ГЛОБАЛЬНОЕ ХРАНИЛИЩЕ ДАННЫХ НА СТОРОНЕ КЛИЕНТА

© М.З. Исаева, М.М. Цуев

ГГНТУ им. акад. М.Д. Миллионщикова, г. Грозный, Россия

Статья посвящена интеграции глобального хранилища в приложения. Также будет рассматриваться обязательное «must have» присутствие данной технологии передачи информации при проектировании современных веб-приложений, демонстрация ключевых проблем и их решения. Семантическое разбиение с последующей эксплуатацией запрашиваемых данных для применения или вывода в отдельные, составные и не зависимые части программы. Будем вникать в модульное разбиение центрального хранилища. Рассмотрим событийные привязки распределения со стороны пользователя и даже без его участия путем автоматизации этих процессов имеющимися инструментами рассматриваемой технологии в контексте темы статьи.

Ключевые слова: компонент, шаблонизатор, реактивность, вуче, состояние, мутации, асинхронные действия, геттеры getters, добытчики, глобальный файл хранения, технологии, web – приложение, скрипты.

При построении архитектуры ПО либо отдельной программы или маленького приложения основная работа направлена на разработку взаимодействий интерфейсных компонентов с запрашиваемыми серверными данными. Интерфейс в своей основе — это интерактивная оболочка между пользователем и исполняемым программным кодом как со стороны клиентского приложения, так и со стороны сервера. Выстраивается прямая линия пользователь → клиент → сервер. А также и обратная линия т.е. возвращение требуемого ответа.

Когда приложение начинает разрастаться с добавлением нового функционала, каких-то расчётных или же аналитических действий, увеличивается также и объем обрабатываемой информации. Появляются всё сложные и сложные структуры. Итак, получается, что компоненты приложения, а это переиспользуемые составные части программы, разбитые по сегментам проекта, внутри себя индивидуально требуют те или иные данные для отображения пользователю. И в каждом из них мы используем логику для запроса одной и той же информации, а это монотонное переписывание логики под каждый сегмент и отдельные элементы программного продукта, настройка под определенные условия ввода и вывода информации для передачи на сервер.

В данном контексте также существуют так называемые «шаблонизаторы» они организуют прямую связь с базой данных, по, конечно, выстроенной определенной логике. Здесь уже без посредника, а именно языка программирования, мы получаем данные прямо в разметку нашего интерфейса. Такие методы позволяют нам выстроить некую иерархию зависимостей, где одни данные уже подгружены и отображают визуальную часть, но также требуют того же языка для привнесения изменений в базу данных, уникально под требуемые критерии определенного шаблона. Получается, что полноценная отображаемая часть здесь выступает некой автономной единицей без прямых зависимостей с клиентской скриптовой логикой.

Скрипты или же сценарии по отдельности создают логику связи, ранее описанная линия пользователь → клиент → сервер линии, а более того и логику взаимодействия составных частей всего проекта. Как от родительского элемента к дочернему, так и в обратном порядке. Мы уже разобрали связь между прямой. А про логику взаимодействия поговорим далее. Прежде всего нам нужно выяснить что мы будем передавать из родителя в потомок? Передаваемые данные

условно называют свойствами элемента. Сначала мы выстраиваем некий объект, например, человек, у которого есть свойства: рост, вес, возраст и т.д. Затем выборочно передаем в дочерний элемент нужные данные, тем самым связывая их.

Сделаем промежуточный итог из всего вышесказанного. У нас возникает ряд трудностей:

- Многократное повторение огромной логики запроса в каждом элементе.
- Проблема адаптации логики запроса под каждый сегмент проекта.
- Постройках сложных запутанных зависимостей отдельных элементов.
- Полная децентрализация распределительных концепций.

Рассмотрим последнюю проблему по подробней. Децентрализация подразумевает под собой отдельное функционирование элементов данных. Нет такого головного офиса, который бы отдавал бумаги в один филиал и на изменения одного из них моментально менял бы бумаги другого филиала. Решением всего этого является глобальная система распределения, которая бы обеспечивала единую «CRUD» систему данных:

- Create — создание
- Read — чтение
- Update — обновление
- Delete — удаление

Идея глобального хранения состоит в том, чтобы каждый мог брать для себя что-то полезную, вне зависимости от привязки к каким-либо родительским элементам.

Отсюда возникает вопрос. А как именно будет работать эта система? Ведь для получения или же распределения нужных данных мы должны иметь определенную концепцию функционирования информации. На данном момент

существует множество разных подходов решения данной задачи, но в нашей статье мы рассмотрим хранилище данных в контексте веб-приложения, на примере vueх.

Vueх — паттерн управления состоянием, библиотека для приложений на vue. Он служит централизованным хранилищем данных для всех компонентов приложения с правилами, гарантирующими, что состояние может быть изменено только предсказуемым образом.

Хранилище состоит из четырех фундаментальных компонентов:

1. Состояние
2. Мутации
3. Функции getter
4. Действия

Опишем их по отдельности.

Состояние — сущность, представляемая в одной условной единице информации или же набора разной информации в большом размере. В совокупности организуется древесная структура данных, где корнем выступает само состояние, а первые ответвления целиком и полностью независимы друг от друга.

Мутации — функция, которая в качестве своих параметров берет состояние и присваиваемое значение. Вторым параметром может быть, как динамически сконструированным, так статически установленным с расширяемой логикой, имеется ввиду что присвоенное значение может иметь не одну единицу определенных данных, всё зависит от поставленной задачи.

Функции getter — функции, которые могут преобразовывать выдаваемую информации под нужные требования. Это может быть фильтрация, сортировка, видоизменение, преобразование и т.д... Getter также как мутации принимает в качестве параметра состояние, которое уже меняет и выдает запрашиваемым компонентам.

Действия — асинхронные функции, отличительной особенностью которой является параллельное выполнение с другими себе подобными функциями. Тогда как мутации и `getter` работает с состоянием напрямую, «действия» заполняют либо меняют состояние только через мутации, путем вызова их внутри себя, но для гибкости существует и прямая работа с состоянием. Однако, подобная работа с хранилищем является плохой практикой, потому что они искажают концепции технологии в контексте `vue`. Основное предназначение «действия» является запрос данных с сервера для клиента и последующее присвоение их через мутации.

Полноценное взаимодействие всех описанных частей состоит следующим образом. На старте у нас бывает пустое именованное состояние. Мутация, которая принимает в себя это состояние и динамически или статически присваиваемое значение. Действие, которое принимает мутацию и передает в неё, в нашем случае динамический параметр, значение. И последнее `getter`, который изменяет под разные нужды заполненное состояние. Теперь в действии. Вызывается «действие» оно передает значение в мутация, которая в свою очередь присваивает это значение состоянию, а следом `getter` выдает это измененную копию значение.

Мы разобрали одну из технологий глобального хранилища, она очень сильно упрощает нам работу с данными как мы рассказали выше. Но слишком большая концентрация данных в одном центре рано или поздно запутает нас. Решением для этого является модульность, иначе говоря, центральное хранилище разбивается на малые тематические секторы. Это значит, что для каждой части будет своя система данных, которая в свою очередь подчиняется главному центру.

Рассмотрим более подробно как это происходит. Компонент запрашивает определенные данные у центра, центр смотрит к чему они относятся и выдает их из своего ответ-

вления. Разбиение может быть только на один модуль. Так как модуль в модуле и т.д. уже создаст путаницу в передаче данных за счет посредников.

Теперь мы знаем как устроена система `vueх`. Теперь мы рассмотрим, как она интегрируется в приложения.

Прежде всего компонент может вызывать любую из частей `vueх`: состояние, мутация, действие, `getter`. Он также может напрямую брать состояние, а не через `getter`. Важно заметить, что состояние в отображаемых компонентах является реактивным, это значит, что, если одно и тоже состояние используемое в разных местах поменяется в одном месте, то оно поменяется везде, где оно задействовано. Также отдельный вызов мутаций обусловлен тем, что компонент может захотеть передать в качестве параметра своё локальное значение. А с оставшимся, логически понятно. «Действия» вызываются на определенные события, а `getter`-ы для получения определенных данных.

В хором выстроенном проекте взаимосвязь компонента и хранилища выстраивается так. Берется нужный `getter` и выводится в интерфейс, если рассматривать в контексте веб-приложения. В роли меняющего состояния выступает «действие», который уже через знакомые механизмы меня состояние и соответственно то, что выдает `getter`, всё это происходит на определённые события со стороны пользователя.

Однако, есть возможность автоматического вызове всех частей хранилища через жизненные циклы компонента иначе и, короче говоря, хуки, важная составляющая автоматизации вызовов, сами по себе они подразделяются на 5 типов.

1. Хуки инициализации позволяют выполнять действия ещё до того, как компонент:

- `BeforeCreate` — выполняется прямо во время инициализации компонента. Данные ещё не стали реактивными, а события не настроены.

- **Created** — вы сможете получить доступ к реактивным данным и активным событиям. Шаблоны и виртуальный DOM ещё не встроены (**mounted**) и не отрисованы.

2. Хуки встраивания — позволяют получать доступ к компоненту непосредственно перед первой отрисовкой или сразу после неё. Однако эти хуки не выполняются в ходе отрисовки на стороне сервера:

- **BeforeMount** — выполняется до первичной отрисовки, а также после компилирования шаблона или функций отрисовки. Вероятно, вам никогда не потребуется использовать этот хук. Помните, что он не вызывается в ходе отрисовки на стороне сервера.

- **Mounted** — получаем полный доступ к реактивному компоненту, шаблонам. Обычно его используют для извлечения данных для компонента.

3. Хуки обновления — вызываются, когда изменилось реактивное свойство, используемое вашим компонентом, или, когда что-то ещё приводит к перерисовке. Эти хуки позволяют получить доступ к циклу «отслеживания-вычисления-отрисовки» компонента:

- **BeforeUpdate** — выполняется после изменения данных в компоненте и начала цикла обновления, сразу перед патчингом и перерисовкой. Этот хук позволяет получить новое состояние любых реактивных данных в компоненте, прежде чем он будет отрисован.

- **Updated** — вызывается после изменения данных в компоненте и перерисовки. Если нужно изменить свойства, такой хук — самое безопасное место для этого.

4. Хуки уничтожения позволяют действовать, например, прибратся или отправить данные для аналитики, после уничтожения компонента. Эти хуки срабатывают при демонтаже элемента:

- **BeforeDestroy** — выполняется непосредственно перед монтажом. Ваш компонент ещё полностью функционирует.

Если вам нужно очистить события или реактивные подписки, то `beforeDestroy` — самое подходящее для этого место.

- `Destroyed` — к тому моменту, как вы добрались до этого хука, от вашего компонента мало что осталось. Всё, что было к нему прикреплено, уже уничтожено. Вы можете использовать хук для последней очистки или, словно подлый тихушник, проинформировать удалённый сервер об уничтожении компонента.

Так мы можем установить на любой из перечисленных хук методы частей хранилища. Тем самым автоматизируя процесс функционирования путем привязки вызовов к циклу жизни компонента, начиная от инициализации и заканчивая уничтожением.

Итак, мы разобрали как работает система без глобального хранилища, выявили следующие из этого проблемы, пришли к определённом решению через ключевой элемент перечисление (проблем) и рассмотрели, как одна из технологий организует это решение.

ЛИТЕРАТУРА

1. Эван Ю. Централизованное управление состоянием для Vue.js [Электронный ресурс]: документация Vue js – 2020 г. – режим доступ: <https://vuex.vuejs.org.ru>

2. Эрик Х., Листуон Б. Vue js в действии [Электронный ресурс]: книга – 2020 г. – режим доступ: <https://vuex.vuejs.org.ru>

3. Эван Ю. Introduction Vue js состоянием для Vue.js [Электронный ресурс]: документация Vue js – 2020 г. – режим доступ: <https://vuex.vuejs.org.ru>

4. Сериков А. Глобальное хранилище данных [Электронный ресурс]: курс лекций в Интуит- 2020г. – режим доступа: https://intuit.ru/studies/higher_education/3406/courses/455/lecture/10158?page=4

5. Компания Инфопульс Жизненные циклы(хуки) [Электронный ресурс]: – 2020 г. – режим доступа: <https://habr.com/ru/company/infopulse/blog/140456/>

GLOBAL DATA STORAGE ON THE CLIENT SIDE

© M.Z. Isaeva, M.M. Tsuev

GSTOU named after acad. M.D. Millionshchikov, Grozny, Russia

The article is devoted to the integration of global storage into applications. The mandatory “must have” presence of this information transfer technology in the design of modern web applications, demonstration of key problems and their solutions will also be considered. Semantic partitioning with subsequent exploitation of the requested data for application or output into separate, composite and independent parts of the program. We will delve into the modular partitioning of the central storage. Let’s consider event-based bindings of distribution on the part of the user and even without his participation by automating these processes with the available tools of the technology in question in the context of the topic of the article.

Keywords: component, template engine, reactivity, vuex, state, mutations, asynchronous actions, getters getters getters, global storage file, technologies, web application, scripts.