Микрофронтенды в рантайме для сложных приложений

Содержание

1. Общее описание каналов коммуникаций между микрофронтендами

1.1. Вступление

Понимание того как работает приложение позволяет лучше формировать требования для функционала, который будет хорошо и однородно вписываться в систему. Частью этого понимания является понимание ролей ключевых подсистем и каналов обмена информацией между подсистемами. В настоящее время для создания клиентских приложений используются разные подходы. Ниже будет описано приложение использующее микрофронтенды интегрирующиеся на клиенте в рантайме как основной архитектурный подход, хотя многие суждения будут справедливы и для других подходов связанных с микрофронтендами вне зависимости от способа интеграции. В частности ниже мы постараемся объяснить какие ключевые подсистемы могут находиться в современном большом веб приложении, какие между ними существуют каналы связи, как они могут участвовать в кешировании информации, опишем структуру нескольких типов микрофронтендов и их назначение, разберем устройство основных систем в коиентском энтерпрайз приложения. Следует оговорится, что данный архитектурный подход хорошо подходит для больших клиентских приложений, над которыми работает несколько команд, при чем больше независимых команд, тем лучше он окупается. Основное преимущество микрофронтендов — независимая от других команд доставка фич конечному пользователю, при правильно настроенных процессах и CICD мы можем вообще отказаться от понятия релиза и поставлять готовые фичи клиенту незамедлительно, что даст нам максимально быструю обратную связь и поможет скорректировать направление дальнейшего развития.

1.2. Типы микрофронтендов

Начнем с того, что для интеграции в рантайме нам нужно иметь базовое приложение, в котором данная интеграция и начнется, назовем этот микрофронтенд **host-microfrontend.** Часто в приложениях существует часто используемая часть на большинстве страниц по типу навигационной панели, футера, хедера и пр. Данную часть будем называть **border-microfrontend.** Название происходит из того факта, что данные панели организуют некую кайму. Иногда можно не выделять дополнительный border-microfrontend, а заимплементировать его сразу в хост приложении, но так как их функциональность принципиально не связана и для разных страниц мы вполне вероятно захотим использовать в дальнейшем разные border-microfrontends (или, возможно, появится задача временно изменить стили и содержание нашего border-microfrontend к Новому Году или по случаю другого события типа ребрендинга), а отказ от выделения дополнительного одного микрофронтенда принесет в лучшем случае очень слабый выйгрыш по премени локально для одной команды, рекомендуем выносить его в отдельный микрофронтенд. Далее, микрофронтенды можно классифицировать с помощью разных критериев в зависимости от необходимого контекста, но нас будет в первую очередь интересовать является ли микрофронтенд отдельной страницей (лэйаутом) или блоком, который встраивается в страницу. Микрофронтенд-страницу будем называть **page-microfrontend** (иногда называют Layout, но так как мы будем подразумевать, что page-micrfrontend может использоваться без использования дополнительных микрофронтендов будет уместнее называть все таки page). В page-microfrontend могут подключаться и настраиваться общие инструменты, такие как лоадеры, обработчики ошибок, каналы связей и прочее, поставляемое в микрофронтенд с участием хост-приложения. А микрофронтенд-блок назовем **fragment-microfrontend,** по сути это специализируемый на конкретной задаче переиспользуемый в разных страницах виджет. Такой виджет сам по себе может быть довольно сложным и подключать в себе дополнительные fragment-microfrontends. Итак, архитектурно выходит такая картина:

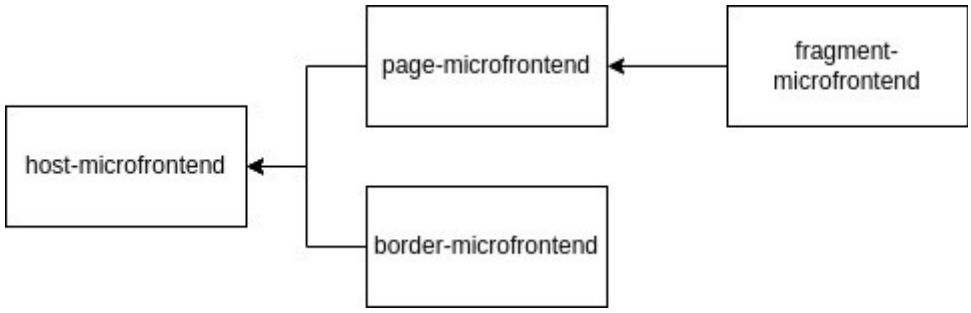

Рис. 1 Типы микрофронтендов.

1.3. Что с чем должно общаться?

Начнем с того, что определим в чем суть коммуникаций и какие потоки данных существуют на клиентском приложении. Данный материал будет справедливым не для всех проектов, а только для достаточно больших приложений над которым потенциально может работать (или будет работать в будущем) несколько независимых команд. Т.е. не все описанные каналы стоит использовать на любом проекте. Их нужно тщательно подбирать в зависимости от конкретной цели. Суть современных подходов к архитектуре в применении к текущему вопросу можно описать следующими тезисами

- нужно позволить командам разрабатывать независимо, так как интеграционные вопросы отнимают много сил в больших проектах
- нужно инкапсулировать функционал таким образом, чтоб функционал мог не только разрабатываться независимо от разных команд, но и независимо поставляться клиенту.

Оба принципа направлены в целом на минимизацию расходов на интеграцию между командами. Это ключевые принципы для любой сложной системы, в дальнейшем мы на них буду опираться.

Итак, какое у нас есть решение удовлетворяющее этим принципам? Правильно, микрофронтенды! Концептуальная идея такая: в приложении выделяются независимые в разработке, потенциально переиспользуемые блоки, каждый из которых может обладать своим состоянием и предоставлять некий функционал. Данные блоки могут поставляться разными способами, например популярные решения:

- HTML, получаемый из разных источников и встраиваемый в хост HTML в рантайме
- iframes
- независимые javascript файлы из разных источников, модифицирующие DOM в рантайме
- web components
- module federation из webpack
- Server-Side Rendering (SSR) с чем то наподобие Server-Side Includes (SSI)

У каждого из способа доставки есть свои преимущества и недостатки с точки зрения скорости доставки первого и последнего байта, простоты реализации, надежности и инкапсулированности и прочих критериев. Кроме того, инфраструктурные ограничения могут сильно влиять на выбор механизма интеграции (например необходимо иметь возможность использовать на сервере SSI для реализации подхода с SSI, а для module federation нужно будет использовать вебпак).

Часто эти блоки могут быть визуальными, но не всегда; например, элемент формы или контейнера может не отображаться на экране, но он несет некий функционал – способность собирать и как-то обрабатывать данные для формы и, допустим, чисто семантическое выделение группы элементов для контейнера.

Но хоть мы и стараемся сделать компоненты максимально независимыми, в сложном приложении они часто должны уметь как-то делиться информацией друг с другом и как-то вместе реагировать на изменение внешних условий (события в системе). Компоненты могут общаться между собой через разные каналы связи, каждый из которых обладает своим достоинствами и недостатками, о чем речь и пойдет ниже.

2. Каналы связи между микрофронтендами.

2.1. Классификация каналов связи

По типу используемых в коммуникациях частей системы можно выделить три группы каналов:

1. Каналы использующие браузерное API (FRONT BROWSER API CHANNELS).
2. Каналы использующие для общения инструменты в коде, по сути высокоуровневое общение через оперативку которое и настраивают в своей работе инженеры ПО (FRONT IN MEMORY CHANNELS).
3. Каналы задействующие бэкенд часть системы (BACK CHANNELS).

Группы предоставлены на рис.1 ниже.

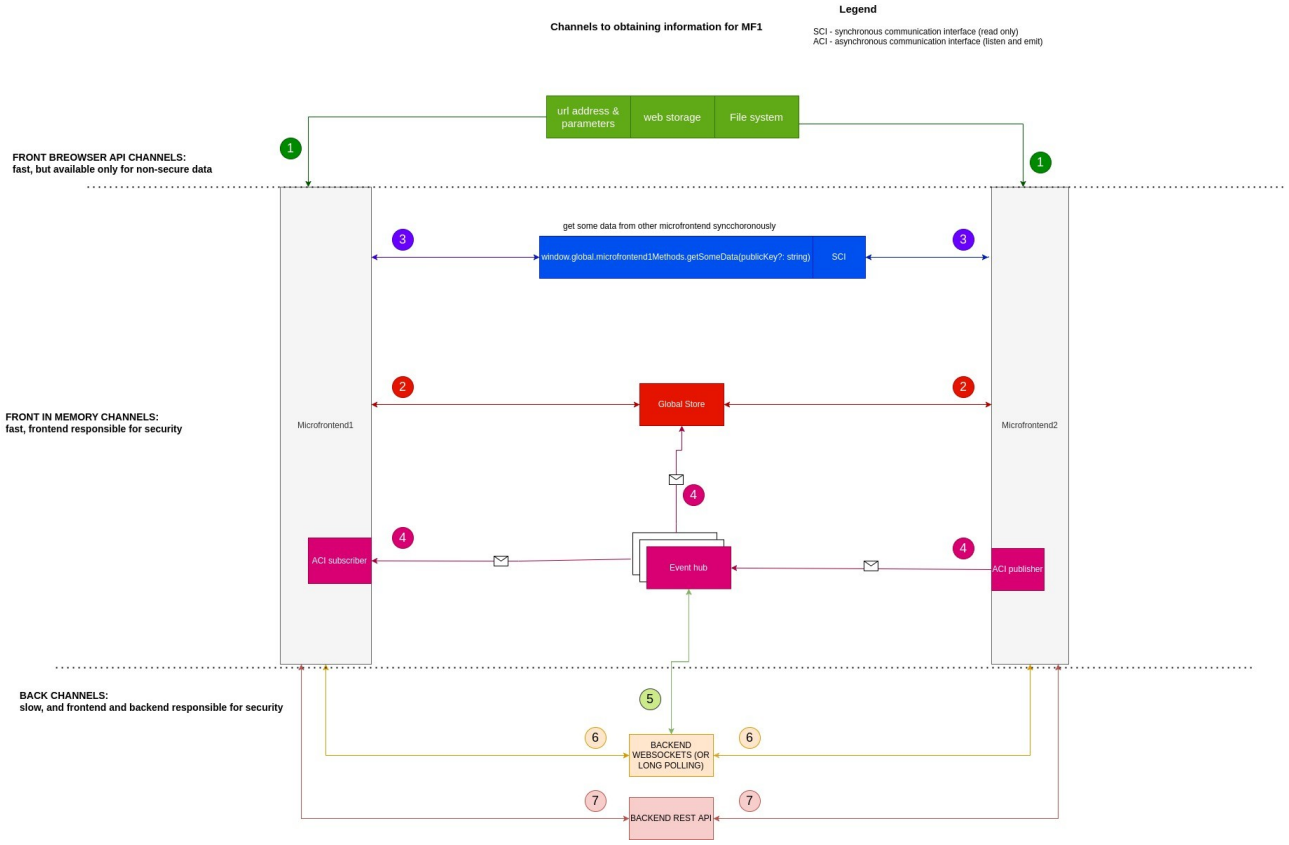

Рис. 2 Классификация каналов связи.

Каждый прямоугольник на этой схеме обозначает некое виртуально место в системе, в котором может храниться информация и таким образом выступать в роли кеша. Стрелки одинаковых цветов с цифрами обозначают группы каналов связи обладающие определенными свойствами, на которых остановимся подробнее ниже с обсуждением того какие элементы для какого типа кеширования информации используются. Вот перечень групп каналов:

1. Каналы использующие browser api.
2. Каналы использующие global storage.

3. Каналы использующие синхронные коммуникационные интерфейсы.
4. Каналы исползующие асинхронные коммуникационные интерфейсы.
5. Каналы использующие пуш уведомления от бэкенда для уведомления всей системы
6. Каналы использующие пуш уведомления от бэкенда для уведомления конкретного микрофронтенда
7. Каналы использующие для коммуникации api на бэкенде

2.2. Описание групп каналов связи

FRONT BROWSER API CHANNELS

2.2.1 Группа каналов использующих браузерно api

В данную группу входят каналы использующие для передачи информации

- url адреса и параметры,
- sessionStorage
- localStorage
- indexDb
- file system
- biometry API
- etc…

Это отличный класс каналов обмена данными между микрофронтендами, который проверен временем, для каждого из конкретного канала есть место применения, но к сожалению часто эти каналы достаточно узко специализированы и обладают ограничениями в использовании, о которых необходимо знать при проектировании приложения. Каждый из этих каналов позволяет кешировать определенную информацию для последующего использования. Это обширный класс каналов, поэтому пока на нем останавливаться не буду.

Пример: пусть при некоторых событиях внутри микрофронтендов нам необходимо знать некую информацию о предыдущем событии, которое могло протекать в другом микрофронтенде (например время выполнения). Давайте при таких событиях сохранять необходимую информации в webStorage, тогда каждый раз при протекании события мы сможем проверять наличие такой информации в хранилище и использовать ее. Проще простого!

FRONT IN MEMORY CHANNELS

Эта группа каналов – именно то, чем занимаются инженеры на frontend. Именно её использование позволяет в большей мере добиться нужной архитектуры и функциональности при создании клиентского веб приложения.

2.2.2. Каналы использующие global storage

Этот класс каналов получил очень широкое использование после популяризации архитектуры flux и redux. В самом деле достаточно просто (особенно с учетом выработанных на сегодня подходов и инструментов) организовать одно общее хранилище для всего приложения. Часто для этого используется некий state manager в котором храниться вся информация которая была запрошена. К тому же данное хранилище отлично может выступать в роли хранилища для кеша, если мы не планируем работать с предыдущими состояниями. Данный канал позволяет достаточно легко шарить информацию между микрофронтендами, при чем для не слишком сложной логики эта информация очень просто синхронизируется (хранилище то одно, значит и синхронизировать нечего, главное уведомлять необходимые части системы об изменениях). К сожалению было выявлено, что несмотря на достаточно высокую эффективность в средних по размеру приложениях данный класс каналов обладает большим минусом -- не позволяет следовать тем самым двум тезисам о которых мы упоминали. Кроме этого если global store становится слишком большим, то его становится трудно контроллировать. Как аналогия для бэкенда – global store – это общая для всего база данных; если приложение не очень большое, то это довольно удобно, но если у нас

много микросервисов, то общая база становится слабым местом за счет проблем с масштабированием и обслуживанием. Global store хорошо подходит для кеширования и шаринга изолированных кусков информации, но нужно все время стараться максимально отдалять себя от искушения расширять его.
Пример: создаем стор в хост приложении и начинаем прокидовать сам стор или его методы напрямую в другие микрофронтенды. Микрофронтенды таким образом могут подписаться на изменения в этом сторе и реагировать на них. При изменении данных одним микрофронтендом все остальные сразу получат доступ к ним и смогут отреагировать.

2.2.3. Каналы использующие синхронные коммуникационные интерфейсы

Две команды разрабатывают два микрофронтенда. У одного из них возникает потребность в получении информации от другого. Как быть? Проще некуда – давайте запросим ее в коде напрямую. Для React разработчика в простейшем случае это значит прокинуть необходимый пропс от родительского компонента к дочернему. А что делать если наши микрофронтенды не подчиняются такой иерархии? Можно предложить использовать что-то наподобие react контекста. Но это будет не правильным, так же как и использовать поднятие свойства состояния или общий стейт – ведь в таком случае состояние становится общим для всей иерархии, а должно принадлежать конкретному микрофронтенду. Вместо этого мы можем вынести наружу функции при вызове которых мы сможем синхронно получать информацию о состоянии микрофронтенда. По аналогии с бэкендом – это api микросервиса. Данный канал хорошо подходит для получения данных при инициализации или принудительной синхронизации со стороны вызывающего микрофронтенда. Так же через него мы можем настроить real-time синхронизация используя что-то наподобие паттерна pub/sub, но это зачастую не лучший вариант, точно так же как не стоит высылать сообщения о некоторых событиях произошедших в одном микросервисе напрямую другому микросервису. Еще один значительный минус данного канала – не способность функционировать, если наш микрофронтенд не примонтирован. Т. е. либо этот канал нужно использовать как вспомогательный (например чтоб не дублировать одинаковый. запросы из разных микрофронтендов), либо нужно принять жесткую связь между коммуницирующими микрофронтендами (чего бы нам не хотелось из-за тех самых двух тезисов).
Пример: простейший пример — общение микрофронтендов находящихся в отношениях родитель-дочка — свойства прокинутые напрямую и есть такой интерфейс. Более сложный случай, когда микрофронтенды находятся в разных ветках дерева иерархии. Тогда мы можем создать в одном из микрофронтендов методы доступа к его данным и прокинуть их через один из других каналов (например через global window, или через global store). Другой микрофронтенд получит эти методы (коллбэки) через другой канал (не синхронный коммуникационный интерфейс) и сможет в последствии синхронно получать данные из первого микрофронтенда. В идеале это можно оформить в отдельную библиотеку которую будет поддерживать команда первого микрофронтенда, для простоты подключения другими командами.

2.2.4. Каналы использующие асинхронные коммуникационные интерфейсы

Если синхронные каналы подходят для получения данных, то для уведомления всей системы о изменениях хорошо подходят асинхронные каналы. При происхождении некоторого события в системе необходимо уведомить всех кто подписан на данный класс событий о случившемся. Обычно это должно быть событие бизнес уровня, ошибкой будет создание событий уровня микрофронтенда, так как обычно мы не хотим знать что “событие А произошло в микросервисе Б”, нам достаточно знать, что “событие А произошло”. Это позволяет проектировать гораздо более гибкие системы которые реагируют на бизнес-изменения в системе в целом, а не на изменения в конкретном микрофронтенде. Хотя конечно, в теле события можно указать и кто его был изначальным источником. Исходя из этих соображений мы создаем некий event hab’ы в каждый из которых можно эмитить

некоторые группы событий (обычно относящиеся к одной доменной области) на которые могут подписаться любые желающие микрофронтенды и получать и обрабатывать полученные от этого хаба события. Для бэкенда это аналог Event Driven Architecture. Данный подход позволяет по настоящему независимо обрабатывать изменения в системе для отдельных микрофронтендов, а так же проводить тестирование микрофронтенда без его интеграции в систему. К минусам можно причислить невозможность (сильную нежелательность) получить при желании необходимые данные в конкретный момент времени, но ведь мы помним, что для этого есть синхронные интерфейсы. В качестве хаба хорошо подходит использование **service worker** или их аналоги, который может инкапсулировать в себе отдельную предобработку событий, их логгирование и прочие вспомогательные операции. Сам event hub обычно не должен хранить никакую информацию, кроме очереди отправленных ему событий. Кроме воркеров для асинхронного общения просто отлично подходит **Broadcastchannel API**, поддерживаемое всеми современными браузерами, особенно уместно использовать данное API если обработка событий отдельно от микрофронтендов не требуется.

Пример: у нас несколько микрофронтендов в приложении которые хотят автоматически реагировать на некое изменение в системе. Создаем для этого изменения event hub, микрофронтенды подписываем на него (теперь они будут получать уведомления, если event hub испустит определенное событие). При определенном изменении в системе (не важно кто его запускает) мы отправляем в хаб событие уведомляющее об этом изменении, а хаб уже уведомляет своих подписчиков-микрофронтендов.

BACK CHANNELS

2.2.5. Каналы использующие пуш уведомления от бэкенда для уведомления всей системы

Мы знаем как поддерживать синхронизацию данных в микрофронтендах с помощью асинхронных интерфейсов на клиенте,но как быть, если событие произошло на backend’е? Ответ прост – доставим это событие в хаб, а он уже сообщит всем подписчикам о нем. Как это сделать? Старые добрые AJAX, webSockets и long polling которые подключены внутри хаба (в нашем случае service worker) отлично справляются с задачей. Backend – это тоже часть системы, с точки зрения событий не отличимая от описанных выше клиентских асинхронных интерфейсов. Этот подход просто отлично интегрируется в клиентсткую систему систему в которую встроены механизмы работы с событиями. При этом, так как мы всегда максимально быстро получаем актуальные данные, их кеширование может даже не требоваться (если мы отказываемся от поддержики работоспособности в оффлайне).

Пример: в приложении есть несколько микрофронтендов на одной странице, каждый из которых использует общую информацию. Можно подписать event hub на получение такой информации, а микрофронтенды подписать на некое событие которое при этом будет испускать event hub. Теперь при изменениях в системе (на бэке) наш фронтенд будет уведомлен автоматически и все микрофронтенды в нем подписанные на данное событие смогут отреагировать на изменения.

2.2.6. Каналы использующие пуш уведомления от бэкенда для уведомления конкретного микрофронтенда

То же самое, что и предыдущий вариант 2.2.7, кроме того, что подписка на получение данных происходит не в хабе, а в самом микрофронтенде. Это нужно в случаях, когда мы не хотим уведомлять всю систему о некоторых изменениях на бэкенде. Это очевидно не должно быть обычным вариантом и должно быть хорошо обосновано.

Пример: получение приватных данных, которое не хочется шарить всем сервисам через общедоступные события, чтоб защититься от XSS – атак, или специфической информации, которую мы хотим инкапсулировать. Это редкий, возможно скорее технический случай, но нужно о нем помнить.

2.2.7. Каналы использующие для коммуникации api на бэкенде

Так как бэкенд обычно является первоисточником информации для фронта то естественно обычный api может служить хорошим каналом синхронизации и обмена данными между фронтендами. Данный вариант плохо подходит для микрофронтендов расположенных на одной странице, так как создает множественные запросы, да и сами запросы требуют синхронизации, да и отображение ответов сделать синхроннам нужно для этого. Но естественно мы всегда пользуемся этим классом каналов для синхронизации наших микрофронтендов на разных устройствах, или задеплоенными на разных доменах (согласен, редковатый случай, но все же), а иногда просто как старый надежный способ. При этом данный класс каналов обеспечивает высокую безопасность при должной реализации на бэкенде. Кроме этого данный класс каналов позволяет синхронизировать между собой микрофронтенды на разных устройствах и разные моменты времени. Согласен, это очевидные вещи, но часто бэкенд не рассматривается как коммуникационный канал, хотя очевидно, с точки зрения фронтенда бэкенд вполне может выступать в такой роли.

Пример: использования: допустим есть два микрофронтенда, которые задеплоены по разным адресам и пока не являются частью одного и того же приложения (согласен, немного искусственный пример, но это для наглядности). Пускай на первом микрофронтенде запускается некий процесс требующий определенных действий от пользователя на первом микрофронтенде, но после некоторого шага осуществляется переход на второй микрофронтенд и процесс продолжается. При этом необходимо хранить некие чувствительные к атаке данные на протяжении всего процесса, т.е. Данные которые нельзя передать через webStorage. У нас просто не остается выхода, кроме как отправить данные на сервер и хранить состояние о протекающем процессе удаленно! При подгрузке второго мирофронтенда мы сможем получить эти данные и использовать!

2.3. Обобщение. Таблица каналов коммуникации микрофронтендов

Приведу сравнительную таблицу каналов коммуникации между микрофронтендами

Тип канала	Плюсы	Минусы
Каналы использующие browser api	1. Имеют специализацию с точки зрения передаваемых данных	1. Ограничения на передаваемую информацию
Каналы использующие global storage	1. Легко и просто синхронизировать микрофронтенды в небольших системах 2. Удобно использовать для кеширования и шаринга общей для всей системы информации	1. Плохо поддерживается при обслуживании несколькими командами 2. Нужно следить, чтоб не разрастался store
Каналы использующие синхронные коммуникационные интерфейсы	1. Удобно при принудительной синхронизации и при инициализации микрофронтенда (заменяет кеширование).	1. Не можем гарантировать работу канала, если не можем гарантировать примонтированность его компонента
Каналы использующие асинхронные коммуникационные интерфейсы	1. Удобно для уведомления и синхронизации всей системы сразу	1. Не позволяет запросить нужную информацию в конкретный момент (например при инициализации микрофронтенда)
Каналы использующие пуш уведомления от бэкенда для уведомления всей системы	1. Отлично для уведомления всей системы. 2. Делает систему максимально консистентной	1. Нужна специальная обвязка на бэкенде 2. Требует дополнительной защиты от XSS
Каналы использующие пуш уведомления от бэкенда для уведомления конкретного	1. Подходит для подписки на приватные данные для защиты от XSS	1. Нарушает архитектурный подход при котором события происходят для системы, а не для

микрофронтенда		микрофронтенда
Каналы использующие для коммуникации api на бэкенде	1. Надежно и проверено временем	1. Сложно синхронизировать независимые микрофронтенды

3. Работа микрофронтендов с данными

3.1 Инициализация микрофронтенда

С общими классами каналов мы разобрались, но как теперь правильно инициализировать микрофронтенд, используя все эти каналы? Все зависит от частного случая, микфронтенд может использовать только часть каналов, или не использовать никакие из коммуникационных каналов при инициализации, или даже вообще не использовать никакие каналы связи на всем своем жизненном цикле, если ему это не требуется. Но если мы хотим интегрировать сложные микрофронтенды, использующие кеширование информации и общение между собой, то задача немного усложняется. Хотя не на много. Диаграмма примера инициализации микрофронтенда представлена ниже на *Рис. 2*:

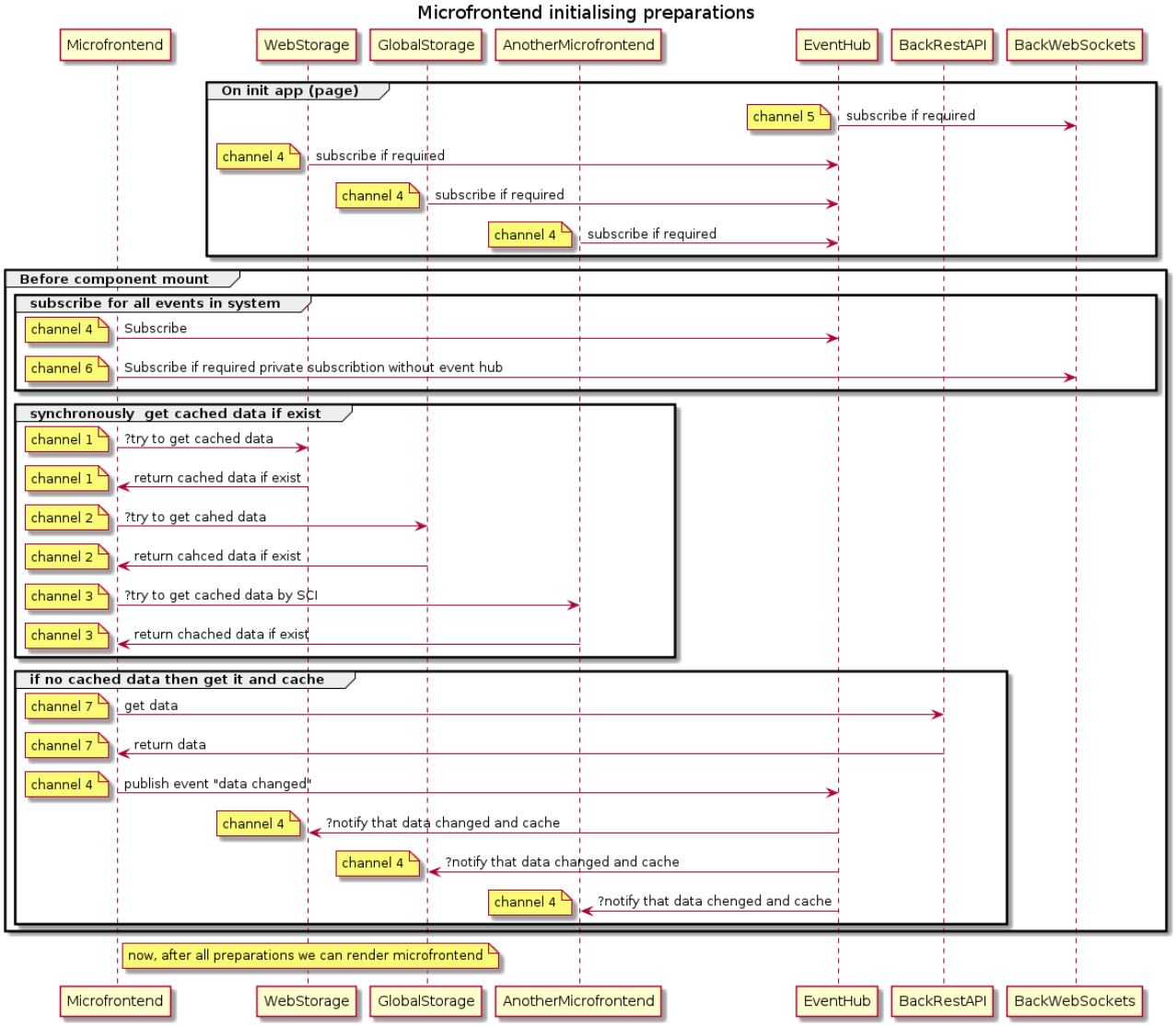

Рис. 3 Диаграмма инициализации микрофронтенда.

На диаграмме представлено 4 блока. Рассмотрим их вкратце.

1. On init app. Идея достаточна проста — перед инициализацией микрофронтенда было бы хорошо провести подготовительную работу по инициализации того, с чем он будет интегрироваться.

2. Subscribe for all events in system. Начинаем инициализацию микрофронтенда с того, что он создает свой личный приватный стор и подписывается на изменения в системе через события.
3. Synchronously get cached data if exist. Для того, чтоб не делать лишние запросы, наш микрофронтенд попытается получить закешированную информацию, если такая существует.
4. If no cached data then get it and cache. Если закешированная информация отсутствует или устарела, то мы микрофронтенд просто попытается запросить актуальные данные и при получении этих данных отправит в систему событие, содержащее информацию о том, что данные должны быть закешированы и обновлены в других частях системы.

Для хорошего UX, все время пока происходит стадия инициализации (Before component mount) нам обычно стоит уведомлять пользователя о подготовке нового микрофронтенда отображая что-то наподобие спиннеров или прогрессбаров на странице.

4. Роутинг в приложении

5. Устройство host-microfrintend

6. Устройство border-microfrontend

7. Устройство page-microfrontend

8. Устройство fragment-microfrontend
9. Шаблоны для проектов

10. Тестирование, документация и коммуникации

10.1 Роль тестирования

10.2 Роль документации

10.3 Роль коммуникаций

11.Организация бэкенда

12. Вспомогательные инструменты

12.1 Для использования внутри микрофронтендов

12.1.1 Логгер

12.1.2 Мониторинг производительности

12.1.3 Интернационализация

12.1.4 UI-библиотеки

12.1.5 Уровни доступа

12.1.6 Автогенераторы кода

12.2 Для контроля инфраструктуры

12.2.1 Система мониторинга зависимостей в микрофронтендах

12.2.2 Система мониторинга зависимостей между микрофронтендами (иерархия использования микрофронтендов)

12.2.3 Система мониторинга зависимостей между микрофронтендами (взаимодействие микрофронтендов)

13. Заключение. Все — компромисс

Понимание принципов и подходов к построению коммуникаций играет важную роль в проектировании системы и для формирования функциональный требований в том числе и кешированию. При возникновении вопросов или предложений обращайтесь по адресу victorveretennikov58@gmail.com.

14. Рекомендуемая литература

- Макконнел
- Microfrontends in action