SSR

- Серверная сборка
- Context выполнения
- Утечка памяти при CSR
- Утечка памяти при SSR
- Async local storage:
- Механизмы SSR
- Router
- State
- SEO
- Полный рецепт

История рендера

Static HTML (статический HTML)

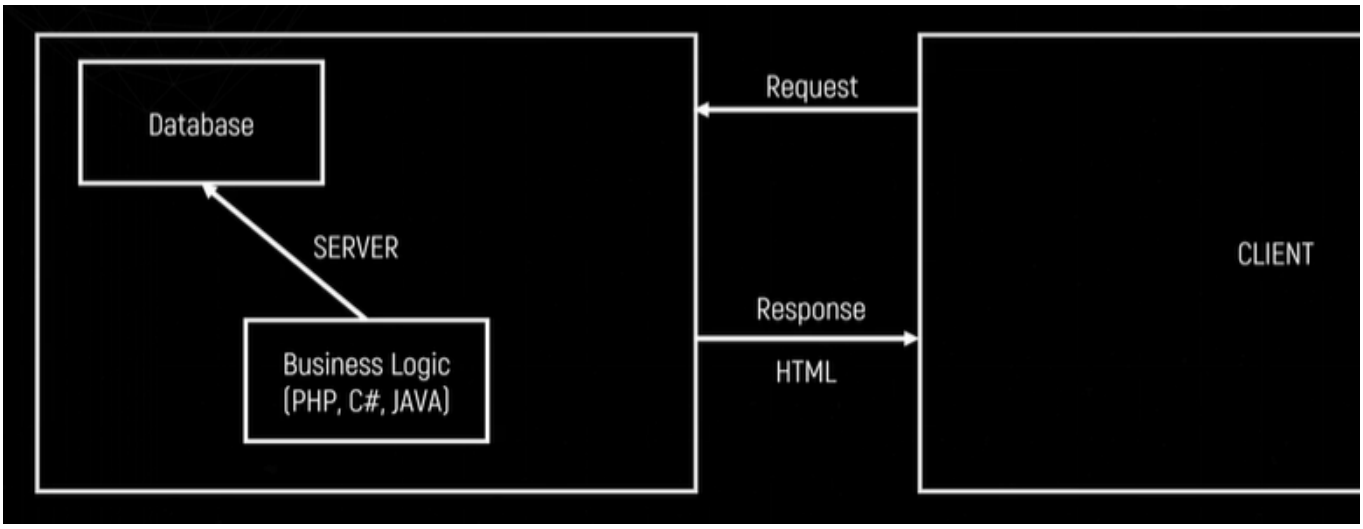
Что это такое:

- HTML-файл полностью написан вручную.
- Сервер просто отдаёт файл «как есть» клиенту.
- Нет логики, нет данных, всё жёстко зашито в коде, нет отдельных css и js файлов.

```
<MULTICOL COLS="3" GUTTER="25">
  <P><FONT SIZE="4" COLOR="RED">This would be some font broken up into
columns</FONT></P>
</MULTICOL>
```

Static HTML — это самый ранний этап развития веба. Примерно 1991–1996 годы.

Server render на заре



Этот слайд как раз иллюстрирует **Server-Side Rendering (SSR)** в его классическом виде, который появился **после эпохи статических HTML-файлов**, примерно в 1996–2010
Все делается внутри монолитного сервиса. То есть был html + php вставки кода.
Современный php выглядит уже не так.
Этот серверный рендеринг существовал прям на заре интернета.

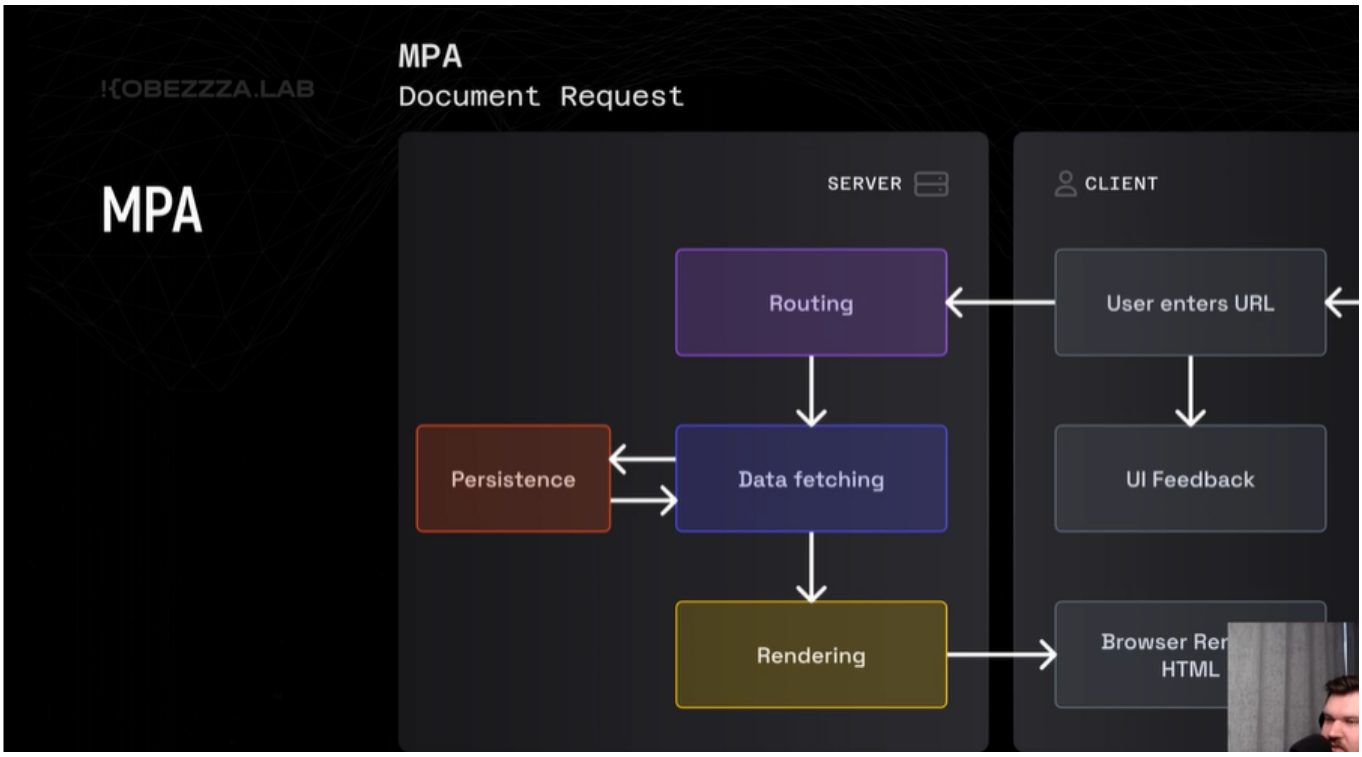
Что происходит на этой схеме:

Цикл запроса-ответа:

1. **Client** отправляет **HTTP-запрос** (например, пользователь открывает `/products/123`).
2. **Server**:
   - Сначала **обрабатывает бизнес-логику** (например, на PHP, C#, Java).
   - Делает **запрос в базу данных**.
   - Генерирует **HTML на лету** с реальными данными.
3. Сервер отправляет **готовую HTML-страницу обратно клиенту**.

MPA (Multi-Page Application)

Это архитектура, при которой **каждая страница сайта — отдельный HTML-документ**, получаемый от сервера при переходе по URL.



Вся логика, которая есть на странице выполняется на сервере. Отлично работает без JS. Хорошая SEO-оптимизация по умолчанию. Полная перезагрузка страницы при каждом переходе.

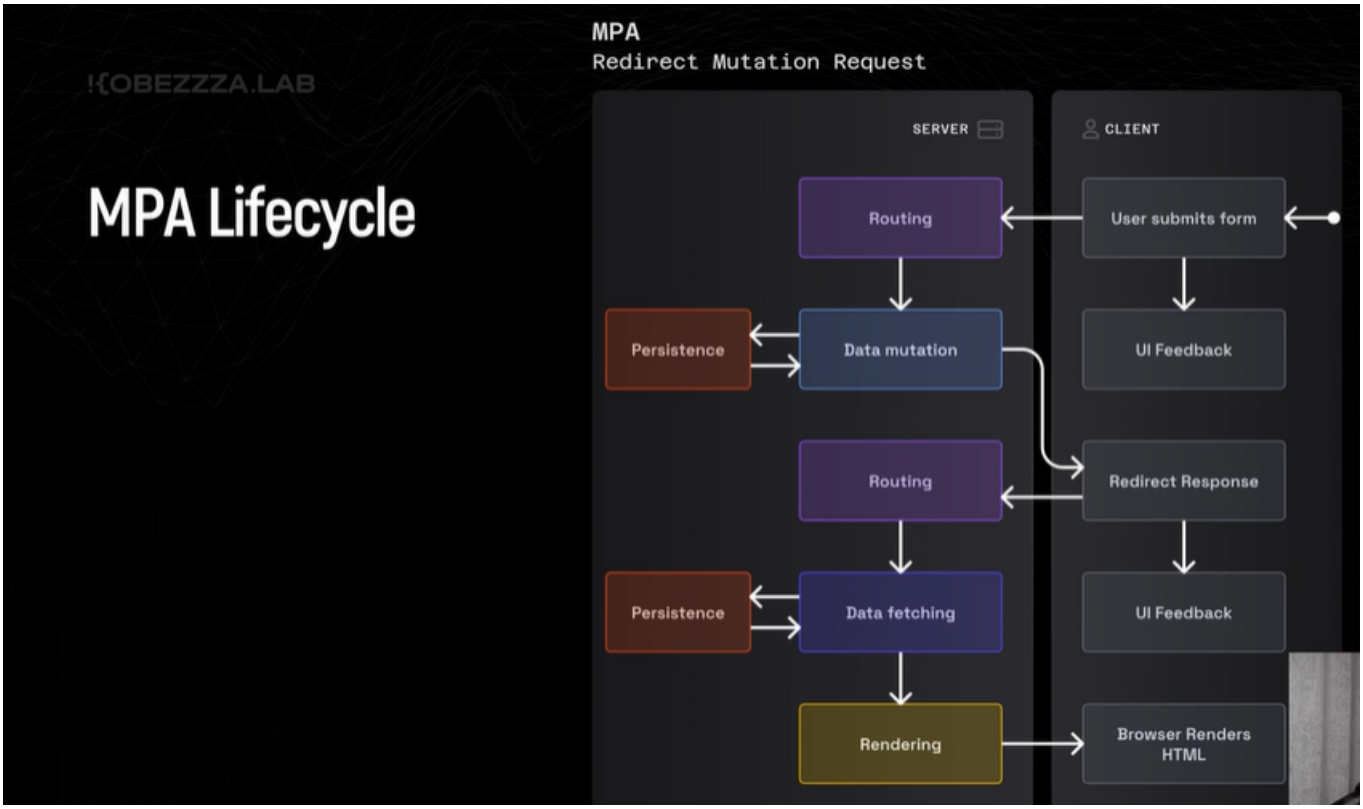
Что происходит на этой схеме:

Клиент:

- Пользователь вводит URL или кликает по ссылке.
- Браузер отправляет **HTTP-запрос на сервер**.
- Пока идёт запрос — браузер может показать какую-то **UI Feedback** (например, спиннер).
  Сервер:
- **Routing** — сервер определяет, какую страницу пользователь запрашивает.
- **Data fetching** — запрашиваются данные (например, из БД).
- **Persistence** — взаимодействие с базой или хранилищем.
- **Rendering** — генерируется HTML.
- HTML отправляется обратно клиенту.
  Клиент:
- Браузер получает **новый HTML-документ** и перерисовывает всю страницу целиком.

MPA Lifecycle (жизненный цикл MPA)

При **мутации данных** — например, когда пользователь **отправляет форму**.



Жизненный цикл MPA: Redirect Mutation Request. Это типичный сценарий: форма → сохранение данных → редирект → отрисовка новой страницы.

Что происходит на этой схеме:

Клиент (правая часть):

- **User submits form** — пользователь отправляет форму.
- **UI Feedback** — браузер может показать индикатор загрузки (спиннер).
- Отправляется **POST-запрос на сервер**.
  Сервер (левая часть):
- **Routing** — сервер определяет, какая мутация должна выполниться.
- **Data mutation** — происходит изменение данных (например, запись в базу).
- **Persistence** — запись в базу данных.
  Далее **редирект**:
- Сервер **отправляет redirect response** (обычно `302` или `303`), чтобы пользователь не пересылал форму повторно при обновлении страницы.
  Повторный запрос (GET):
- Снова **Routing** — теперь на новую (или ту же) страницу.
- **Data fetching** — данные подгружаются (например, актуальный список).
- **Persistence** — читается из базы.
- **Rendering** — генерируется HTML-страница.

- **Browser Renders HTML** — клиент получает и отображает обновлённую страницу.

Почему редирект? - Это классический паттерн **PRG (Post/Redirect/Get)**: Чтобы избежать повторной отправки формы при обновлении страницы.

Server + Client

Далее появляется более быстрый интернет и JavaScript:

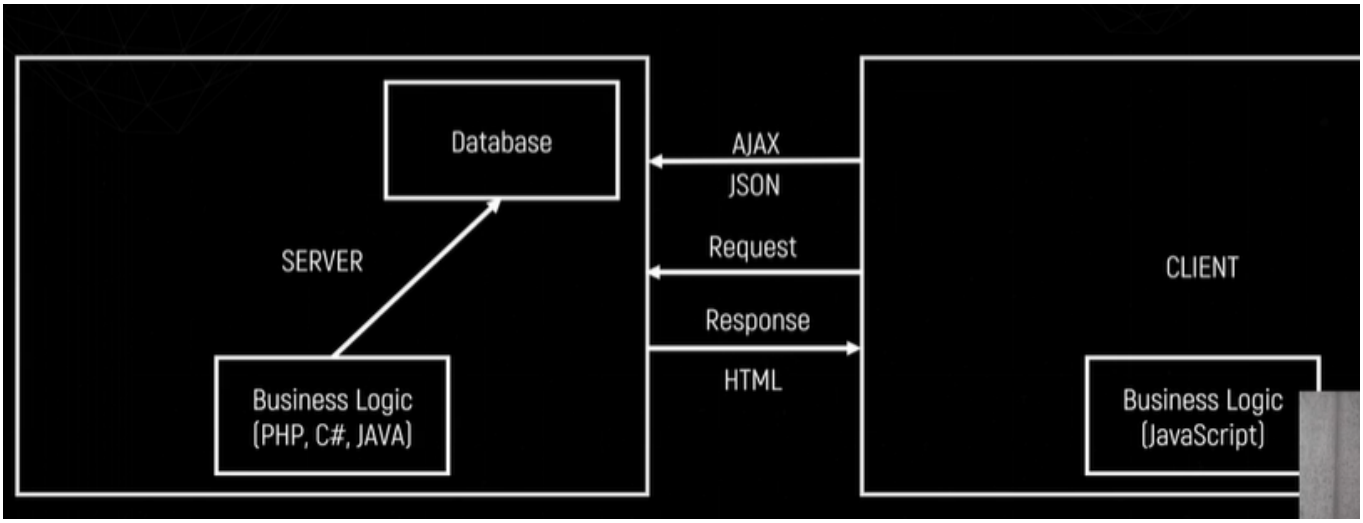


Схема иллюстрирует **переходный этап от MPA к SPA** — важную эру, где **сервер по-прежнему отдает HTML**, но **клиент уже активно подключает JavaScript и бизнес-логику**.
Появляется система AJAX и JSON. Теперь на клиенте можем дорендеривать.
Это примерно 2005–2012 года.

- Появляется **JavaScript как активный участник логики**.
- Начинает использоваться **AJAX (XMLHttpRequest, позже fetch)** для запросов без перезагрузки страницы.
- Полученные данные (обычно **JSON**) используются для динамического обновления UI.
- Но **рендер HTML всё ещё может частично происходить на сервере**.

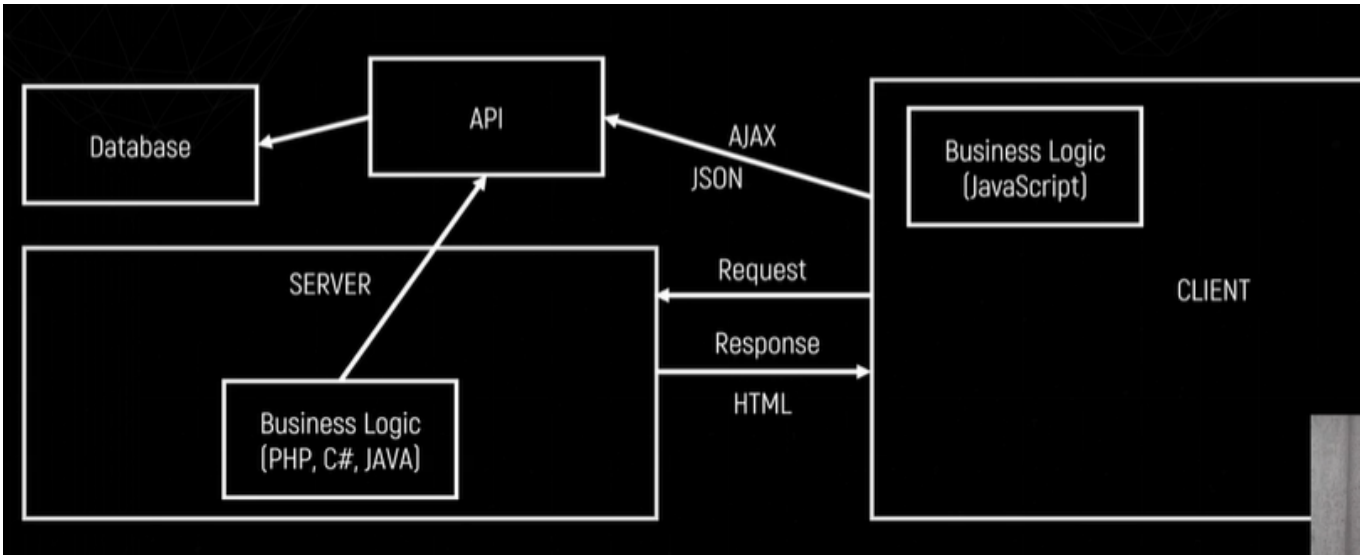
Ключевой недостаток эпохи Server + Client

Проблема дублирования рендера в эпоху Server + Client. Когда сайты начали становиться интерактивными, возникла необходимость обрабатывать данные **и на сервере, и на клиенте**. Вот что из этого вышло:
Контекст:

- Сервер (PHP, Java, C#) всё ещё **рендерит HTML** для начальной загрузки страницы.
- Но клиент (JavaScript) тоже начинает **добавлять интерактивность и подгружать данные через AJAX**.
  В результате один и тот же UI может быть:

- Сначала **отрисован на сервере** (например, список товаров в `index.php`).
- А потом **повторно отрисован на клиенте** (например, при фильтрации товаров через JS).

Server + Client + Api

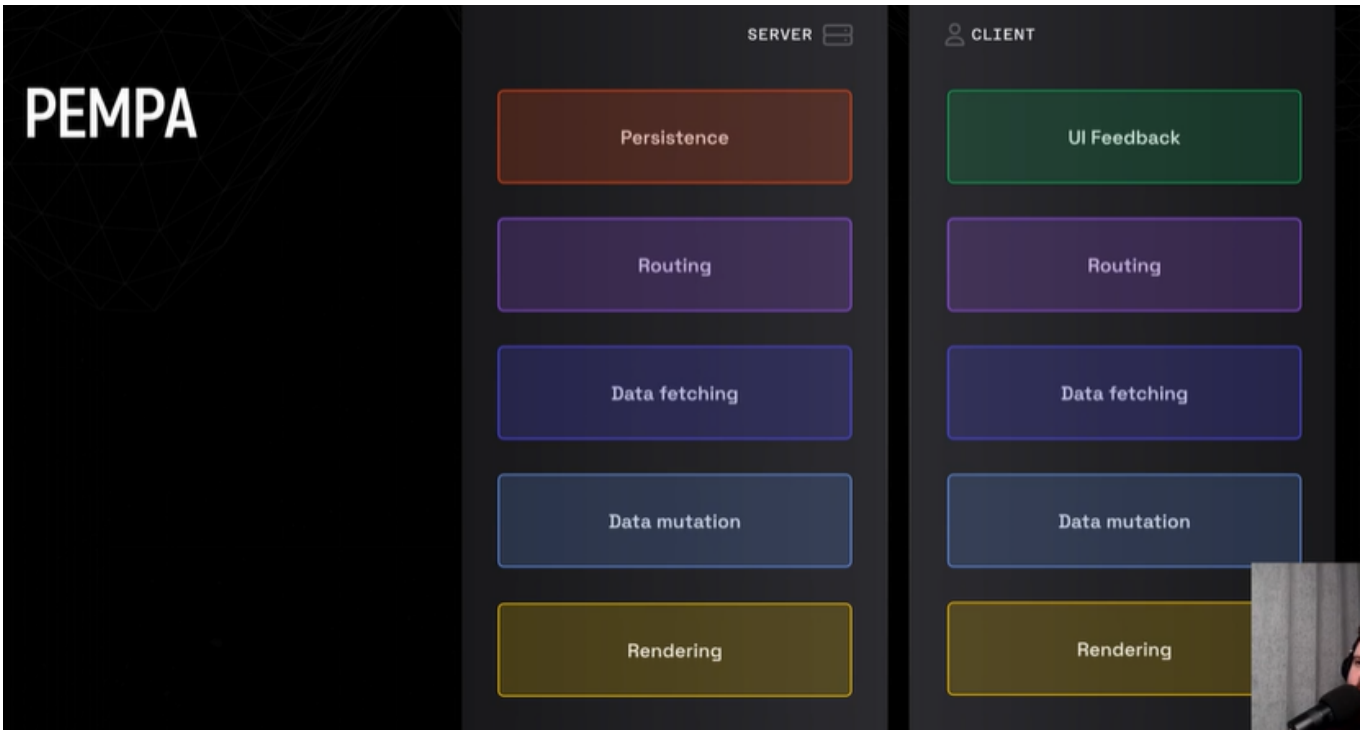


Эта схема показывает **ключевой этап в эволюции веб-приложений**, когда в архитектуре появляется **API как самостоятельный слой**.
Неудобно когда сервер отвечает за рендер и за отдачу данных.
Поэтому вместо того чтобы монолитно хранить базу данных, у нас есть api и сервер туда ходит и клиент тоже. Сервер отвечает теперь только за рендер.

Главная идея:

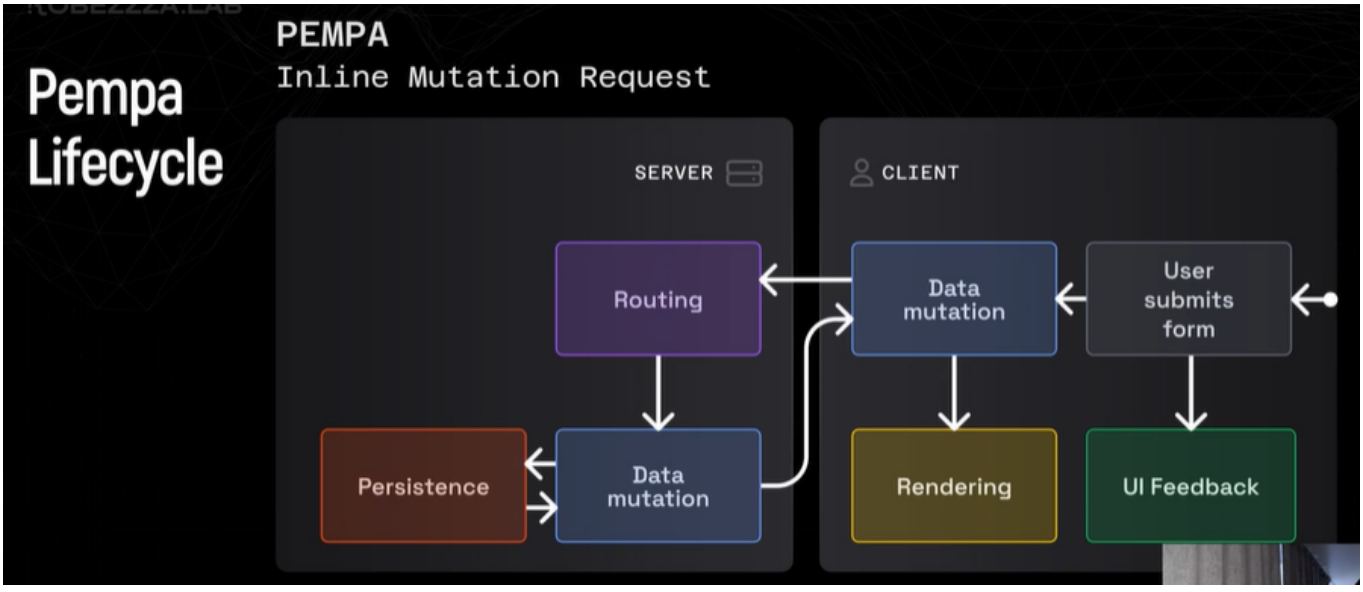
- Разделить **отдачу данных** (API) и **рендеринг** (UI) — чтобы **сервер больше не был монолитом**.

PEMPA (Progressively Enhanced Multi-Page App)



MPA стал так называемым Progressively Enhanced.
PEMPA — это аббревиатура, отражающая **параллельное дублирование ответственности на сервере и на клиенте**.
Клиент умеет все то-же, что и сервер и полностью дублируется логика. Дублирование кода очень серьезное. Клиент также ходит за данными, отвечает за роутинг тоже, как-то данные меняет, ну и занимается рендерингом.

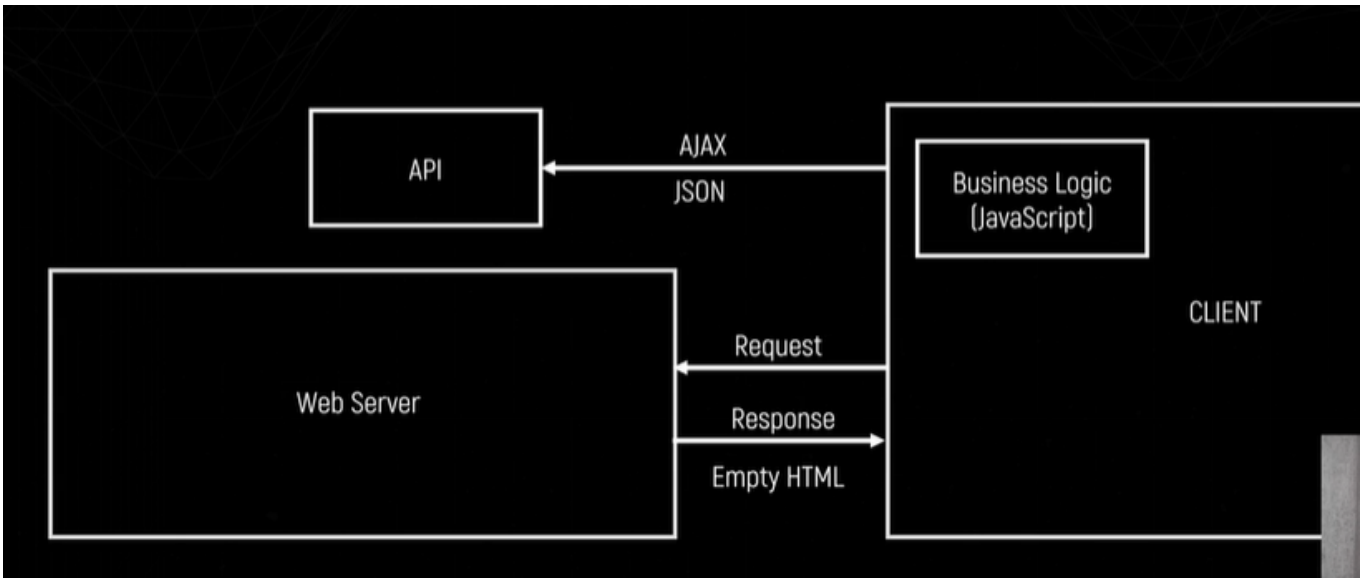
PEMPA Lifecycle (Inline Mutation Request)



Теперь изменения данных на стороне клиента, а клиент при этом ходил на сервер за данными, чтобы их получить и отреагировать на изменения.

CSR - Client side render

Появилось логичное утверждение: было MPA, все было просто, все было на сервере. Потом появилась PEMPA, все стало сложно, потому что логика дублируется на клиенте и на сервере. Почему бы теперь не перенести всю логику на клиент. Так и появилась CSR:

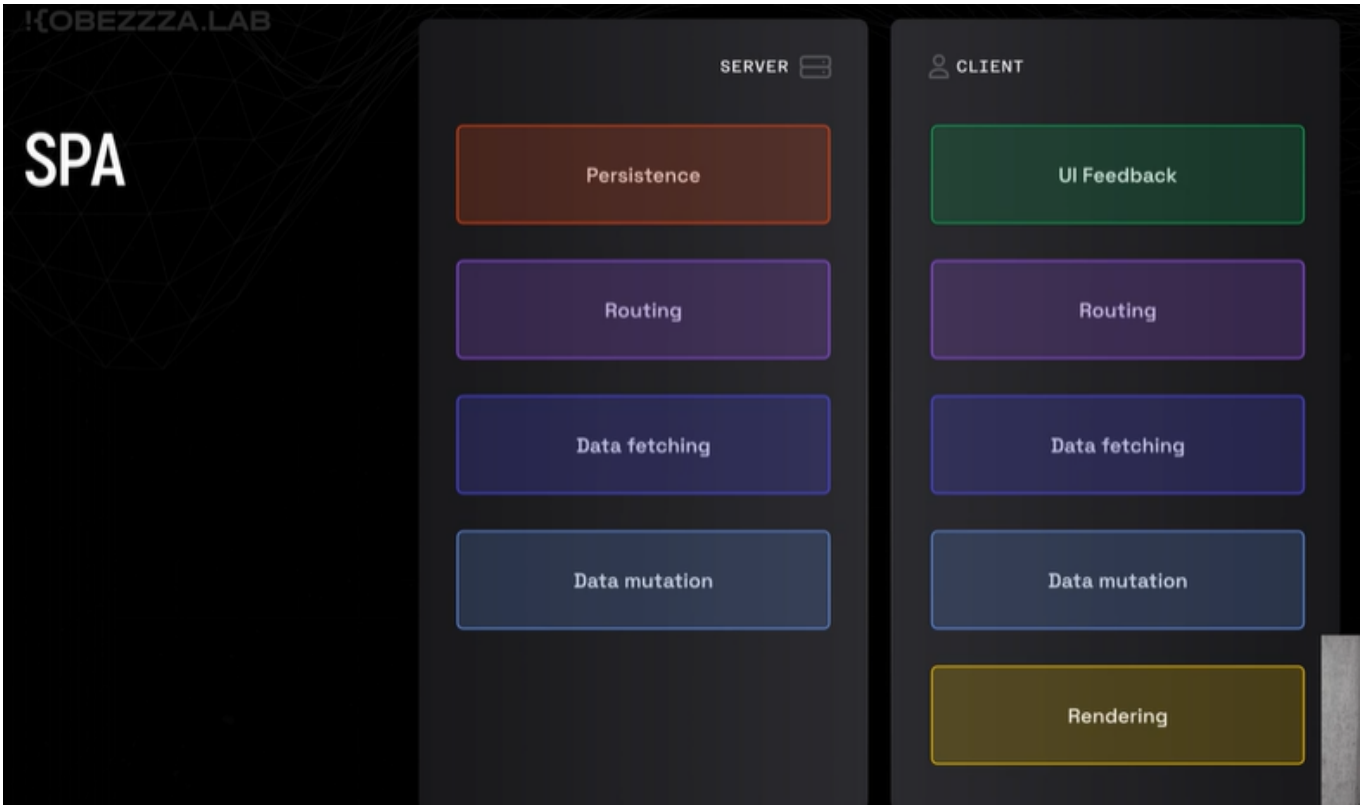


Идея CSR:

- Основной рендеринг происходит **в браузере**, на стороне клиента.
- Сервер не "рисует" страницу, он просто отдает заготовку и скрипты.
- Хорошо работает для SPA (Single Page Applications), но у него минусы:
  - Первая загрузка может быть медленной (нужно сначала получить JS и API).
  - Проблемы с SEO (поисковики не видят контент сразу).
  Чаще всего Web Server = Nginx / nodejs, он просто отдает html. Html была пустая.

SPA

Далее появляется концепция SPA:



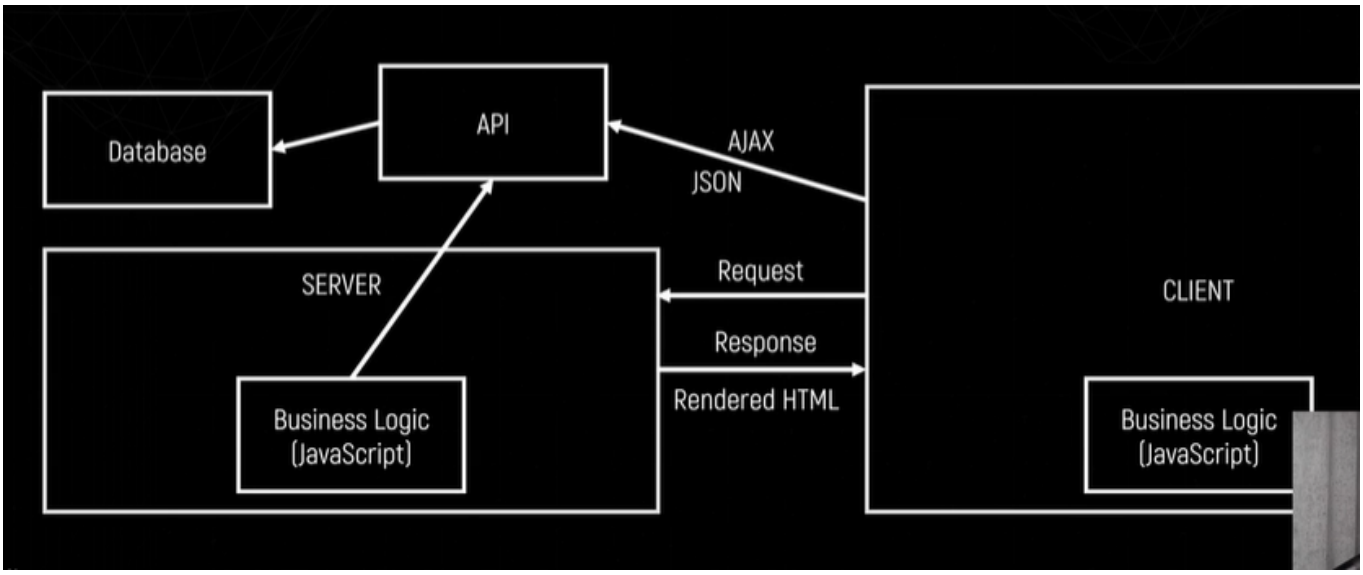
На этой схеме как раз показано **что делает SPA (Single Page Application)** и как распределены ответственности между **сервером** и **клиентом**.
Большую часть работы унесли на клиент.

- На сервере остается роутинг, потому что часть роутов на Web Server'e живет. Например тот же robots.txt
- Работа с данными, потому что еще Api есть.
  В SPA **вся логика рендеринга и навигации перенесена на клиента**, а сервер — это просто "поставщик данных" и обработчик изменений.

Чем отличается SPA от CSR

Понятие	Что это?	Пример	Отношение
SPA	Архитектурный подход	React, Vue, Angular SPA	Это **формат приложения**
CSR	Способ рендеринга	ReactDOM.render в браузере	Это **технический способ рендеринга**

SSR (Server Side Render)



Опять вернулись к серверному рендерингу, но случилась большая революция. Также есть клиент-сервер. Но разница в том, что мы описываем логику на стороне сервера `JavaScript'ом`. Самое главное в том, что бизнес-логика на JS пишется один раз и запускается и на клиенте и на сервере.
SSR — это когда HTML-страница рендерится **на сервере**, а не в браузере.

Как узнать SSR это или CSR?

1. Выключить js - пустой html, значит CSR.
2. Посмотреть какой html возвращается - пустой html, значит CSR.

Как произошел переход от CSR к SSR

С точки зрения разработчика CSR это супер удобная и понятная штука. Понятная концепций, тогда в чем причина создания SSR:

Минусы CSR

- Толстый клиент, очень много js загрузить, распарсить и выполнить.
- Проблема с производительностью, метрики скорости не очень.
- Самый главная проблема SEO, на csr невозможно сделать seo.
- Гугл не умеет парсить.

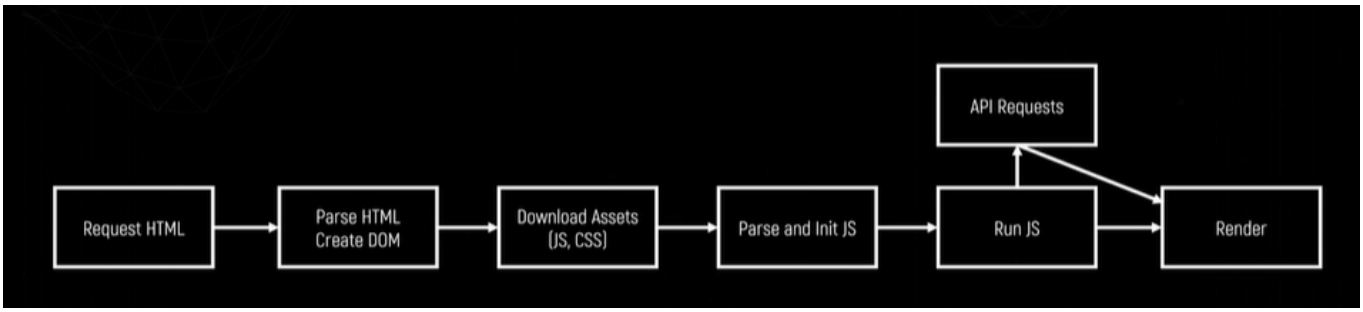
При SSR

- Клиент становится обычным. Первый рендер только происходит на сервере, а дальнейшие рендеры уже на клиенте. Количество JavaScript, которое надо загрузить не особо меняется.
- Возможность делать SEO.

- Проблемы с производительностью никуда не делись.

CSR vs SSR

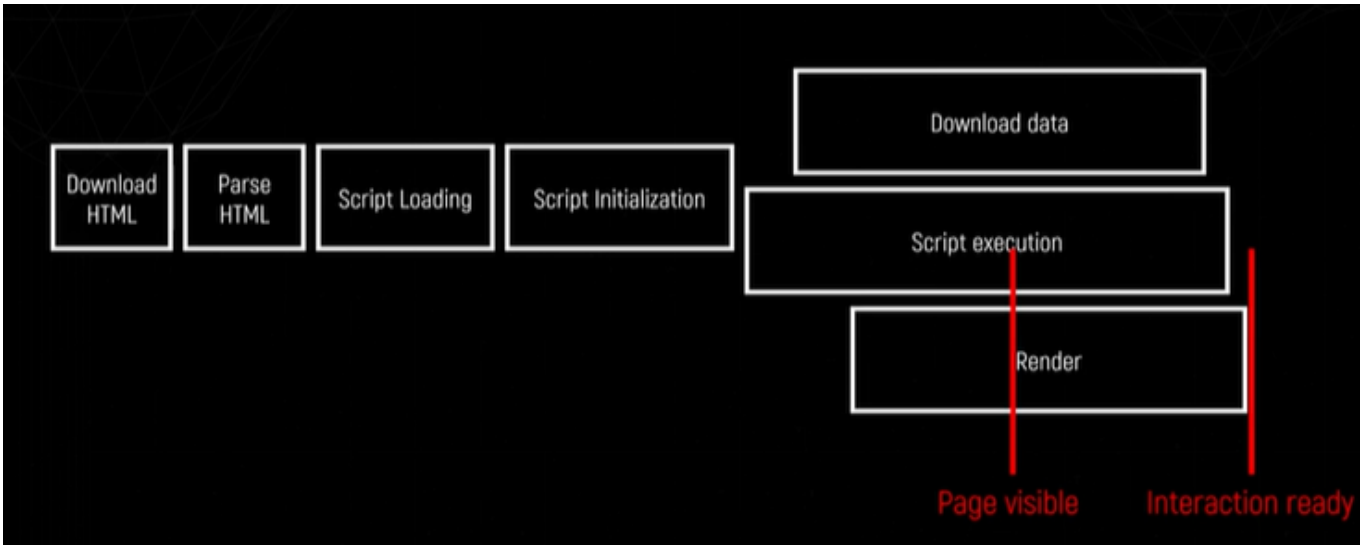
CSR



- Запрашиваем HTML, она пустая.
- Парсим HTML, создаем DOM-дерево.
- Начинаем загружать js-скрипты и css.
- Запарсили, за инициализировали js .
- Выполнили JS.
- После чего совершаем рендер.
- Запросы за данными происходят только после того, как мы начали выполнять JS.

CSR Timeline

Размер квадратика это как долго происходит:



Загружаем html / парсим супер быстрый - контента нет. Загрузка скриптов длительный процесс, так как скриптов много. Инициализация скриптов еще дольше, так как очень много js. Затем идет выполнение - еще дольше, так как нам надо теперь все это выполнить.
И вот здесь когда мы отрендерили начальный html и уже получили верстку, а

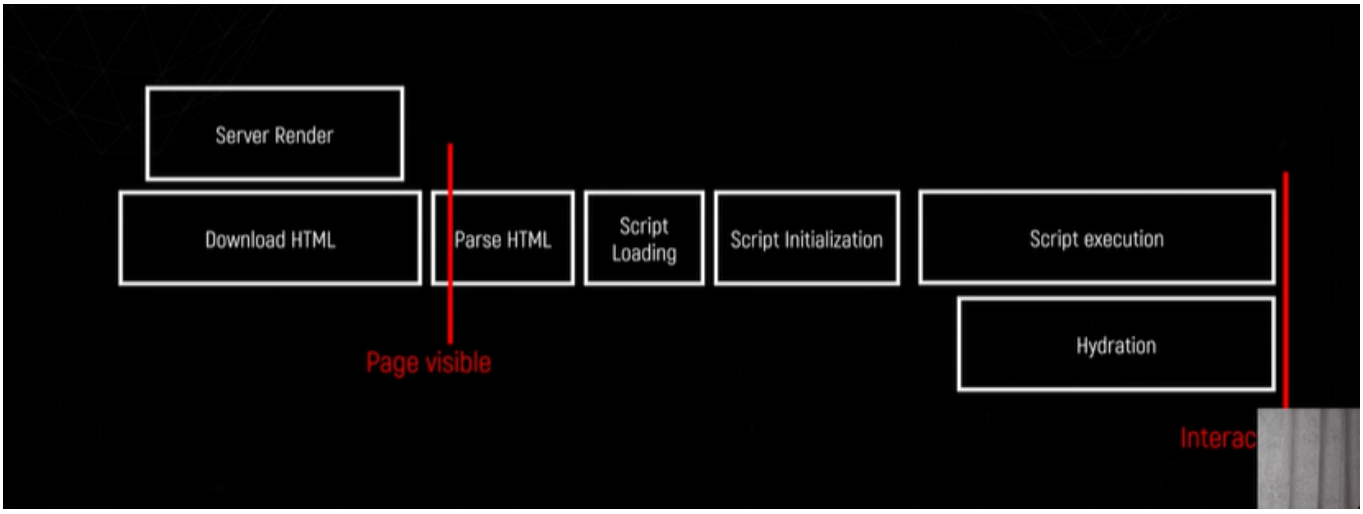
параллельно идет загрузка данных и она же еще триггерит рендеры еще раз, чтобы отобразить контент полностью.
SEO нет и все упирается в количество JS.

SSR

В SSR по другому.

- Запрашиваем HTML, уже сразу отрендеренную.
- Под капотом целая история на стороне сервера: Сервер выполняет JS, совершает запросы и также выполняет рендер. В результате отдается готовая HTML.
- Далее парсится, создается DOM.
- Далее также работа с JS-скриптами
- Гидрация (Hydration)

SSR Timeline

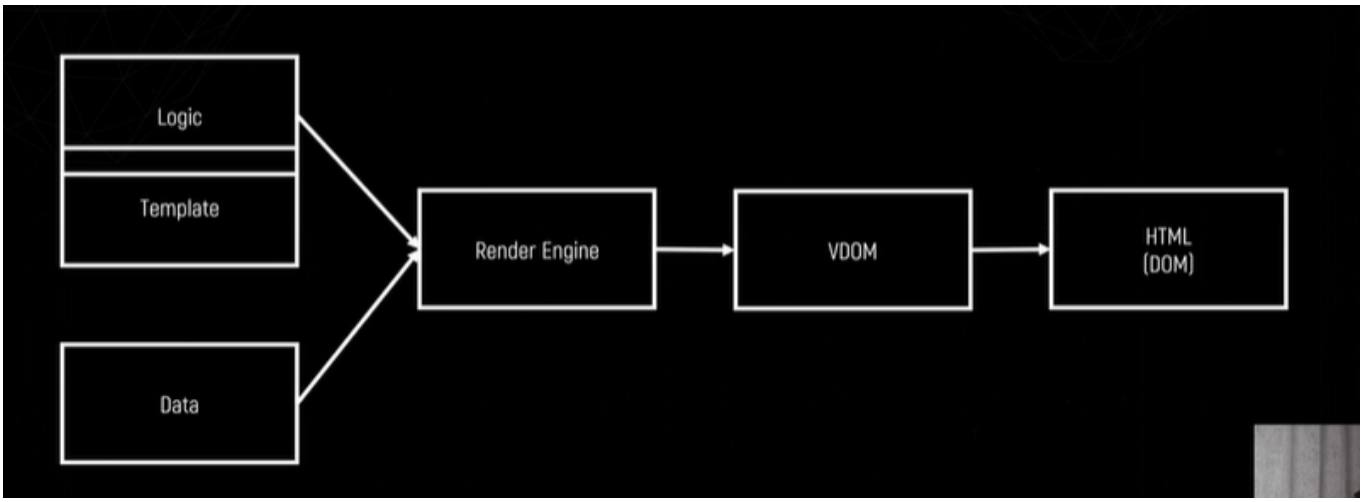


- Загрузка HTML теперь очень долгая. Потому что время которое мы тратили на загрузку данных на клиенте, теперь происходит на сервере. Большая часть загрузки HTML - это ожидание сервера.

- Затем парсим HTML, подольше так как теперь не пустой body. Но важный нюанс в том, что страничка становится видимой уже сейчас. Это метрика `First Content full Paint`.
- Работа со скриптами, здесь SSR позволяет оптимизировать этот этап.
- И далее гидрация.

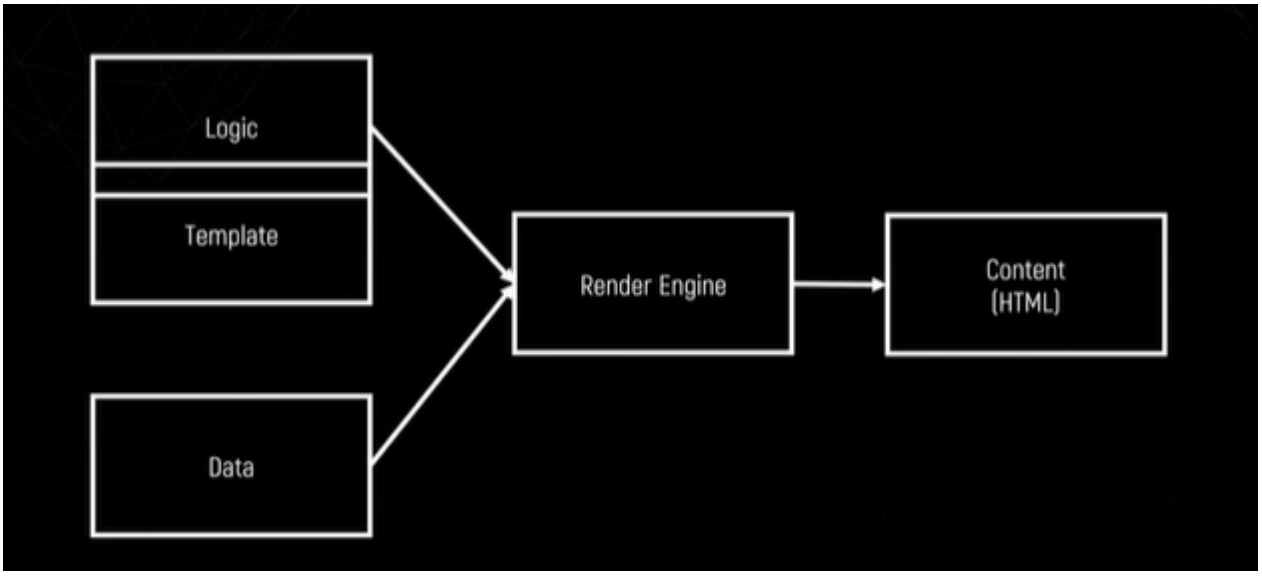
Render CSR.

Рендер состоит из двух больших частей: логика вместе с шаблонами. Обычно пишем шаблон (jsx), берем какие-то данные (Date), пропускаем это все через движок рендера и мы получаем VDOM, далее получаем уже реальный DOM-дерево. По сути VDOM это костыль.

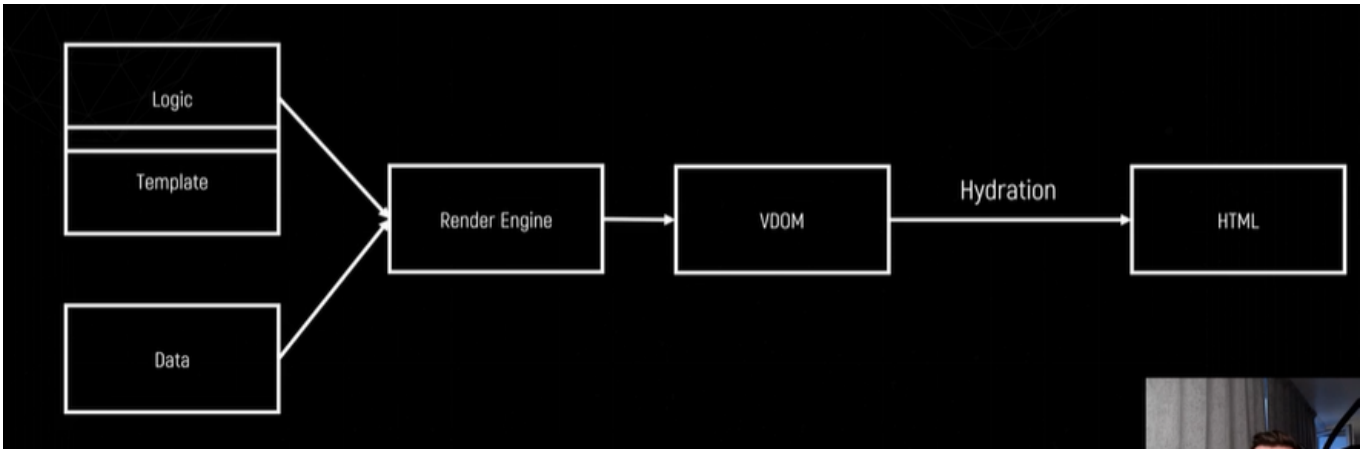


Server Render (SSR).

Тоже есть логика и шаблоны. Есть данные. Также пропускаем черезе движок рендера, но на выходе получаем сразу HTML. Так работали старые темплы и движки (например Handlebars).



Hydration



- Это алгоритм, который также берет логику и шаблоны, берет данные.
- Засовывает в движок рендера, на выходе получает VDOM.
  Гидрация выполняется уже на отрендеренном DOM. То есть на сервере сделали Server Render. Получили HTML, этот HTML уже есть на страничке
- И затем Гидрация по сути синхронизирует свое состояние с тем что есть на клиенте и добавить то, что на сервере сделать невозможно:
  - Сравнивает VDOM с тем HTML, который уже есть - они должны быть полностью одинаковы.
  - Самое важное - **навешивает обработчики ошибок**.
  Это очень спорная штука в концепции SSR.

Переходы после гидрации

После того, как мы загрузили сайт с SSR, провели гидрацию. Затем все взаимодействие со страничкой это обычный CSR.
С точки зрения развития React и Vue, SSR это скорее фича, чем база. Все равно эти фреймворки изначально были созданы для работы в браузере.
Новые фреймворки без сервера уже не умеют работать, они прям full-server'ные.
Новые фреймворки: `HTMX / HOTWIRE (Turbo + Stimulus)`.

Интегрируем SSR. План интеграции

Как интегрировать самописный SSR без фреймворков и библиотек, только React.

1. Создать отдельный entrypoint и сборка для сервера. Код, который собираем для клиента и сервера сильно отличается. Поэтому разные сборки. У React есть отдельный для сервера.
2. Интеграция SSR / реализация BFF.

3. Адаптация кода. Много кода, который рассчитан на работу в браузере, теперь не работает.
4. Оптимизация кода. Защита от утечек памяти самая главная проблема.
5. Этап деплоя

Серверная сборка

Как работает сборка.

- Самое главное, что build target - "node". Она под NodeJS.
- Также выключаем бандлер зависимостей. Disable dependincie bundling.
  В клиентской сборке, все что мы импортируем из node_modules, в итоге попадает в конечный js-бандл. На клиенте никаких node_modules папки нету. Все должно быть на клиенте.
  На сервере это не так, у нас серверное окружение. При импорте они в рантайме честно будут ходить в node_modules.
- Runtime import.
- Ignore css / assets. Не нужны картинки и css.

Context выполнения

Вообще-то на сервере и на клиенте разный контекст выполнения:

- server - request
- client - window

- На клиенте просто window.location. На сервере можем из request узнать адрес странички (для роутинга например)
- Единственное хранилище данных, которое мы можем передавать между клиентом-сервером это куки (в куки важная вещь это информация об авторизации).

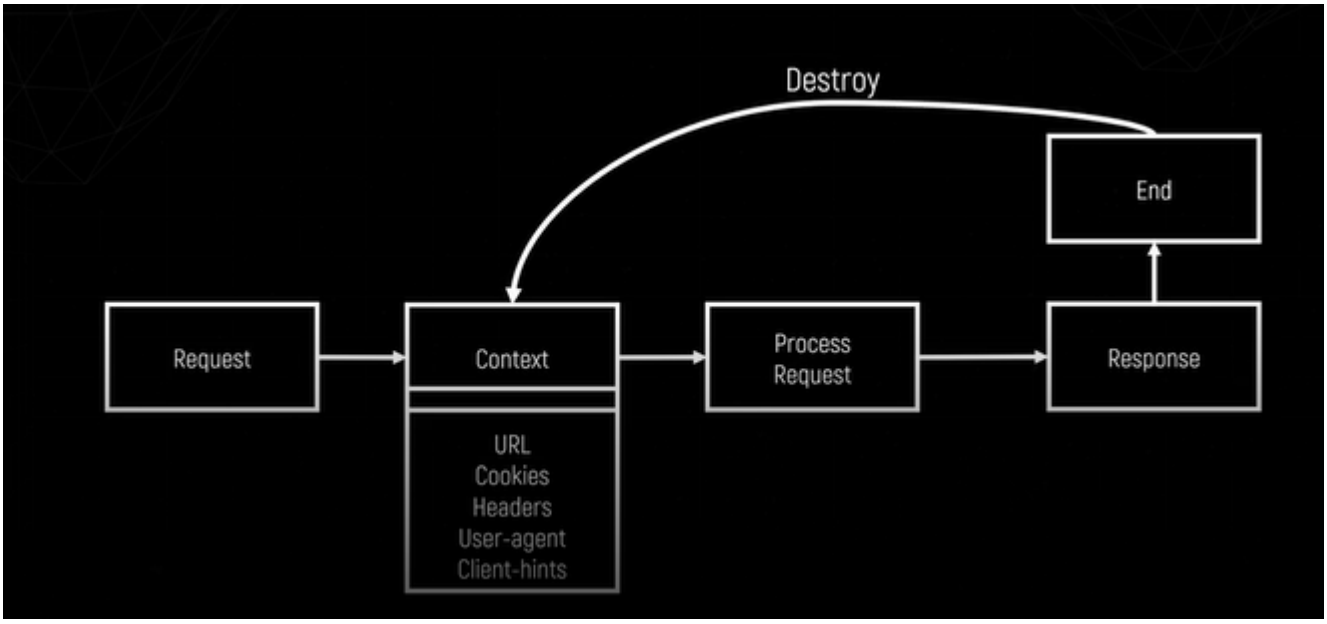
- Дальше другие данные, например ширина странички, высота, ее на сервере нельзя получить.

При переезде с CSR на SSR, всегда нужно отвечать на вопрос: В каком контексте будет выполняться код? Код должен теперь работать в Nodejs. Так как появляется серверный-контекст.

Утечка памяти при CSR

На клиенте контекст создается один раз для работы приложения. Он инициализируется один раз под конкретного пользователя. Например в window мы можем положить какой-нибудь объект userInfo, объявить какой-нибудь singleton и оттуда экспортить инфу и использовать ее в рендере. И самое главное, при закрытии страницы, весь контекст уничтожается. Чтобы реально заметить утечку памяти, нужно держать вкладку открытой несколько часов на слабом компе.

Утечка памяти при SSR

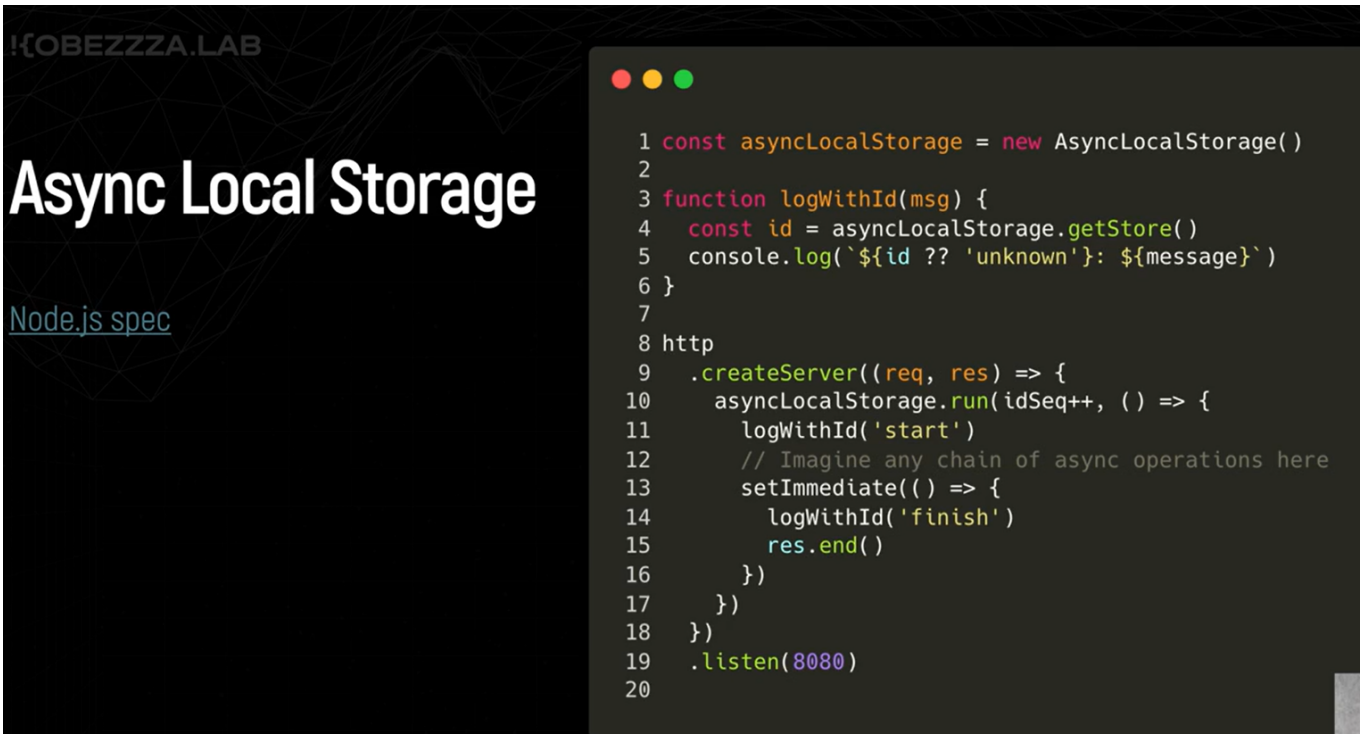


Однако когда мы начинаем рендерить на сервере, появляется много нюансов:

- Код также инициализируется один раз.
- Но потом этот код начинает выполняться 1000 и млн раз. И если хоть где-то чуть-чуть есть утечка памяти, то это будет очень быстро видно.
- Очень важно после окончания запроса, важно уничтожить контекст!!!
  На каждый request нужно заново создавать контекст!!! И этот код выполнять 1000 и 1000 раз.
- Внутри useEffect безопасно обращаться к API браузера.

Внутри фреймворков (Nextjs) сервер очень плотно обвязан вместе со всем рендером. Там буквально собирается entrypoint-server и внутри уже собирается HTML с нуля прям.

Async local storage:



Это по сути аналог React Context. Когда берем и оборачиваем приложение в контекст и можем в любой момент обратиться к нему и достать данные.
Нам также ничего по дереву пропсов прокидывать не нужно. Прям как контекст в реакте.
Полезно при написании middleware на сервере.
В стор валидно складывать какие-то данные из request. Nodejs сама чистит этот стор.
Nextjs использует его под капотом.

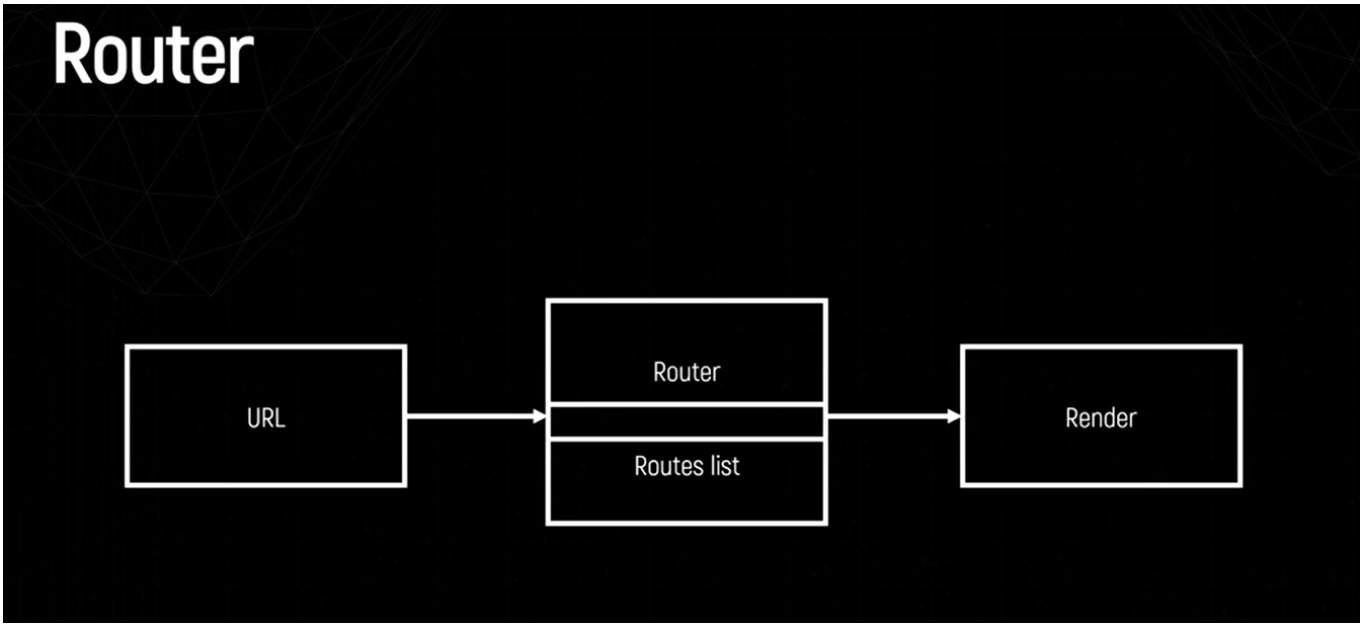
Механизмы SSR

Адаптируем что раньше работало на клиенте под сервер:

- Router - переносим роутер, чтобы он работал не только на клиенте, но и на сервере.
- API requests - сервер тоже должен ходить за данными.
- Context & State.
- SEO.

Router

Что такое роутер?

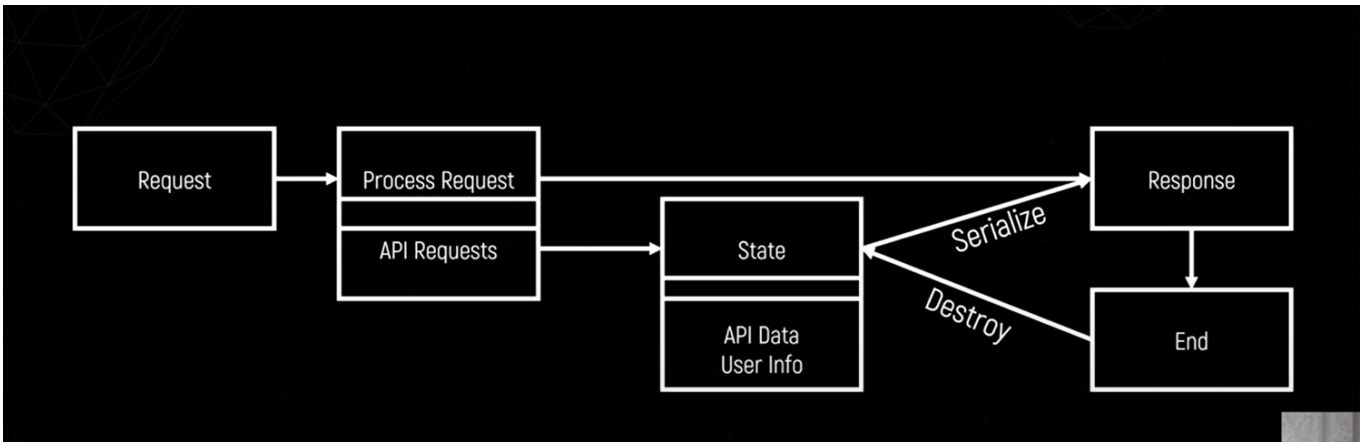


Примерный его алгоритм такой:

- С помощью роутера мы по сути маппим адрес странички на то, как эта страничка должна выглядеть, что за компонент надо зарендерить.
- Роутер берет на вход адрес странички. Также у нас есть лист-роутов. В Nextjs это fs-routing.
- Сопоставляем и рендерим нужный компонент
- Вернули отрендеренный компонент
  Очень часто данные привязаны к роутингу. Данные которые нужны на разных страницах, они отличаются. То есть мы уже должны рендерить с данными.

State

В итоге мы смогли завести данные на сервере и отрендерить контент. Однако у нас может не работать гирация. Потому что на клиенте данные получим только в useEffect. То есть у нас приходит HTML с данными с сервера, алгоритм видит, что HTML с сервера и DOM разные и выбрасывает ошибку.



- Есть запрос на сервере.
- Мы его процессим (делаем запросы в api, делаем рендер).
- Далее мы складываем все эти данные в стейт. В стейте лежат ответы апишек, информации о юзере и так далее. Прям context.data это и есть стейт.
- И вот этот state, который мы использовали, чтобы отрендерить верстку, мы затем сериализуем и отправляем на клиент в виде строки. Чтобы клиент с точно такими же данными отрендерил нашу верстку.
- Далее запрос заканчиваем и обязательно уничтожаем стейт.

Важные замечания:

- Честный SSR - вся разметка вся верстка возвращается именно с сервера.
- State складываем в noframes.
- Можно использовать какую-нибудь синглтонину между несколькими запросами (например конфиг). Но нужно быть аккуратным, чтобы этот синглтон не стал зависимым от пользовательских данных.
- Важно не хранить данные вне глобального стейта или вне хука useState. Чревато утечкой.

SEO

- Статус ответа. Честно возвращать 404, редирект итд.
- Мета тэги (title, description, OG, canonical).
- Контент страницы (h1)

SEO зависит от страницы, в роутере или рядом описывать надо. Типа getSeo

Также нужно делать статус-код. Проблема в том, что мы должны во время рендера получить данные и понять то что эта страничку рендерить не надо, ее не существует.

Рендерим страничку и вдруг оказывается ее не существует. при серверном рендере. рендер прирываем и возвращаем 404. В таком случае принято выбрасывать исключение ахаха
В try catch оборачиваем и в catch смотрим что за ошибку мы получили. если ли свойство статус или редирект. А иначе честно возвращаем 500.

Архитектурно можно выкинуть исключение в getSeo.

Полный рецепт

- две сборки клиент и сервер
- bff nodejs

- понимание где клиентский, где серверный код
- каждый рендер создается новый контекст.
- доп серверные механизмы по типу передачу стейта

- В продолжение вопроса о дублировании useEffect: получается, если у нас SPA, то SSR отработает только на первой загруженной странице? А дальнейшая навигация в SPA будет использовать CSR? - да
- то есть гидрация тоже строит VDOM чтоб сравнить с DOM после рендера на сервере? Но фактически не перерисовывает в браузере? - да, но если контент другой, дом может быть поменяться.
  -можете уточнить еще раз, SSR только при первой загрузке работает? при переходе между роутами потом это уже работает как CSR ? - да
- т.е все эти танцы для того, чтобы работали "Клики" на страницах которые отрисованы на сервере? - да, ради всех событий клиента.

Трамвай - у некста беда с микрофронтами. а в Т-банк нужен module federation. Трамвай умеет подключать микрофронты на сервере и делать рендеринг микрофнтронтов на сервере.
Была проблема с тем что домен т-банка может любой момент заблочить, у некста в рантайме нельхя изменить домен. А в трамвай в рантайме можно менять.

Next на webpack, а Трамвай на rspack планируется переводить.

---

open auth
иногда куки 3 стороны. это только клиентская, упираемся в то что может случится рендер (мы не знаем авторизован ли он или нет) и мы не знаем что ему рендерить.
Если роут закрыт за авторизацией, то ей вообще не нужен ssr. Для seo она бесполезна.