ChatGPT

Веб-разработка без бэкенда: клиентские решения, анонимность и P2P

Введение

Современные веб-приложения обычно полагаются на сервер (бэкенд) для хранения данных и выполнения бизнес-логики. Однако существуют подходы, позволяющие создавать полностью **клиентские приложения**, работающие **без собственного сервера**. В таких фронтенд-ориентированных приложениях все данные обрабатываются и хранятся на стороне клиента (в браузере пользователя), что несет ряд преимуществ: отсутствие затрат на содержание сервера, отсутствие необходимости администрирования и улучшение приватности/анонимности пользователя, так как данные не передаются на сторонние серверы. Рассмотрим подробно различные методы и технологии, позволяющие разрабатывать **веб-приложения без бэкенда**, включая хранение данных в браузере, шифрование на клиенте, peer-to-peer взаимодействие, использование блокчейна и других децентрализованных решений на фронтенде. Приведем концепции, примеры кода, их объяснения, а также обсудим плюсы, минусы и варианты применения каждого подхода.

Хранение данных на стороне клиента

Самый прямолинейный способ обойтись без бэкенда – **хранить все данные локально в браузере пользователя**. В HTML5 доступны разные механизмы хранения данных на клиенте:

- **Web Storage API**: предоставляет два хранилища ключ-значение – `localStorage` и `sessionStorage`. *LocalStorage* сохраняет данные между сессиями и доступна на постоянной основе (до очистки), а *SessionStorage* живёт только в пределах текущей сессии/вкладки браузера [1]. Пример использования `localStorage` в коде:

```
// Сохранение значения
localStorage.setItem('username', 'Alice');
// Чтение значения
const user = localStorage.getItem('username');
console.log(user); // "Alice"
```

Однако хранение чувствительной информации в localStorage небезопасно: эти данные доступны любому скрипту на странице, сохраняются между сеансами и могут быть украдены в случае XSS-атаки [2]. SessionStorage несколько безопаснее для временных данных (она автоматически очищается при закрытии вкладки) [3].

- **IndexedDB**: полноценная NoSQL-база данных в браузере. Она позволяет сохранять структурированные объекты, большие объемы данных и поддерживает асинхронные операции. IndexedDB лучше подходит для сложных или больших данных, чем localStorage (например, можно хранить бинарные файлы, использовать индексы для поиска и т.д.) [4] [5]. Работа с IndexedDB сложнее – обычно взаимодействие идёт через транзакции или с

помощью обёрток. Простой пример (с использованием библиотечной обёртки `idb`) для сохранения записи:

```
import { openDB } from 'idb';
const db = await openDB('app-db', 1, {
  upgrade(db) {
    db.createObjectStore('notes');
  }
});
// Сохранить объект в хранилище "notes"
await db.put('notes', { title: 'Заметка1', text: 'Привет!' }, 'note1');
// Получить объект по ключу
const note = await db.get('notes', 'note1');
console.log(note.text); // "Привет!"
```

Плюсы локального хранения данных:

- **Простота и быстрота**: не требуется настраивать сервер или базу данных — все сохраняется в браузере. Данные доступны мгновенно, без сетевых задержек.
- **Работа офлайн**: приложение может функционировать без доступа к сети. Например, Progressive Web App может кэшировать ресурсы и данные, позволяя пользоваться сервисом даже в режиме офлайн.
- **Приватность**: данные не отправляются на внешний сервер, остаются под контролем пользователя, что повышает анонимность. Нет риска утечки данных с сервера, если сервер отсутствует.
- **Отсутствие затрат на бэкэнд**: разработчику не нужно оплачивать хостинг базы данных или сервера для хранения пользовательских данных.

Минусы локального хранения:

- **Нет синхронизации между устройствами**: данные по умолчанию привязаны к конкретному браузеру на конкретном устройстве. Если пользователь откроет приложение с другого устройства, он не увидит свои прежние данные (если не предусмотреть механизм экспорта/импорта).
- **Риск потери данных**: пользователь может очистить данные браузера или переустановить приложение, и тогда все локально сохраненные данные будут удалены. Без сервера сложно обеспечить резервное копирование.
- **Ограничения по памяти**: браузеры ограничивают объем локального хранилища (для localStorage обычно порядка 5–10 МБ, для IndexedDB лимиты больше, но зависят от устройства). Хранение очень больших объемов данных на клиенте затруднительно.
- **Безопасность**: локальные данные уязвимы, если на страницу внедрен вредоносный скрипт. Например, токены в localStorage могут быть украдены через XSS. Поэтому для конфиденциальных данных **крайне желательно применять шифрование перед сохранением** [6] (смотрите следующий раздел).

Таким образом, локальное хранение подходит для приложений, где данные в основном используются самим пользователем (например, личные заметки, настройки приложения), либо для кеширования данных, полученных из внешних источников, чтобы минимизировать обращения к сети.

Шифрование и безопасность данных на фронтенде

Если приложение должно сохранять или передавать чувствительные данные без участия бэкэнда, важнейшую роль играет **шифрование на стороне клиента**. Шифруя данные **до** их сохранения или отправки, мы можем гарантировать, что никакая третья сторона (даже если данные попадут на внешний сервис или в облако) не сможет их прочитать без ключа расшифровки. Такой подход обеспечивает **конфиденциальность и анонимность**: серверу (если он вдруг участвует) передаются только зашифрованные блоки, не содержащие явных персональных данных.

В JavaScript доступны два основных вида криптографии:

- **Симметричное шифрование**: используется один общий ключ для шифрования и дешифрования данных. Это быстро и подходит для больших объемов данных. На практике часто применяют алгоритм AES (Advanced Encryption Standard). Например, с помощью библиотеки CryptoJS можно зашифровать сообщение так:

```
const key = CryptoJS.enc.Utf8.parse('16-byte secretkey');       // секретный
ключ
const iv  = CryptoJS.enc.Utf8.parse('16-byte initvector');      // вектор
инициализации
const ciphertext = CryptoJS.AES.encrypt('Привет, мир!', key, {
  iv: iv,
  mode: CryptoJS.mode.CBC,
  padding: CryptoJS.pad.Pkcs7
});
console.log(ciphertext.toString());
```

Пример: шифрование строки AES-алгоритмом с использованием CryptoJS [7].

В этом примере текст *"Привет, мир!"* будет преобразован в шифртекст с помощью ключа и IV длиной 16 байт. Для расшифровки понадобится тот же самый ключ. Симметричное шифрование удобно для локального хранения: например, можно перед сохранением в `localStorage` шифровать JSON с данными пользователя. **Важно:** ключ не должен быть захардкожен в коде; его можно сгенерировать из пароля пользователя (например, через PBKDF2) или хранить в безопасном месте [8].

- **Асимметричное шифрование**: используются два разных ключа – открытый (public key) для шифрования и закрытый (private key) для расшифрования. Это позволяет, например, шифровать данные на одном устройстве, а расшифровывать на другом, не передавая секретный ключ. В браузере доступен Web Cryptography API, позволяющий генерировать пары ключей RSA и применять их. Пример генерации ключей RSA-OAEP и использования открытого ключа для шифрования:

```
// Генерация пары ключей (асинхронно)
const keyPair = await crypto.subtle.generateKey(
  { name: "RSA-OAEP", modulusLength: 2048, publicExponent: new
Uint8Array([1, 0, 1]), hash: "SHA-256" },
```

```
  true,
  ["encrypt", "decrypt"]
);
const publicKey = keyPair.publicKey;
// Шифрование сообщения открытым ключом
const encoder = new TextEncoder();
const data = encoder.encode("Secret message");
const encryptedData = await crypto.subtle.encrypt({ name: "RSA-OAEP" },
publicKey, data);
console.log(new Uint8Array(encryptedData));
```

Здесь сгенерирована 2048-битная RSA-пара ключей. Полученный `publicKey` можно свободно распространять и использовать для шифрования данных, а расшифровать потом сможет только обладатель соответствующего `privateKey` (например, сам пользователь на своем устройстве).

Шифрование на фронтенде позволяет реализовать **концепцию End-to-End Encryption**: данные шифруются непосредственно в браузере отправителя и могут быть расшифрованы только получателем (или самим же пользователем на другом устройстве) – промежуточные узлы (например, серверы-посредники или облачные хранилища) видят лишь шифрованный поток. Это повышает анонимность, так как даже если приложение использует какие-то внешние сервисы хранения, эти сервисы не имеют доступа к содержимому данных.

Лучшие практики при клиентском шифровании данных:

- **Шифровать перед сохранением** – любые чувствительные данные (пароли, токены, личные заметки и т.д.) шифруются до того, как сохранить их в `localStorage`, IndexedDB или отправить через интернет [6].
- **Надежные алгоритмы** – использовать проверенные алгоритмы: AES (с длиной ключа 128/256 бит) для симметричного шифрования, RSA или эллиптические кривые для асимметричного. Не изобретать собственных методов шифрования [6].
- **Управление ключами** – не хранить ключи шифрования в открытом виде в коде. Лучше генерировать их на основе пароля пользователя либо хранить в `sessionStorage` (пока пользователь авторизован) или в памяти. Можно использовать API устройства (напр. Web Crypto API может сохранять ключи в контейнеры KeyStore браузера). **Никогда не выводить секретные ключи в консоль и не отправлять на сервер** [8].
- **Защита от компрометации страницы** – даже при шифровании нужно помнить, что если злоумышленник получит возможность выполнять свой код на странице (XSS), он потенциально может подменить логику шифрования или выкрасть ключ. Поэтому важно соблюдать общие меры безопасности фронтенда (Content Security Policy, экранирование вводов пользователя и пр.) [9] [10].

Плюсы шифрования на клиенте:

- Данные защищены даже при утечке: если даже зашифрованные данные попадут не в те руки (например, кто-то получит доступ к вашему локальному хранилищу или перехватит трафик), без ключа они бессмысленны.
- Отсутствие необходимости доверять серверу: при отсутствии бэкенда этот пункт очевиден, но даже если приложение все же общается с каким-то API, шифрование на клиенте обеспечивает приватность поверх HTTPS. Сервер хранит только зашифрованную информацию и не может ее прочитать [11] (если, конечно, ключ не хранится на том же сервере).

- Выполнение требований по безопасности/конфиденциальности: важно для анонимных или защищенных приложений (например, мессенджеры с шифрованием, приватные заметки, менеджеры паролей в браузере и т.д.).

Минусы и сложности:

- Необходимость управления ключами: пользователю нужно где-то хранить пароль или ключ для расшифровки. Если он его потеряет, восстановить данные будет невозможно (ведь никакой сервер не хранит копию). Это требует продумать UX (например, предупреждать о невозможности восстановления доступа).
- Дополнительная сложность разработки: внедрение криптографии требует тщательного подхода и тестирования. Ошибки в реализации могут привести к уязвимостям.
- Производительность: шифрование/дешифрование требует ресурсов CPU. Для небольших объемов это обычно не заметно, но шифрование больших файлов на слабых устройствах может занять время. В таких случаях можно использовать Web Workers, чтобы не блокировать основной поток приложения.

На практике шифрование на фронтенде часто применяется в сочетании с другими рассматриваемыми методами. Например, можно хранить зашифрованные данные пользователя в **облаке или децентрализованном хранилище** – при этом никто, кроме пользователя, их не расшифрует (примером являются “zero-knowledge” приложения, где даже провайдер сервиса не знает содержимого пользовательских данных).

P2P-коммуникация в браузере (WebRTC и др.)

Другой подход к созданию приложений без центрального сервера – это организация **прямого обмена данными между клиентами**, то есть по схеме *peer-to-peer (P2P)*. В веб-разработке основная технология для этого – **WebRTC** (Web Real-Time Communications). WebRTC позволяет двум (или более) браузерам устанавливать прямое соединение для передачи потокового видео, аудио или произвольных данных (через Data Channels) без прохождения через сервер.

Как работает WebRTC в общих чертах: чтобы два клиента установили связь, им нужно узнать сетевые адреса друг друга и обменяться служебной информацией о соединении (SDP – Session Description Protocol). Обычно для этого используют небольшой *сигнальный сервер*, который знакомит пиры друг с другом. Но после установления связи медиаданные или сообщения идут напрямую от одного пользователя к другому (через UDP, с шифрованием). Для обхода NAT/ WebRTC также может использовать публичные STUN/TURN-серверы, но это не сервер приложения, а инфраструктурный узел для связи.

Возможна ли работа WebRTC совсем без сервера? Да, если передать сигнальные данные вручную. Например, можно обменяться SDP-описаниями через любой внешний канал (да хоть по электронной почте или в чате) – это не масштабируемо, но демонстрирует принцип. Ниже приведен упрощенный пример установки P2P-соединения *без серверной сигнализации*: один пользователь копирует сгенерированное описание соединения и отправляет его второму, который вводит его в свое поле. После обмена описаниями два браузера устанавливают прямой канал данных:

```
<!-- Пример HTML страницы, устанавливающей P2P DataChannel без сервера -->
<h3>Локальное SDP (скопируйте и отправьте другому пиру)</h3>
<textarea id="local" readonly></textarea>
```

```
<h3>Удаленное SDP (вставьте полученное от другого пира)</h3>
<textarea id="remote"></textarea>
<button onclick="connect()">Установить соединение</button>

<script>
  const pc = new RTCPeerConnection();
  // Когда локальный SDP сформирован, выводим его в текстовое поле для
копирования
  pc.onicecandidate = (event) => {
    if (event.candidate === null) {
      local.value = JSON.stringify(pc.localDescription);
    }
  };
  // Обработчик входящего канала данных
  pc.ondatachannel = (event) => {
    const receiveChannel = event.channel;
    receiveChannel.onmessage = (e) => console.log("Получено:", e.data);
    receiveChannel.onopen = () => receiveChannel.send("Привет от
получателя!");
  };
  // Создаем исходящий канал данных
  const channel = pc.createDataChannel("chat");
  channel.onopen = () => channel.send("Привет от инициатора!");
  channel.onmessage = (e) => console.log("Получено:", e.data);
  // Генерируем описание соединения (offer)
  pc.createOffer().then(offer => pc.setLocalDescription(offer));
  // Функция вызывается при нажатии кнопки, устанавливает удаленное описание
  function connect() {
    const remoteSDP = JSON.parse(remote.value);
    pc.setRemoteDescription(new RTCSessionDescription(remoteSDP));
  }
</script>
```

Пример: установление WebRTC DataChannel-соединения без сигналинг-сервера (обмен SDP производится вручную) [12] [13].

В этом коде два пользователя должны открыть одну и ту же страницу. Один нажимает "Создать соединение" (в нашем упрощенном примере оно создается сразу при загрузке, вызывается `pc.createOffer()`), после чего в поле "Локальное SDP" появляется текст – его нужно скопировать и передать второму пользователю любым способом. Второй пользователь вставляет этот текст в поле "Удаленное SDP" и нажимает кнопку "Установить соединение". После этого браузеры обменяются сетевой информацией и установят прямой канал. В консоли каждого можно увидеть обмен сообщениями: *"Привет от инициатора!"* и *"Привет от получателя!"*. Здесь мы реализовали минимальный чат P2P без посредников.

Конечно, вручную копировать SDP – неудобно. В реальных приложениях обычно присутствует *небольшой сигналинговый механизм*: это может быть выделенный сервер (который не имеет доступа к самим передаваемым данным, а только пересылает служебные сообщения), либо использование существующей инфраструктуры (например, отправлять SDP через WebSocket-соединение на публичный сервер или даже через электронную почту автоматически). Тем не

менее, сам медиаканал после установления связи работает без участия сервера – данные идут напрямую между клиентами.

Плюсы P2P-подхода с WebRTC:

- **Отсутствие центрального узла**: данные не проходят через сервер, что снижает задержки и исключает точку отказа. Два пользователя могут обмениваться информацией напрямую, достаточно иметь браузер с поддержкой WebRTC [14]. Это подходит для сценариев, когда разработчик не хочет или не может поднять сервер для передачи данных.
- **Конфиденциальность передачи**: WebRTC по умолчанию шифрует медиа- и датаграммы (DTLS/SRTP), поэтому содержимое защищено от прослушивания со стороны посторонних. А отсутствие промежуточного сервера означает, что негде вести централизованный лог трафика.
- **Высокая скорость для медиа**: прямое соединение позволяет стримить видео/аудио с минимальными задержками и хорошей пропускной способностью (ограничено лишь каналом связи между peer'ами).
- **Распределённая масштабируемость**: в некоторых приложениях (например, сети обмена файлами) P2P-схема масштабируется лучше, так как нагрузка распределена между участниками, а не концентрируется на сервере.

Минусы и ограничения P2P на фронтенде:

- **Сигнализация и подключение**: как мы видели, полностью без *какого-либо* сервера сложно обойтись, по крайней мере на этапе установления связи. Вручную копировать параметры соединения нереально в пользовательском приложении [15]. Обычно все же нужен сигналинговый сервер (пусть и небольшой и не имеющий доступа к самим данным) или использование публичных услуг для передачи SDP. Это добавляет компонент инфраструктуры.
- **Проблемы NAT и сетевые ограничения**: прямое соединение может не установиться, если оба участника находятся за NAT-брандмауерами. WebRTC требует STUN-сервер (для определения публичных адресов) и иногда TURN-сервер (для ретрансляции трафика, если прямой канал не удался). STUN-сервера доступны бесплатно (например, Google предоставляет stun.l.google.com), но TURN-сервер – это фактически сервер-посредник для передачи данных, что возвращает модель "клиент-сервер" при неблагоприятных сетевых условиях [16]. Настройка своего TURN-сервера сложнее и требует ресурсов.
- **Безопасность узлов**: каждый участник раскрывает свой IP-адрес другому участнику (иначе как они свяжутся?). Это может быть проблемой с точки зрения анонимности: пользователь A узнает IP пользователя B. В клиент-серверной модели IP пользователей скрыты друг от друга, а здесь – нет. Решение: использовать прокси (той же сети TOR для WebRTC пока нет штатно) или TURN-сервер, но тогда теряем прямой P2P.
- **Непостоянность соединения**: если приложение подразумевает, что данные должны быть доступны даже в отсутствие отправителя, P2P не поможет. Например, при видеозвонке P2P отлично подходит (оба участника онлайн одновременно). А вот для хранилища данных – нет, т.к. чтобы получить файл от соседа, сосед должен быть в сети. То есть P2P-сети без серверов лучше для **транзакционного обмена в режиме реального времени** (чаты, звонки, совместная работа), но плохо подходят для долговременного хранения (если не комбинировать с другими подходами).

Примеры и применение P2P во фронтенде:

- **Децентрализованные чаты и файлообменники**: существует, например, библиотека *WebTorrent* (реализация BitTorrent в браузере через WebRTC). Она позволяет пользователям обмениваться файлами прямо между браузерами. Ни один сервер не хранит сам файл – он раздается кусочками от одного клиента к другому. Нужно лишь где-то опубликовать торрент-файл или магнет-ссылку (можно через статический сайт). Подобным образом работает приложение ShareDrop для обмена файлами – оно использует WebRTC Data Channels: файлы передаются напрямую между браузерами без загрузки на сервер.
- **Совместные редакторы без центрального сервера**: благодаря WebRTC можно синхронизировать состояние приложения между несколькими клиентами. Например, можно реализовать совместный редактор текста или доску, где каждый браузер посылает изменения всем остальным через mesh-сеть соединений. Для разрешения конфликтов используют алгоритмы типа CRDT. Пример библиотек – *Automerge*, *Yjs*, которые могут работать p2p (хотя зачастую для удобства добавляют сервер-ретранслятор).
- **Локальные сети и WebRTC**: WebRTC может работать не только через интернет, но и в локальной сети (если устройства могут напрямую соединиться). Это открывает возможность коммуникации между устройствами близко друг к другу без выхода в интернет – например, веб-приложение может искать поблизости другие браузеры (через сервис-объявления, хотя это сложнее без сервера) и общаться с ними.

В целом P2P на фронтенде – отличное решение для тех случаев, когда нужно **прямое взаимодействие пользователей в реальном времени** без посредников. Однако для **хранения данных** или асинхронного взаимодействия (когда отправитель и получатель не одновременно онлайн) P2P-связи недостаточно – тут на помощь приходят децентрализованные сети хранения и блокчейн.

Децентрализованные базы данных и хранилища на фронтенде

Чтобы обеспечить **обмен данными между несколькими клиентами** или хранение данных без единого сервера, разработаны особые решения – **децентрализованные базы данных**, работающие поверх P2P-сетей. Одно из популярных решений – библиотека **GunDB (Gun.js)**.

GunDB – это peer-to-peer база данных (графовая), которая **синхронизирует данные между множеством браузеров** напрямую. Каждый узел сети (каждый браузер или нода) хранит у себя часть данных, и все узлы вместе образуют как бы единую базу [17]. Проще говоря, вместо одного сервера, хранящего данные, хранение распределяется между *участниками сети*. При этом Gun использует сложные алгоритмы мерджа (по сути, CRDT) чтобы изменения, внесённые разными узлами, сливались правильно.

Особенность GunDB в том, что она написана на JS и может работать **прямо в браузере** – не нужно устанавливать отдельный СУБД-сервер [18] [19]. Достаточно подключить скрипт Gun и можно сохранять данные:

```
<script src="https://cdn.jsdelivr.net/npm/gun/gun.min.js"></script>
<script>
  // Инициализация Gun (без указания peer-сервера)
  const db = Gun();
```

```
  // Создаем/получаем объект с ключом "123" и сохраняем в него данные
  const car = db.get("123").put({ make: "Toyota", model: "Camry" });
  // Считываем данные обратно
  car.once(data => console.log("Марка машины:", data.make)); // -> "Toyota"
</script>
```

Пример: использование GunDB в браузере для сохранения и получения данных без обращения к серверу [19].

В этом коде мы создаем локальный экземпляр Gun (`Gun()`). Затем методом `get("123")` обращаемся к узлу с ключом `"123"` (если его нет, он будет создан) и записываем объект с данными автомобиля через `put()`. Метод `once` позволяет один раз прочитать сохраненное значение (async, аналогично промису). Gun автоматически сохраняет данные в *локальном хранилище браузера* (IndexedDB) и пытается их синхронизировать с другими узлами сети, если они найдены.

Как происходит синхронизация? Если мы ничего не указали, GunDB работает в одиночку, используя только локальное хранилище. Чтобы несколько клиентов образовали сеть, они должны подключиться друг к другу. GunDB поддерживает различные транспорты: WebRTC, WebSocket и др. На практике часто используется гибридный подход: разворачивается *небольшой ретранслятор (relay peer)* – например, Node.js-сервер с `gun` на котором, к которому клиенты подключаются при запуске [20]. Этот узел не является традиционным бэкендом (он не выполняет бизнес-логику), но помогает хранить данные, когда ни один клиент онлайн не держит их, и пересылает изменения другим клиентам. Однако если пользователей достаточно и хотя бы один из них в сети содержит нужные данные, приложение могло бы работать и без постоянного узла [20]. Например, в полностью децентрализованной социальной сети каждый браузер хранит данные своих друзей, и сеть сама раздает необходимые фрагменты.

Безопасность и анонимность в GunDB: Эта библиотека изначально спроектирована с акцентом на приватность. В нее встроен модуль **SEA (Security, Encryption, and Authorization)**, позволяющий легко шифровать данные и управлять доступом к ним. По сути, GunDB может создавать для каждого пользователя пару ключей и шифровать все данные, принадлежащие этому пользователю, этим ключом [21] [22]. В результате даже если вы используете внешний ретранслятор, данные хранятся там в зашифрованном виде (подобно end-to-end шифрованию). Сам пользователь автоматически расшифровывает свои данные при входе, предоставив пароль. Таким образом, достигается *zero-knowledge*: даже админ узла-ретранслятора не узнает содержимого. Пользователи же могут делиться своими публичными ключами, чтобы другие могли безопасно записывать/читать часть данных.

Плюсы децентрализованных БД (на примере GunDB):

- **Нет центрального хранилища**: данные распределены между множеством узлов. Это устраняет единую точку отказа – приложение теоретически будет работать, пока онлайн есть хотя бы кто-то с необходимыми данными.
- **Масштабируемость и экономия**: нагрузка на хранение/трафик распределяется между участниками. Разработчику не нужно содержать мощный сервер и оплачивать большой трафик – пользователи сами вкладывают свой ресурс (память и сеть). Это соответствует принципам "Web 3.0".
- **Приватность и шифрование**: встроенная поддержка шифрования (SEA) позволяет обеспечить, что каждый пользователь контролирует доступ к своим данным [21]. Без его

ключа остальные видят лишь шифрованный мусор. Это решает проблему открытости данных, актуальную для публичных блокчейнов, о которой речь далее.
- **Offline-first**: приложения на Gun способны работать офлайн и синхронизироваться позже. Пользователь может внести изменения без подключения к сети; они сохранятся локально (IndexedDB), а когда появится связь – Gun распространит изменения другим узлам.
- **Конфликт-устойчивость**: Gun реализует механизм, сходный с CRDT – конфликтующие правки данных с разных узлов сливаются автоматически (по определенным правилам), что упрощает разработку коллаборативных приложений.

Минусы и сложности:

- **Сложность обеспечения постоянной доступности**: если *все* пользователи ушли офлайн, данные могут стать временно недоступны. Для надежности обычно хотя бы один узел должен быть онлайн постоянно (как раз роль ретранслятора). Его можно разместить на бесплатном хостинге или на компьютере энтузиаста, но полностью избежать узлов инфраструктуры трудно.
- **Консистентность и отладка**: отладка распределенной БД сложнее, чем классического сервера. При большом числе узлов возможны задержки в распространении обновлений, сложности с разрешением конфликтов данных.
- **Безопасность узлов**: раз данные частично хранятся у пользователей, возникает вопрос доверия – а не модифицирует ли злоумышленник свою копию данных злонамеренно? Шифрование решает только проблему конфиденциальности, но не целостности. GunDB предоставляет инструменты (подписи, сертификаты доступа) для верификации данных, однако разработчику нужно правильно их использовать. Концепция "доверяй, но проверяй" важна в любой P2P-системе.
- **Ограниченный выбор запросов**: децентрализованные БД обычно не такие гибкие в запросах, как SQL/NoSQL-серверы. Например, в Gun нет сложных запросов по условию – данные представляют граф или ключ-значение, и разработчик сам организует структуру. Это цена за распределенность.

Применение: GunDB и похожие решения подходят для **социальных сетей**, **чатов**, **совместных документов**, **игр** – всего, где есть взаимодействие пользователей и нужно хранить состояние без центрального сервера. Практический пример – прототип "децентрализованного Google Docs" на Gun: каждый участник редактирует документ, правки распространяются по P2P-сети, а GunDB заботится о слиянии изменений [17] [20]. Конечно, для продакшна обычно комбинируют с каким-то постоянным узлом, но даже тогда разработчик не управляет традиционной БД – узел лишь один из равных peer'ов.

Альтернативные проекты: кроме Gun, существуют *OrbitDB* (поверх IPFS), *Scuttlebutt*, *Hypercore protocol (Dat)* и др. Все они несколько разные по модели, но цель общая – **убрать центральный сервер из хранения данных**.

Блокчейн как бэкенд (децентрализованные приложения)

Особого внимания заслуживает использование **блокчейн-технологий на фронтенде**. В контексте нашего обсуждения блокчейн можно рассматривать как альтернативу традиционному бэкенду. Если обычное веб-приложение хранит данные в своей базе данных на сервере, то **децентрализованное приложение (DApp)** хранит данные и логику в блокчейне, а фронтенд выступает лишь интерфейсом к этой децентрализованной системе [23].

Например, возьмем блокчейн Ethereum: разработчик может написать смарт-контракт – программу, которая будет выполняться на блокчейне и хранить какие-то данные (в своем состоянии). Затем контракт разворачивается в сеть. После этого **весь необходимый бэкенд-функционал доступен через блокчейн**: фронтенд (работающий локально или как статический сайт) взаимодействует с контрактом, вызывая его функции через специальные библиотеки (web3.js, Ethers и т.п.). **Сервер приложению не нужен**, ведь *блокчейн – и есть распределенный сервер*, поддерживаемый тысячами узлов по всему миру.

Пользователь взаимодействует с DApp через **криптокошелек** (например, расширение MetaMask или аналог). Кошелек хранит его закрытый ключ (который заменяет логин/пароль) и позволяет подписывать транзакции. Таким образом, **криптокошелек – это и средство аутентификации, и средство оплаты операций**. Для нашего фронтенда кошелек предоставляет API для подключения.

Рассмотрим небольшой пример кода на фронтенде, использующего библиотеку Ethers.js для связи с Ethereum-блокчейном:

```
<script src="https://cdn.ethers.io/lib/ethers-5.2.umd.min.js"></script>
<script>
  // Подключаемся к провайдеру Ethereum через MetaMask (в браузере должен
быть установлен MetaMask)
  const provider = new ethers.providers.Web3Provider(window.ethereum);
  await provider.send("eth_requestAccounts", []); // Запрос доступа к
аккаунту пользователя
  const signer = provider.getSigner();            // Получаем "подписанта" –
привязанный к кошельку аккаунт

  // Подключаемся к смарт-контракту по адресу и ABI (описывает интерфейс
функций контракта)
  const contractAddress = "0x...";      // адрес развернутого контракта
  const contractABI = [ /* ABI контракта (массив объектов функций) */ ];
  const contract = new ethers.Contract(contractAddress, contractABI, signer);

  // Вызовем чтение данных из контракта (не требующее транзакции)
  const value = await contract.getValue();
  console.log("Данные из контракта:", value);

  // Вызовем изменение состояния контракта (требует подписи и отправки
транзакции)
  const tx = await contract.setValue(42);
  await tx.wait(); // ждем подтверждения транзакции майнерами
  console.log("Значение обновлено на 42 в блокчейне");
</script>
```

Пример: взаимодействие фронтенда с блокчейн-контрактом через провайдер MetaMask (ethers.js) [24].

Здесь фронтенд, запущенный в браузере, подключается к блокчейну Ethereum через **провайдер**, предоставляемый MetaMask (`window.ethereum`). Мы запрашиваем у пользователя разрешение (он должен подтвердить во всплывающем окне MetaMask, что разрешает странице видеть его

адрес и выполнять транзакции). Затем получаем объект `signer`, представляющий аккаунт пользователя, которым можно подписывать операции. С его помощью создаем объект `contract` – абстракцию смарт-контракта по заданному адресу. Далее пример демонстрирует вызов функции `getValue()` (допустим, контракт хранил какое-то число) – эта операция бесплатна и не меняет состояние, она просто читает данные из блокчейна (т.е. все узлы хранят копию, и мы читаем из локальной ноды или удаленного RPC). Вторая операция `setValue(42)` – изменяет состояние контракта (допустим, сохраняет число 42). Эта операция формирует транзакцию, которую пользователь подписывает своим ключом и отправляет в сеть. Дальше уже майнеры/валидаторы включают транзакцию в блок, после чего новое значение будет закреплено в блокчейне. Мы дожидаемся подтверждения (`tx.wait()`), чтобы знать, что сеть приняла изменения.

Обратите внимание, здесь **нет ни одной нашей серверной функции** – все взаимодействие идет через публичный блокчейн. Мы лишь *пользуемся инфраструктурой блокчейна*, подобно тому как пользуемся, скажем, сетью Bitcoin для перевода средств.

Плюсы подхода “блокчейн вместо бэкенда”:

- **Нулевая настройка и сопровождение**: разработчику не нужно поднимать серверное приложение, базу данных, обеспечивать их работу 24/7. Смарт-контракт, развернутый в блокчейне, автоматически доступен постоянно, пока жив сам блокчейн [25]. У популярных сетей аптайм очень высокий, часто близок к 100%, особенно для чтения данных [26]. Таким образом, *блокчейн работает как ваш бекенд* с гарантированной доступностью – даже если вы перестанете поддерживать проект, данные и функции не исчезнут [27] [28].
- **Отсутствие центральной точки отказа и цензуры**: данные на блокчейне хранятся децентрализованно (на множестве узлов). Никто не может единолично отключить ваше приложение, удалить или подменить данные – они защищены криптографически. Пользователи могут доверять, что правила, заложенные в смарт-контракт, будут выполняться точно, без произвольных изменений (код контракта открыт и проверяем). Это повышает доверие пользователей к приложению [29] [30].
- **Прозрачность и проверяемость**: практически все данные в публичном блокчейне открыты (если не зашифрованы намеренно). Любой желающий может посмотреть историю транзакций, текущее состояние контракта и убедиться, что приложение работает честно. Для пользователя это дополнительная гарантия прозрачности (важно для финансовых приложений, голосований и т.д.). *Примечание:* если приложение требует приватности, данные можно шифровать перед отправкой в блокчейн, тогда в блокчейне будет храниться шифрованный текст [11].
- **Встроенная идентификация пользователей**: как упоминалось, блокчейн использует криптографические адреса (кошельки) вместо традиционных учетных записей. Для DApp не нужно реализовывать регистрацию/логин – достаточно, чтобы у пользователя был кошелек. Аутентификация сводится к проверке цифровой подписи. При этом **пользователь остается анонимен**, если не привязал свой адрес к реальной личности: для использования DApp не надо вводить email, имя или другие персональные данные, достаточно криптоключа [31].
- **Разграничение доступа и права пользователей**: смарт-контракты могут быть написаны так, что части данных доступны или модифицируемы только определенными пользователями (по адресам). Например, только владелец записа может ее менять, или любой может просмотреть, но изменить – заплатив определенную комиссию и т.п. Все эти правила хранятся в блокчейне, что опять же не требует отдельного администрирования.
- **Пользователи сами несут расходы на вычисления**: интересная особенность – в традиционном приложении разработчик платит за серверное время и БД, а пользователь

платит лишь своим вниманием или данными. В блокчейне же, **за выполнение операций платит инициатор транзакции**. Т.е. если пользователь хочет записать что-то в вашу dApp (изменить состояние контракта), он заплатит комиссию (gas fee). Обычно комиссии небольшие (зависят от сети), например на одной из популярных сетей (Polygon) запись 32 байт данных может стоить доли цента [32]. Это **перекладывает издержки** с разработчика на пользователя, хотя, конечно, может оттолкнуть часть аудитории. В сообществе блокчейна такой подход принят – многие пользователи согласны платить небольшие комиссии за использование децентрализованных сервисов [33]. Для разработчика плюс – приложение может масштабироваться, а он не разорится на серверных счетах.

- **Богатая экосистема и интеграции**: данные на блокчейне *совместимы* с другими приложениями. Например, если ваше приложение хранит токены или NFT, другие dApps могут легко взаимодействовать с ними – торговать на биржах, отображать в кошельках пользователей и т.д. Ваш бэкенд по сути становится **частью единого глобального состояния**, и можно опереться на уже существующие библиотеки и сервисы (например, использовать смарт-контракты стандарта ERC-20/ERC-721 для выпуска своих объектов, вместо написания с нуля). Существуют публичные API (Infura, Alchemy и т.п.) и ноды, которые позволяют бесплатно или дешево читать данные из блокчейна, подписывать события и др. [34] – разработчику не надо хостить даже узел блокчейна, можно воспользоваться этими провайдерами (хотя это уже не столь децентрализовано, но удобно).

Минусы и ограничения блокчейн-подхода:

- **Комиссии и стоимость хранения**: хотя чтение данных из блокчейна обычно бесплатное, **запись данных стоит денег**. Блокчейн – удовольствие не из дешевых, особенно если пытаться хранить много информации. Хакернун называет хранение даже пары байтов *дорогостоящим*, например, запись 32 байт на Ethereum mainnet может стоить десятки центов или долларов (в зависимости от загрузки сети). На более дешевых сетях (Polygon, BSC) это копейки, но все равно **хранение больших объёмов невыгодно** [32]. Блокчейн лучше всего подходит для малого критичного объёма данных (например, хеши файлов, ссылки, балансы, права доступа). Широко распространен подход, когда в блокчейне хранят **только ссылки или хеши**, а сами файлы хранятся в IPFS или другом хранилище (об этом – далее).
- **Ограниченная производительность и задержки**: децентрализованный консенсус медленнее обычной базы. Транзакции в блокчейне требуют времени на подтверждение (от нескольких секунд до минут, в зависимости от сети). Это не подходит для некоторых интерактивных сценариев (нельзя сделать блокчейн-чат, отправляя каждое сообщение транзакцией – ждать 10 секунд никто не будет). Смарт-контракты также ограничены по вычислительной сложности – за слишком тяжелые вычисления придется много заплатить, либо они вообще не пройдут из-за лимитов по газу [35]. Поэтому сложные расчеты на блокчейне не делают – их выносят на клиент или используют специальные решения (off-chain вычисления, Layer2). Для большинства dApps это не проблема (им хватает возможностей языка Solidity/Rust для базовой логики), но например, реализовать видеокодер или машинное обучение на блокчейне невозможно.
- **Публичность данных (проблема приватности)**: все записи в публичном блокчейне видны всем. Если приложение предполагает хранение приватных данных, это противоречит природе блокчейна. Решения есть – как сказано, можно шифровать данные перед помещением в блокчейн, однако тогда возникает проблема управления ключами и возможности поиска по зашифрованным данным (почти нет). Есть и приватные блокчейны или L2 с шифрованием (например, Secret Network, где данные в контракте могут быть секретны). Но большинство популярных блокчейнов – прозрачны. С точки

зрения анонимности: пользователь может не раскрывать свою личность, но все его действия по адресу видны. Если адрес как-то скомпрометирован (раскрыта связь с личностью), вся история становится явной. Поэтому DApp'ы не гарантируют анонимность в строгом смысле, они дают лишь псевдонимность. Для повышения анонимности нужно использовать дополнительные инструменты (миксеры, анонимные сети как TOR, специальные privacy-коины).

- **Порог входа для пользователя**: чтобы воспользоваться dApp, пользователю нужен криптовалютный кошелек, а иногда и на счету должно быть немного криптовалюты для оплаты комиссий. Для неподготовленной аудитории это может стать препятствием (нужно разобраться, установить расширение, купить крипту). Хотя уже есть решения, позволяющие абстрагировать эти детали (метатранзакции, когда разработчик спонсирует газ, или социальные кошельки и т.п.), классический DApp все же требует чуть больше усилий от пользователя, чем Web2-приложение с регистрацией по email.
- **Сложность разработки и отладки**: написать смарт-контракт – задача, требующая внимания к безопасности. Ошибка в контракте может стоить очень дорого (невозможность исправить баг после деплоя, потеря средств). Инструменты для разработки постоянно улучшаются, но все равно разработчику фронтенда придется выучить новый стек (Solidity/Rust, адреса, ABI, работа с кошельками). Хотя, как отмечают специалисты, порог входа не так страшен – концепции разработки похожи на привычные, нужно просто сменить майндсет [36]. Тем не менее, DApp-разработка – отдельная область со своими нюансами.

Применение блокчейна на фронтенде:

- Классический пример – **криптовалютные кошельки и обменники**, где фронтенд показывает баланс с блокчейна, формирует транзакции для перевода средств и подписывает их.
- **Децентрализованные финансы (DeFi)**: приложения, позволяющие пользователям вкладывать средства, брать кредиты, обменивать токены – все операции происходят через смарт-контракты, а сайт лишь визуализирует и дает удобный интерфейс. Например, Uniswap – обменник токенов, у которого фронтенд – это статический React-приложение, общающееся с контрактами Uniswap в Ethereum. Пользователь через фронтенд посылает транзакцию, а обмен выполняется контрактом, ликвидность хранится децентрализовано.
- **NFT-галереи и маркетплейсы**: отображают данные NFT (которые хранятся в блокчейне), позволяют купить/продать NFT, опять же вызывая функции контрактов (передача токена, перевод денег).
- **Игры на блокчейне**: фронтенд рисует игру, а состояние (владение объектами, результаты раундов) записано в блокчейне.
- **Голосования, DAO**: когда нужно, чтобы результаты голосования были проверяемы и неизменны, делают смарт-контракт для сбора голосов, а фронтенд – для удобства голосующих. Никто не сможет нарисовать лишние голоса, т.к. контракт этого не позволит, и любой может проверить в блокчейне.
- **Обмен сообщениями**: простейший чат на блокчейне возможен, но дорого и публично. Однако есть проекты по децентрализованным зашифрованным мессенджерам, которые используют блокчейн для установления соединения или обмена ключами, а дальше переходят на p2p-каналы.

В контексте нашей темы – блокчейн позволяет **полностью избавиться от своего сервера**, но зачастую комбинируется с децентрализованным хранением файлов.

Децентрализованное хранение файлов (IPFS и альтернативы)

Блокчейны плохо подходят для хранения больших данных (изображения, видео, архивы и т.д.) из-за вышеописанных ограничений. Поэтому появились децентрализованные сети хранения, **ориентированные на файлы**. Самая известная – **IPFS (InterPlanetary File System)**. IPFS – это P2P-сеть, в которой файлы адресуются не по месту (URL сервера), а по содержимому (хеш). Файл, добавленный в IPFS, получает уникальный *CID* (контент-идентификатор, похожий на хеш). Этот CID можно публиковать – по нему любой узел IPFS может загрузить файл из сети [37].

Как это работает: когда вы добавляете файл в IPFS (например, через локальный IPFS-нод или через API), он распространяется по сети. Узлы могут добровольно «закреплять» (pin) контент – т.е. хранить его у себя на диске постоянно. Если файл нигде не закреплен и все узлы, державшие его, отключатся, то при попытке снова его запросить – сеть его не найдет. Поэтому, чтобы файл был **доступен постоянно**, обычно используют **pinning сервисы** – например, *Pinata, Infura IPFS, Eternum* и т.д., которые за бесплатно или небольшую плату хранят ваши файлы на своих узлах [38] [39]. Это не полностью децентрализовано (мы доверяем конкретному сервису хранить), но вы можете закрепить копии на нескольких сервисах + у себя, тогда исчезновение одного не приведет к потере данных [40] [37].

Для **фронтенд-разработчика** IPFS интересна тем, что **позволяет загружать и скачивать файлы напрямую из браузера без собственного сервера хранения**. Существует клиентская библиотека `ipfs-http-client` и даже возможность запускать легковесный IPFS-нод в самом браузере (js-ipfs). Вот пример кода, который запускает IPFS-нод в браузере и добавляет файл:

```
// Запуск IPFS нода (понадобится ipfs-core библиотека)
const IPFS = require('ipfs-core');
const ipfs = await IPFS.create();
const file = document.querySelector('input[type=file]').files[0];
const { cid } = await ipfs.add(file);
console.log("Файл добавлен в IPFS, CID:", cid.toString());
```

Пример: добавление файла в сеть IPFS прямо из фронтенда [41].

Этот код создает локальный IPFS-узел внутри страницы. Когда мы вызываем `ipfs.add(file)`, файл разбивается на блоки, хешируется, и блоки начинают распространяться. CID можно сохранить – это идентификатор, по которому другие смогут файл скачать. Но важный момент: **когда пользователь закроет страницу, его IPFS-узел остановится** и он перестанет раздавать файл [42]. Если больше *никто* в сети его не держит, файл станет временно недоступен. Поэтому, как обсуждалось на форуме IPFS, без бэкенда проблему постоянного хранения можно решить либо побуждая пользователей держать страницу открытой, либо добавляя файл на какие-то публичные узлы [43] [44]. Обычно поступают проще: после `ipfs.add` вызывают API pinning-сервиса через HTTP, чтобы тот закрепил CID.

Можно пойти и другим путем: **не запускать IPFS-нод в браузере, а использовать IPFS API-шлюз**. Есть публичные шлюзы (например, gateway.pinata.cloud, cloudflare-ipfs.com) через которые можно выполнить HTTP-запрос `POST /api/v0/add` с файлом (вот только они не всегда открыты для всех). Другой вариант – использовать *decentralized storage services* вроде *Web3.storage* или *NFT.storage*, где фронтенду достаточно отправить файл вместе с токеном доступа, и сервис сам

добавит его в IPFS и закрепит. Такие сервисы предоставляют ограниченное бесплатное хранение (за счет децентрализации на Filecoin, иным образом монетизируемое).

Применение IPFS и плюсы:

- **Статический хостинг сайтов**: IPFS позволяет хостить целый сайт (HTML/CSS/JS). Достаточно загрузить папку сайта в IPFS, получить корневой CID и через шлюз или специальный IPNS-ссылку сделать его доступным. Это значит, что ваш фронтенд тоже может быть децентрализованным! Пользователи могут загружать его из IPFS сети. Уже сейчас возможно привязать даже доменное имя через DNSLink к IPFS-хэшу и иметь сайт без традиционного хостинга. Такой сайт, разумеется, должен быть **статическим** – только фронтенд, никакого серверного кода [45].
- **Хранение пользовательского контента**: если ваше фронтенд-приложение позволяет пользователям загружать файлы (например, картинки, видео), вы можете направлять их сразу в IPFS вместо своего сервера. Пользовательский контент тогда будет раздаваться P2P. Это снимает нагрузку с вашего сервера и повышает устойчивость – популярные файлы могут кэшироваться на многих узлах.
- **Контент-адресуемость и интегритет**: благодаря тому, что адрес файла – это его хеш, пользователь может проверить целостность: если файл скачался и его хеш совпадает с ожидаемым CID, значит файл не искажен. Также одинаковые файлы не дублируются в сети (экономия места).
- **Отсутствие цензуры**: нет единой точки, через которую проходит весь трафик. Если файл закреплен в нескольких местах, удалить его полностью сложно – подобно торрент-сети. В контексте анонимности: публикация через IPFS может быть более анонимной, чем загрузка на конкретный сервер, так как нет регистрации – но все равно IP-адрес узла, добавляющего контент, может быть виден другим узлам. Решить можно использованием VPN или TOR (есть проекты по интеграции IPFS и TOR).

Недостатки IPFS и децентрализованного хранения:

- **Не мгновенная доставка**: если контент не популярен, сначала должен найтись узел, у которого он есть, и скачать от него. Это может быть медленнее, чем от CDN или сервера. Однако для популярных данных IPFS может быть очень быстрым, так как ближайшие к вам узлы могут иметь копию.
- **Нет гарантий хранения**: если вы сами (или с помощью сервиса) не закрепите контент, он может быть забыть сетью. IPFS не гарантирует хранение – только адресацию и обмен. Это отличие от блокчейна, где данные навечно вписаны (но блокчейн дорогостоящ для файлов).
- **Пиннинг-сервисы**: по сути возвращают элемент централизации (вы доверяете Pinata или другому сервису хранить файлы). Но у вас остается возможность в любой момент мигрировать – т.к. данные адресуются хешем, один и тот же CID можно закрепить на другом сервисе или своей ноде, и приложения продолжат его находить.
- **Безопасность доступа**: данные в IPFS по умолчанию публичны (если знаешь CID, можно скачать). Нет встроенных средств ограничить доступ, так как сеть не управляет правами. Если нужно приватно хранить файлы, их обычно **шифруют перед добавлением** (аналогично принципам, описанным ранее). Тогда даже получив CID, никто без ключа не узнает содержимого файла. Организовать же выборочную раздачу можно на прикладном уровне (например, не раскрывать CID неавторизованным, или использовать шифрование с разными ключами).

Комбинация блокчейн + IPFS (и фронтенд): В Web3-приложениях часто фронтенд хранится на IPFS (статический), важные данные/метаданные – на блокчейне, а массивные объекты (файлы) – в

IPFS. Блокчейн может хранить лишь хеш файла (CID) или ссылку на него. Например, NFT токены обычно хранят свои изображения именно как IPFS CID, записанный в блокчейне. Таким образом, ни один сервер не контролирует полностью приложение: фронтенд можно получить от сети, данные проверить по блокчейну, файлы скачать P2P.

Для нашего сценария (разработка без бэкенда и с сохранением анонимности) IPFS полезна тем, что **позволяет публиковать контент, не раскрывая свою личность**. Можно запустить IPFS-нод под любым псевдоустроем, и он будет частью сети. Нет центрального регистратора. Однако стоит помнить: действия в IPFS сейчас не привязаны к крипто-адресам (в отличие от блокчейна), поэтому аутентификацию автора контента нужно решать отдельно, если требуется (например, подписывать хеши файлов приватным ключом и публиковать эту подпись где-то).

Практические сценарии и заключение

Мы рассмотрели ряд технологий и подходов, позволяющих реализовать веб-приложение *только на фронтенде*, с минимальным использованием внешней инфраструктуры. На практике, многие реальные приложения сочетают эти методы для достижения лучшего результата. Например:

- **Личный офлайн-блокнот**: Вся логика и UI – фронтенд (может быть PWA, чтобы работать офлайн). Данные (заметки) хранятся в IndexedDB, **зашифрованы** ключом, производным от пароля пользователя. При этом пользователь может экспортировать резервную копию вручную. Бэкенд не нужен вовсе – ни для хранения, ни для авторизации. Анонимность полная, т.к. данные никогда не покидают устройство.
- **P2P-чат или файлообмен без сервера**: Используется WebRTC для прямого соединения между пользователями. Фронтенд может быть хоститься как статическая страница (например, на GitHub Pages или IPFS). При открытии страницы пользователи обмениваются сигналами через условный канал (можно даже вручную вводить код-собеседника, либо использовать публичный сигналинговый сервис без сохранения данных). После установления WebRTC-канала сообщения и файлы идут напрямую, **сквозное шифрование** можно обеспечить на уровне приложения (например, обменявшись заранее публичными ключами собеседников).
- **Децентрализованный блог/форум**: Фронтенд – PWA, которое при старте подключается к децентрализованной базе (GunDB, OrbitDB). Все посты и комментарии хранятся распределенно у читателей и писателей. Для надежности, сообщество может держать несколько постоянно включенных узлов. Каждый пост может подписываться ключом автора (а публичный ключ автора можно привязать к его профилю). Таким образом, нет центрального сервера-модератора, каждый участник хранит кусочек информации. Анонимность зависит от того, как распространяются ключи – можно быть под псевдонимом.
- **DApp для голосования**: Фронтенд – статический React на IPFS. Авторизция – через криптокошелек (Metamask). Голоса отправляются на смарт-контракт в блокчейне (например, на Ethereum или более дешевом Polygon) – таким образом, ни одна центральная сторона не может подделать результат или узнать, как проголосовал конкретный кошелек (если голос зашифрован и потом расшифровывается совокупно). Подсчет происходит либо на контракте, либо off-chain, но проверяемо. Анонимность голосующего сохраняется на уровне кошелька (если он не ассоциирован лично).
- **Комбинация IPFS и блокчейна для приложений**: например, фотогалерея, где пользователи публикуют фото. Фото загружается через фронтенд прямо в IPFS (через API), получает CID. Фронтенд затем вызывает транзакцию к смарт-контракту `publishPhoto(CID, description)` – тот сохраняет CID (и, например, привязывает к адресу автора, времени). Другие пользователи через контракт могут получить список CID

фотографий, затем через IPFS-гейтвей или локальный IPFS-нод скачать сами файлы. Нет центрального сервера: фронтенд из IPFS, данные (список фото) – из блокчейна, файлы – тоже распределенно. Единственные затраты – газ за публикацию метаданных и возможно оплата пиннинга фото.

Выводы: Разработка веб-приложений без собственного сервера стала реальностью благодаря развитию браузерных технологий и децентрализованных сетей. Используя локальное хранение, сервис-воркеры и PWA, можно создавать автономные приложения. С помощью шифрования и p2p-связи – обеспечивать безопасный обмен данными напрямую между пользователями, сохраняя конфиденциальность. А благодаря блокчейнам и распределенным БД – реализовывать **полноценные серверные функции "на фронтенде"**, переложив хранение и вычисления на децентрализованную инфраструктуру.

Конечно, полностью избежать любых внешних сервисов сложно – зачастую нужен хотя бы минимальный «координатор» (сигналинг-сервер для WebRTC, ретранслятор для P2P БД, узлы IPFS, RPC-узлы блокчейна). Однако ключевое отличие в том, что **эти сервисы не находятся под полным контролем разработчика и не хранят пользовательские данные в открытом виде**. Это значит, что приложение масштабируется без классических затрат (многие из таких сервисов предоставляются сообществом или по модели freemium), а пользователь получает больше контроля над своими данными и приватностью.

При создании фронтенд-онли приложений важно тщательно продумать **криптографию, пользовательский опыт и угрозы безопасности**, ведь ответственность за данные теперь распределена. Но при правильном подходе такие приложения открывают двери к более свободному и безопасному вебу, где **анонимность и децентрализация** – не только идеология, но и практический архитектурный выбор.

Источники:

- Виктор Огонна. *Protecting User Data: Encryption and Secure Storage in Frontend* – шифрование данных на фронтенде, AES/RSA примеры, хранение в localStorage и IndexedDB [2] [6] [4].
- HexShift. *Building a Minimal WebRTC Peer Without a Signaling Server* – пример установления P2P соединения WebRTC без сервера, код и разбор плюсов/минусов [12] [13] [46] [16].
- Codesphere Team. *Decentralized Google Docs with React and GunDB* – описание GunDB, принцип работы P2P-базы, необходимость ретранслятора [17] [20].
- Fleek. *Welcome GunDB to our Open Web Protocol Stack!* – интеграция GunDB, упор на приватность, шифрование через SEA [47] [22].
- Alex Kit. *Blockchain as a Backend* (HackerNoon, 2025) – плюсы и минусы использования блокчейна вместо сервера (на примере EVM) [25] [48] [32] [35].
- QuickNode Guides. *Web3 Development and Blockchain as Backend* – сравнение Web2/Web3, концепция блокчейна как бэкенда [23].
- Jaco V. *Hosting a Decentralized Website with IPFS* – практические аспекты размещения сайтов и контента в IPFS, требования статичности фронтенда [45] [37].
- IPFS Forum – обсуждение загрузки файлов в IPFS с фронтенда, возможности js-ipfs в браузере и необходимость пиннинга для постоянного хранения [41] [42].

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] Protecting User Data: Encryption and Secure Storage in Frontend - DEV Community

https://dev.to/outstandingvick/protecting-user-data-encryption-and-secure-storage-in-frontend-53ak

11 25 26 27 28 29 30 31 32 33 34 35 36 48 Is Developing Blockchain as a Backend Really All That Hard? | HackerNoon
https://hackernoon.com/is-developing-blockchain-as-a-backend-really-all-that-hard

12 13 14 15 16 46 Building a Minimal WebRTC Peer Without a Signaling Server (Using Only Manual SDP Exchange) - DEV Community
https://dev.to/hexshift/building-a-minimal-webrtc-peer-without-a-signaling-server-using-only-manual-sdp-exchange-mck

17 20 Set Up a Decentralized Database with React and GunDB
https://codesphere.com/articles/set-up-a-decentralized-database-with-react-and-gundb

18 19 GunDB: a Graph Database in JavaScript
https://frontendgods.com/gundb-a-graph-database-in-javascript/

21 22 47 Welcome GunDB to our Open Web Protocol Stack! - DEV Community
https://dev.to/fleek/welcome-gundb-to-our-open-web-protocol-stack-ic7

23 How to Become a Web3 Developer Roadmap | QuickNode Guides
https://www.quicknode.com/guides/web3-fundamentals-security/how-to-become-a-web3-developer-roadmap

24 Getting Started
https://docs.ethers.org/v5/getting-started/

37 38 39 40 45 Hosting a Decentralized Website with IPFS | Medium | Medium
https://jacovan.medium.com/hosting-a-decentralized-website-with-ipfs-7cbc0fc2cef5

41 42 43 44 How to upload file to IPFS with only front-end - IPFS Forums
https://discuss.ipfs.tech/t/how-to-upload-file-to-ipfs-with-only-front-end/11277