type: FETCH_DATA_REQUEST }); // включение индикатора загрузки
    try {
      const data = await api.get('/data'); // запрос данных с сервера
      dispatch({ type: FETCH_DATA_SUCCESS, payload: data }); // сохранение полученных
данных
```

```
      const maxRowId = findMaxRowId(data); //автоматически выбирать строку с max value
      dispatch({ type: SELECT_ROW, payload: maxRowId });
    } catch (err) {
      dispatch({ type: FETCH_DATA_FAILURE, error: err }); // сохранение ошибки в стейт
    }
  }
}

// синхронные экшны для:
function clearData() {
  return { type: CLEAR_DATA } // для очистки таблицы
}

function saveCsv() {
  return { type: SAVE_CSV } // для запуска скачивания csv-файла
}

function selectRow(rowId) {
  return { type: SELECT_ROW, payload: rowId } //для изменения selectedRowId при клике
// по строке
}
```

Редюсеры Redux:

```
state = {
  tableData: {
    headers: ["title", "x1", "x2"...],  // заголовки столбцов
    rows: [
      {id: 1, title: "abc", x1: y1, x2: y2},  // данные строк
      {id: 2, title: "cde", x1: w1, x2: w2},
    ]
  },
  selectedRowId: id строки с максимальным значением в выбранном столбце
  selectedColumn: выбранный столбец для анализа
  isLoading: false,
  error: null
}

function rootReducer(state = initialState, action) {
  switch (action.type) {
    case FETCH_DATA_REQUEST: // пользователь нажал fetch - включение загрузки и
//сброс ошибок
          return { ...state, isLoading: true, error: null }
    case FETCH_DATA_SUCCESS: // если данные успешно пришли - выключаем
//индикатор загрузки и
          return { ...state, isLoading: false, tableData: {
         headers: action.payload.headers,
```

```
        rows: action.payload.rows,
      },
}
   case FETCH_DATA_FAILURE: // выключаем загрузку и сохраняем ошибку
         return { ...state, isLoading: false, error: action.error }
   case CLEAR_DATA: // сброс к изначальному состоянию
         return { ...initialState }
   case SELECT_ROW: //пользователь кликнул на строчку или она выбрана
//автоматически
         return { ...state, selectedRowId: action.payload }
   case SAVE_CSV:
     // сбор CSV из tableData.rows и скачивание
     triggerCsvDownload(state.rows)
     return state // состояние не меняется
   default:
     return state
 }
}
```

Стор:

```
// создание redux хранилища и подключение thunk для диспатчинга функций
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
const store = createStore(rootReducer, applyMiddleware(thunk));
```

Компоненты React:

```
function ButtonPanel() {
  const dispatch = useDispatch();
  return (
    <div class="button-panel">
      <Button onClick={() => dispatch(fetchData())}>Fetche</Button>
      <Button onClick={() => dispatch(clearData())}>Clear</Button>
      <Button onClick={() => dispatch(saveCsv())}>Save</Button>
    </div>
  )
}

// компонент отвечает за отображение таблицы с данными, пользователь может кликом
// выбирать строку, выбранная строка подсвечивается
function Table() {
  const rows = useSelector(s => s.rows);
  const selectedId = useSelector(s => s.selectedRowId);
  const dispatch = useDispatch();

  return (
```

```
    <table>
     {rows.map(row => (
       <tr
         key={row.id}
         className={row.id === selectedId ? 'selected' : ''}
         onClick={() => dispatch(selectRow(row.id))}
       >
         <td>{row.id}</td>
         <td>{row.values.join(',')}</td>
       </tr>
     ))}
    </table>
  )
}

// визуализация данных выбранной строки:
function Chart() {
  const rows = useSelector(s => s.rows);
  const selectedId = useSelector(s => s.selectedRowId);
  const data = rows.find(r => r.id === selectedId)?.values || [];

  const data = Object.entries(selectedRow)
    .filter(([key]) => key !== 'id' && key !== 'title')
    .map(([key, val]) => ({ name: key, value: val }));


  return (
    <BarChart data={data} />
  )
}

function App() {
  return (
    <Provider store={store}>
      <div class="app">
        <ButtonPanel />
        <Chart />
        <Table />
      </div>
    </Provider>
  )
}
```

Стратегия тестирования

1. Юнит-тестирование: проверяем кейсы редьюсера (FETCH_DATA_SUCCESS должна корректно записывать значения headers и rows, CLEAR_DATA сбрасывает таблицу корректно), проверяем findMaxRowId (правильно ли выделена строка по умолчанию), можно посмотреть, что возвращают экшены (clearData или selectRow(rowId)), можно проверить обработку компонентов на странице (панель кнопок).
2. Интеграционное тестирование: проверяем сохранение csv (при клике запускаются преобразование данных, а затем процесс загрузки файла), взаимосвязь таблицы и графика (при клике на строку таблицы меняется `selectedRowId`, график обновляется в соответствии с новой строкой), получение и отображение данных (при клике на кнопку "fetch": должна вызывается `fetchData`, данные попадают в Redux стор, данные отображаются в таблице, автоматически выделяется строка с максимальным значением, график отображается по этой строке).
3. E2E: проверяем загрузку данных по клику на кнопку, очистку данных после загрузки данных, выбор строки вручную (строка должна быть выделена, график должен перестроиться), скачивание csv (проверяем, что началась загрузка после нажатия на кнопку save).