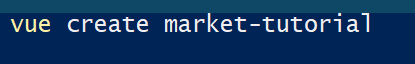
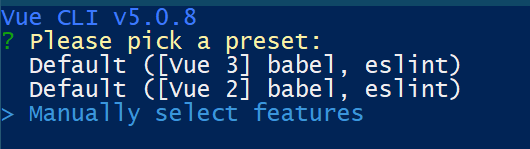
# Введение

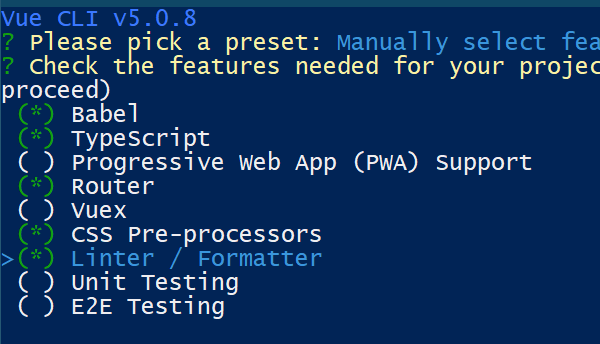
Сегодня мы познакомимся с Vue composition API, TS и стейт менеджером pinia на примере простенького интернет-магазина. Результатом будет каталог, данные которого мы будем получать через axios, в каталоге будет реализован фильтр по категориям, цене и поиск. Помимо каталога реализуем корзину, которая будет запоминаться в Local Storage. По ходу урока вам будут встречаться ссылки на документацию для более глубокого ознакомления. Урок рассчитан на то, что вы уже знакомы с Vue хотя бы на начальном уровне (директивы, хуки, синтаксис). Приступим.

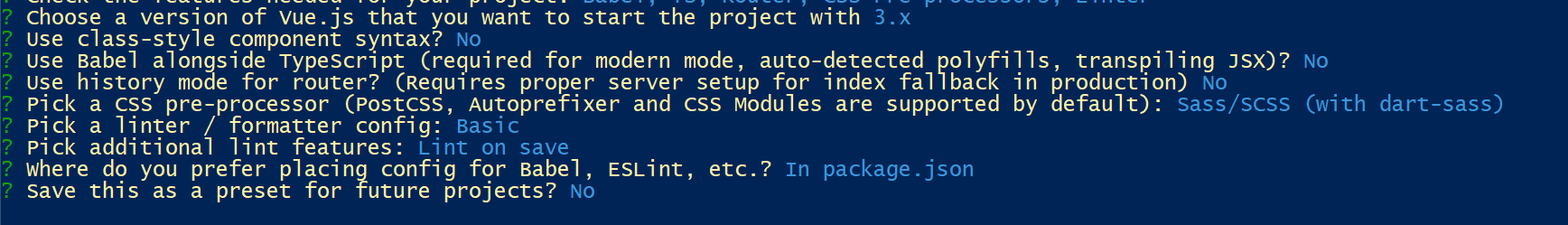
# Подготовка проекта

Откройте cmd или PowerShell и создайте новый проект. Используйте настройки проекта, которые приведены ниже









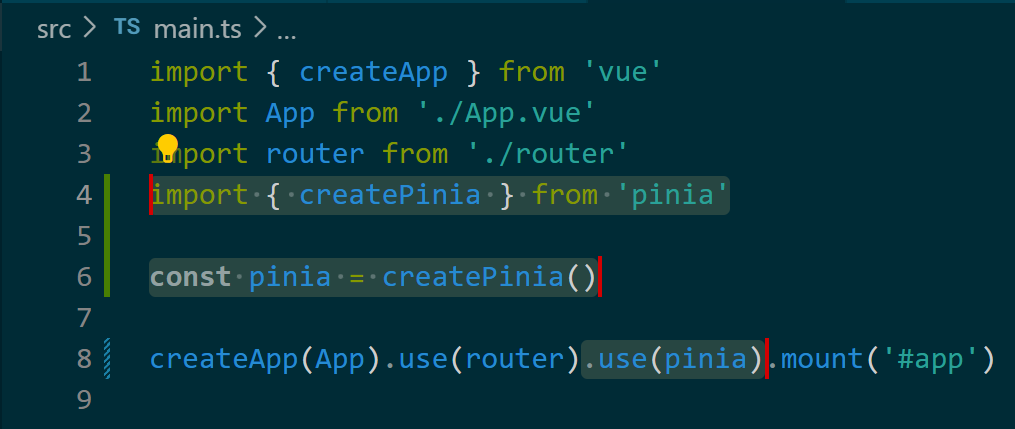
После того, как проект собрался, установим необходимые зависимости, начнем с pinia.

Наверняка вы слышали или даже работали с Vuex. Vuex, как и pinia – это библиотека управления состояниями (стейт менеджер) для Vue.js, как не удивительно, нужны такие библиотеки для управления состоянием приложения, если проще – у нас есть глобальная переменная, которую мы можем читать или менять, чем-то похоже на локальное хранилище за некоторыми отличиями:

1. Масштабируемость и производительность:
   1. - Localstorage: встроен в браузер, позволяет хранить небольшие (<5МБ) объемы данных в виде пар ключ-значение.
   2. - Стейт менеджер: основан на концепции использования компонентной архитектуры и более масштабируем, что позволяет управлять большим объемом данных и работать с ними более эффективно.
2. Управление состоянием:
   1. - Localstorage: Пользователи могут устанавливать, получать и удалять значения, связанные с ключами, используя методы Localstorage. Однако нет никакого встроенного механизма для управления состоянием или реакции на изменение данных.
   2. - Стейт менеджер: позволяет легко управлять состоянием вашего приложения и обновлять компоненты только в случае изменения данных. Предоставляет API для создания хранилищ, выполняя действия, изменяя состояние и получая доступ к данным из компонентов Vue.
3. Реактивность и отслеживание изменений:
   1. - Localstorage: не предоставляет возможности для реактивности или автоматического отслеживания изменений данных. Приложение должно явно обновлять данные после сохранения в Localstorage.
   2. - Стейт менеджер: предоставляет реактивные getter и setter для объектов хранилища, что позволяет автоматически отслеживать и реагировать на изменения данных.

Помните, localstorage хранит в себе данные до того момента, пока их не удалит разработчик или пользователь, стейт менеджер хранит данные в ОЗУ до того момента, пока страница не обновлена.

Вернемся к знакомству с [pinia](https://pinia.vuejs.org/). Установите ее npm i pinia и проинициализируйте в файле main.ts, который находится в папке src. В дальнейшем, путь к файлу не будет писаться, вы сможете его видеть на скриншоте в самой верхней его части.



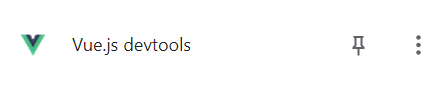
Создадим наше хранилище



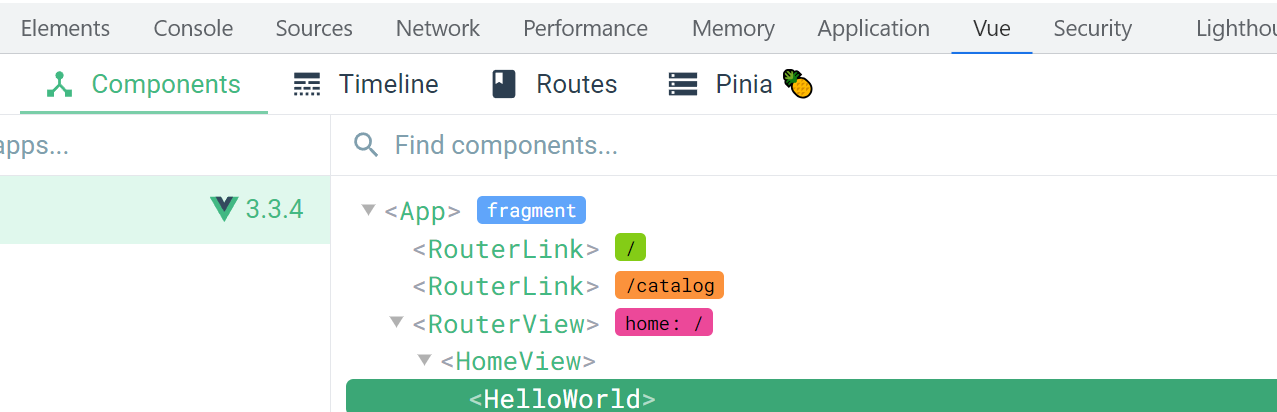
Вы можете назвать константу хранилища как-угодно, но лучшей практикой будет использование наименований useЧтоХранимStore. Функция [defineStore](https://pinia.vuejs.org/core-concepts/) принимает в себя строковый ключ и объект, отражающий наше хранилище. Не будем сейчас долго останавливаться на хранилище, укажем только state. [State](https://pinia.vuejs.org/core-concepts/state.html) хранит наши начальные данные, выполнен он в виде колбэка, возвращающим объект с данными, в нашем случае – пустой массив продуктов.

Для доступа к данным с сервера будем использовать axios, вы уже должны быть с ним знакомы.

npm i axios

Дополнительно его настраивать не нужно. Рекомендую проверить, есть ли в вашем браузере vue-испектор. В панели расширений он выглядит примерно так 

И следующим образом в консоли

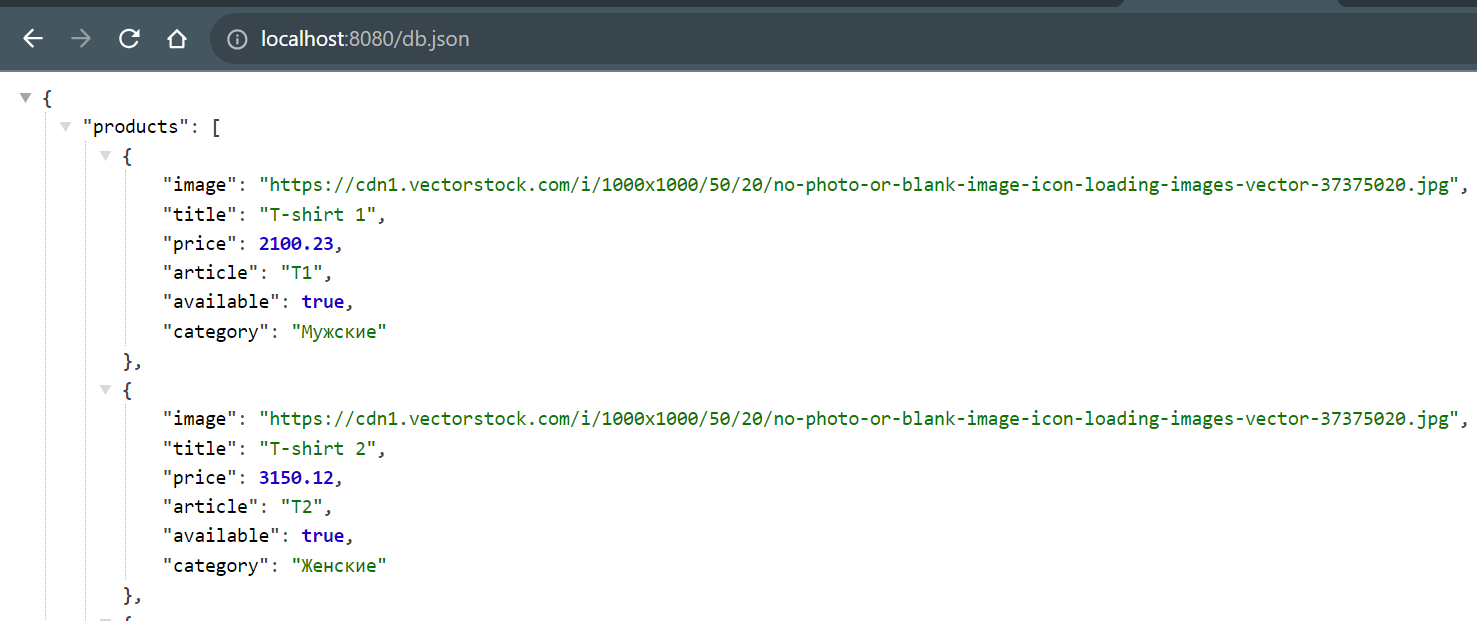


Если такого вы у себя не нашли, вот ссылка на расширение для [Chrome](https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd) и для [Edge](https://microsoftedge.microsoft.com/addons/detail/vuejs-devtools/olofadcdnkkjdfgjcmjaadnlehnnihnl?hl=ru).

Изначально, планировалось, что мы будем использовать json-server для доступа к данным, но он работает нестабильно. Однако, аналогично json-server мы можем использовать наш vue сервер, для этого просто расположите json файл с данными в папке public



Заполните вашу импровизированную бд продуктами. Бд будет доступна по адресу http://localhost:8080/имя\_файла.json



# Не придумал заголовок

Опишем подробнее наше хранилище. Как я говорил ранее, сегодня мы немножко познакомимся с TS. Для описания структур данных в TS предусмотрены две очень похожие вещи: интерфейсы и типы.

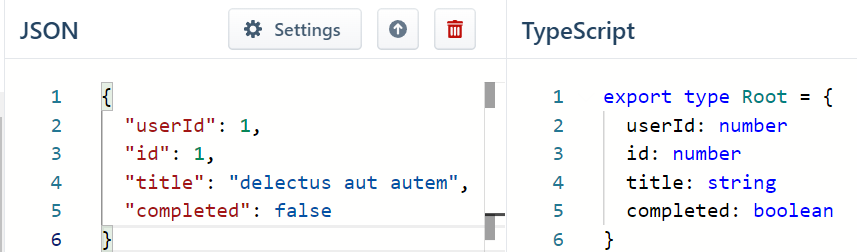
Интерфейсы используются для объявления формы объекта или класса. Они определяют список свойств и их типы, которыми должен обладать объект, чтобы считаться совместимым с интерфейсом.

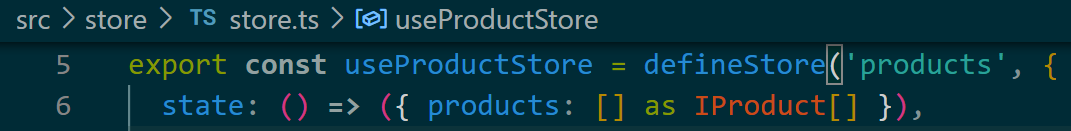
Типы в TypeScript позволяют нам создавать пользовательские типы данных и сочетать их с помощью операторов, таких как «union» и «intersection». Они более гибкие и могут использоваться для объединения или пересечения нескольких типов, а также работы с более сложными структурами данных.

Определим тип для продуктов. Для этого просто укажите поля продукта и их тип



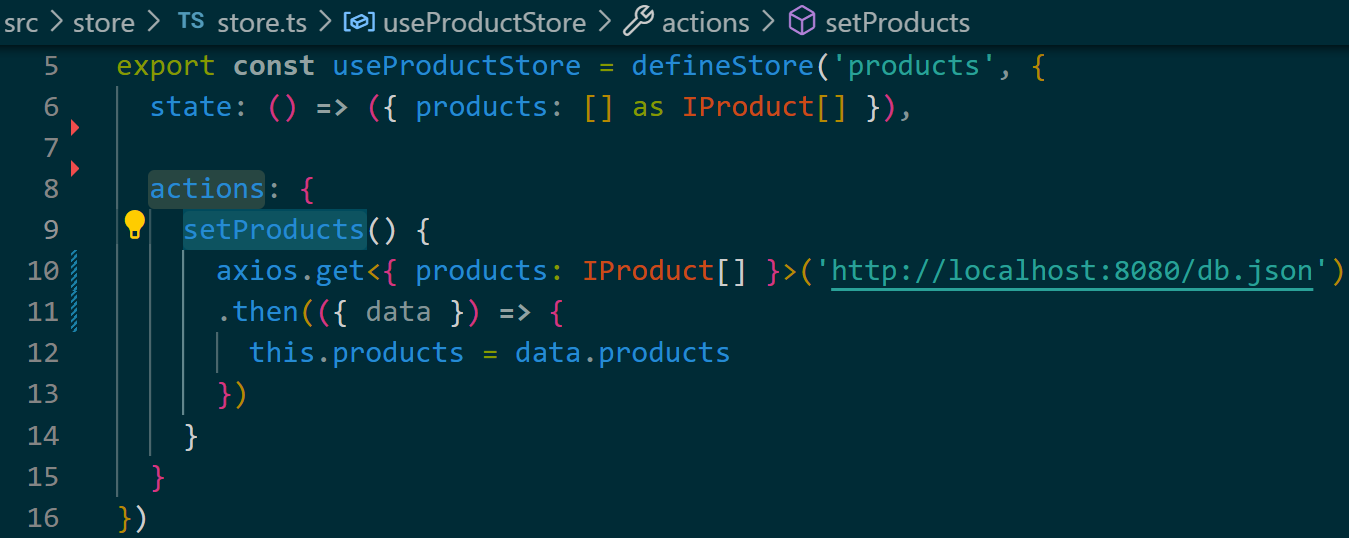
Если это вам показалось сложным, для конвертации JSON в типы есть отличный [инструмент](https://transform.tools/json-to-typescript)



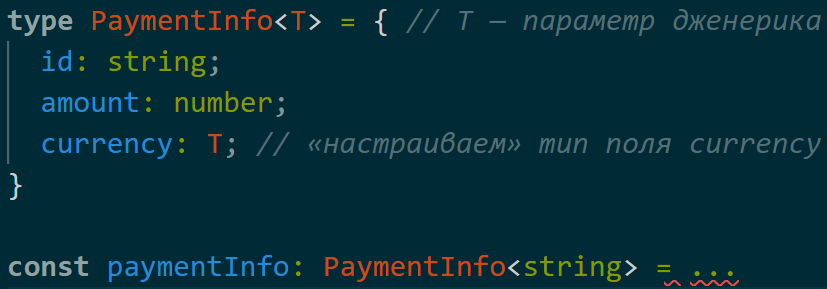
В нашем state укажите тип для продуктов, если автоматического импорта не произошло, выполните его самостоятельно

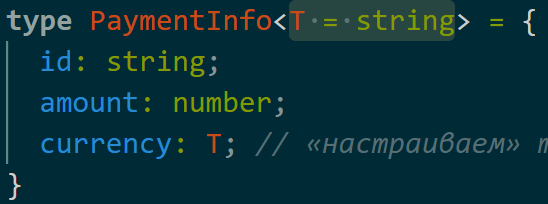
Мы буквально говорим создай ключ объекта (имеется ввиду объект state) products в котором будет лежать массив, массив будет заполнен нашим типом. В конце нам обязательно следует указать [] обозначающий массив. IProduct описан как объект, очевидно мы не можем сказать «создай мне массив как объект» ([] as {}).

# Actions

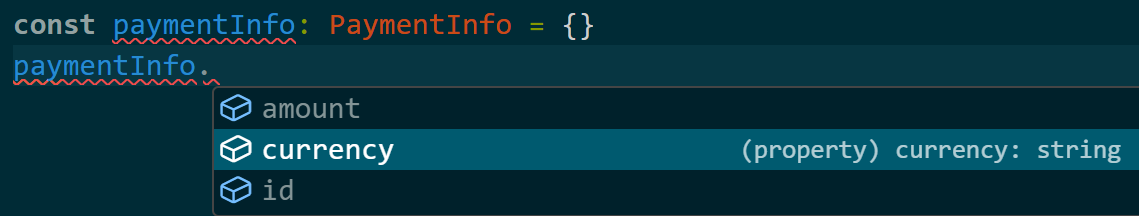
Давайте познакомимся с [actions](https://pinia.vuejs.org/core-concepts/actions.html). Actions во многом похожи на методы (methods) в vue компонентов, actions могут быть асинхронными, что отлично подойдет для обращения к API

Обратите внимание на то, что идет после get. «Штучка» которая обернута в треугольные скобки носит название Generic (дженерик) это такой обобщённый тип, используемый для описания похожих типов. Дженерик — это каркас, внутренности которого заполняет разработчик (в нашем случае разработчик axios). Программист, который описывает обобщенный тип, никогда не знает, что именно туда решит записать тот, кто будет этот тип использовать. Дженерики пишутся примерно так

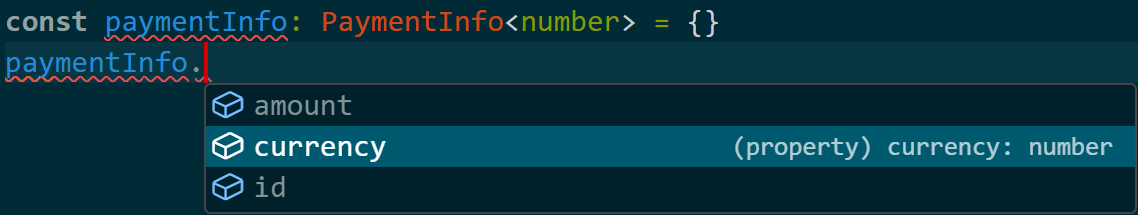


На выходе получим, что у константы paymentInfo должны быть поля id – string, amount – number и currency – string. Для T можно задать дефолтное значение, делается это так 

Теперь мы знаем, что у нашей константы есть конкретные поля. Без указание типа для Т currency стал строкой

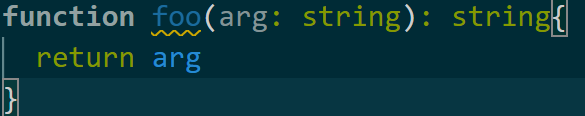


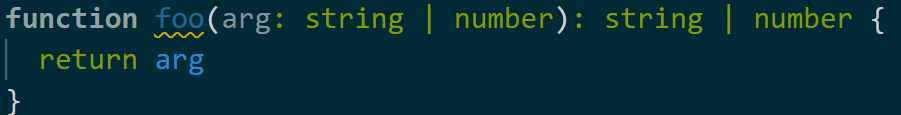
Если мы передадим в наш дженерик тип для Т, он очевидно поменяется

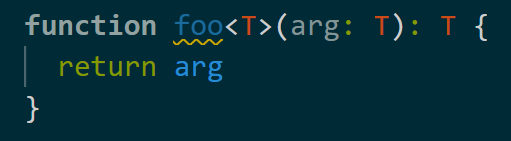


Возможно это пока что выглядит бесполезным, приведу последний пример, затем увидим еще раз как это помогает нам на практике

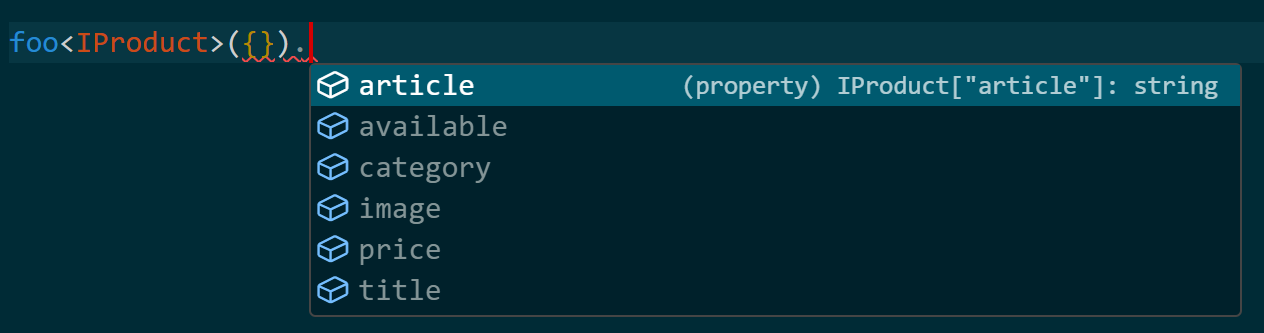
Для примера опишем какую-то функцию, которая вернет нам данные которые мы ей передали



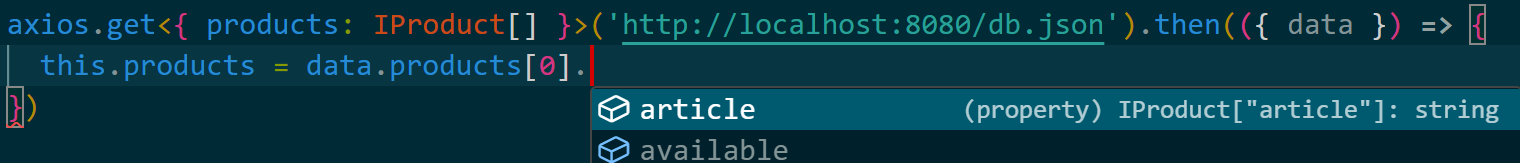
Мы строго описали, что arg – строка, функция вернет тоже строку. Но если нам нужно принимать в функцию не только строку, а еще например числа? Мы можем дописать функцию до такого вида, но стоит понимать, что сильно страдает расширяемость такой функции, возможно нам придется работать с нашими собственными типами в этой функции. Для таких задач, которые требуют масштабируемости придуманы дженерики



Теперь мы знаем что ждать от фукции foo



Отлично, вернемся к нашему API запросу. Благодаря типизации, мы (и что немало важно, IDE) знаем, что нам придет в ответе от сервера

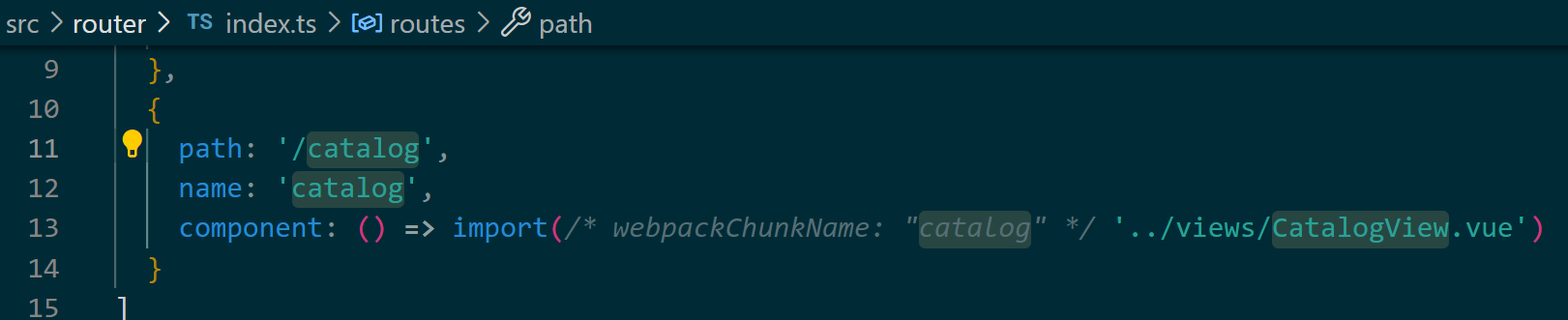


Возможно, вас заинтересовала запись ({data}) вместо условного (response)=>response.data, так мы напрямую обращаемся к тому самому полю дата в возвращаемом объекте промиса от Axios, получилась короткая запись response.data

Давайте вызовем наш action, чтобы данные наконец пришли в state. Делать это будем в самом главном компоненте



Опять что-то новое? Начнем сверху, в одной из ссылок я поменял директорию, можете пока этого не делать, но если решитесь, не забывайте менять и файл роутера



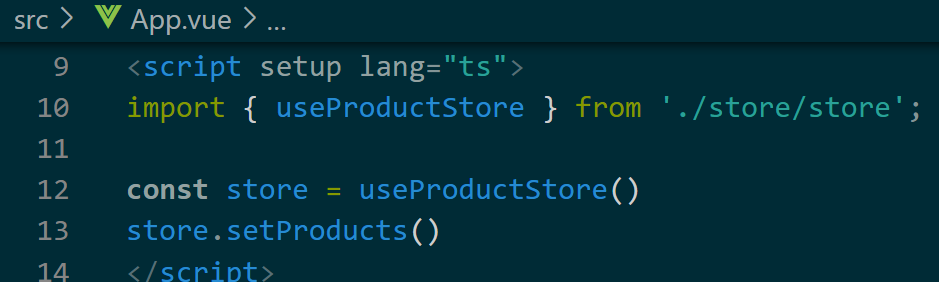
И создать страницу CatalogView. Теперь к интересному.

# Введение в Composition API

Зачем это все? Использовали старый добрый Options API, VUEX и не жаловались, все было хорошо. Причин несколько:

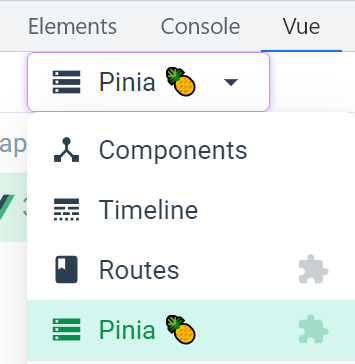
1. Типизация. Что vuex, что options API сложно поддаются типизации. TS используется сейчас повсеместно, на это есть причины: ускорение написания и рефакторинга кода(если разработчик знаком с TS, скорость разработки действительно растет), более простая масштабируемость, подсказки, автоматический отлов возможных ошибок, опечаток
2. Простота. Pinia в отличии от предшественника имеет только state,actions, getters, у Vuex в дополнение к этим трем, есть еще и отдельный слой мутаций, pinia легка не только в понимании и типизации, но еще и в весе (1KB). В сочетании с новым Composition API компоненты разделяются до условной бесконечности, писать чисто стало проще, можно делать переменные нереактивными, благодаря чему повысилась производительность.

В паре пунктов описать достоинство новых технологий над старыми не выйдет, оставлю ссылку на [документацию](https://vuejs.org/guide/essentials/reactivity-fundamentals.html) для ознакомления и приступим к разбору происходящего.

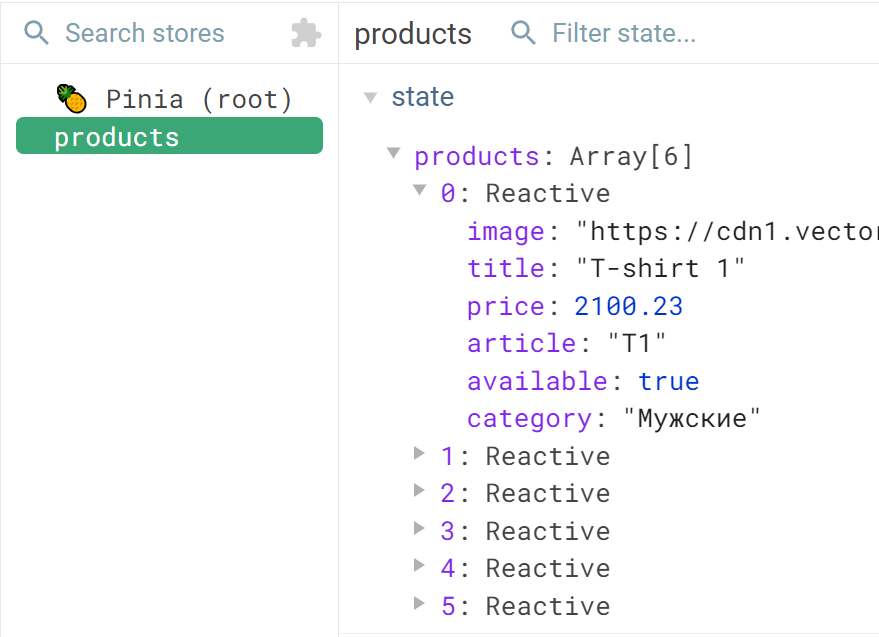


Изменения заметны с первой строчки, lang указывает, что мы намереваемся писать на TS, setup – один из способов сказать, что мы используем новый АПИ. Импортируем наше хранилище и записываем его в константу store, после чего вызываем action. Всё. С старым АПИ нам пришлось бы писать побольше, верно? А без знания хуков, еще и гадать, где же выгоднее вызвать action.

Запустите ваш проект, если еще не сделали этого, откройте его в браузере. Посмотрим что у нас произошло в хранилище, для этого нужно открыть vue-devtools (тот самый инспектор, о котором мы говорили ранее). До конца проекта, советую вовсе не закрывать консоль. Перейдите в раздел Pinia

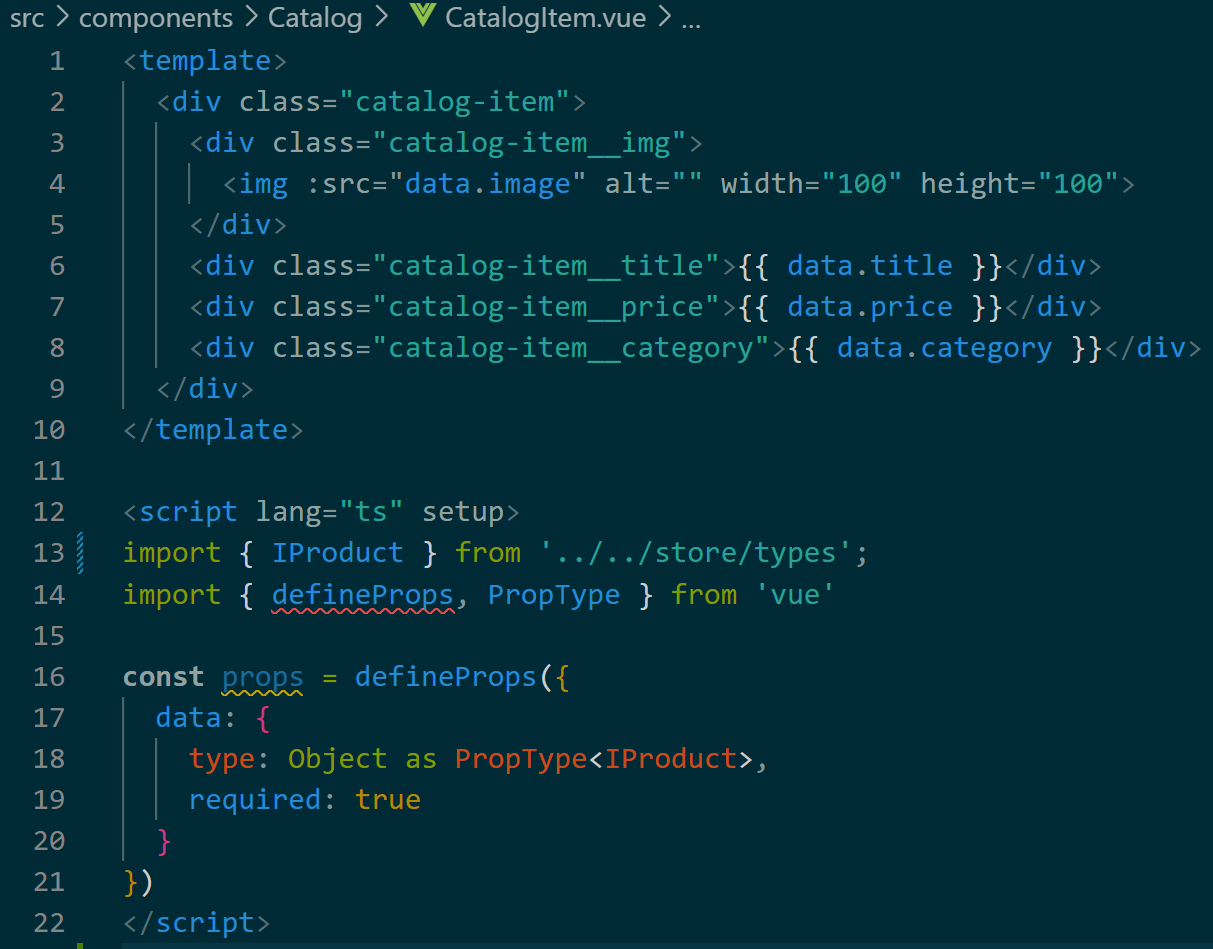


Products – ключ, который мы указывали в самом начале для defineStore



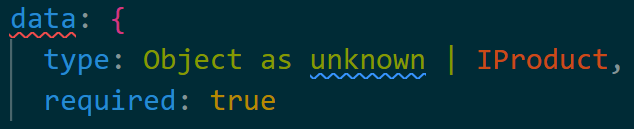
Отлично, давайте выведем все наши продукты. Уверен, вы уже представили, как просто в CatalogViews выводите массив полученный через useProductsStore().products, обращаясь напрямую в state, но это не совсем хорошая практика.

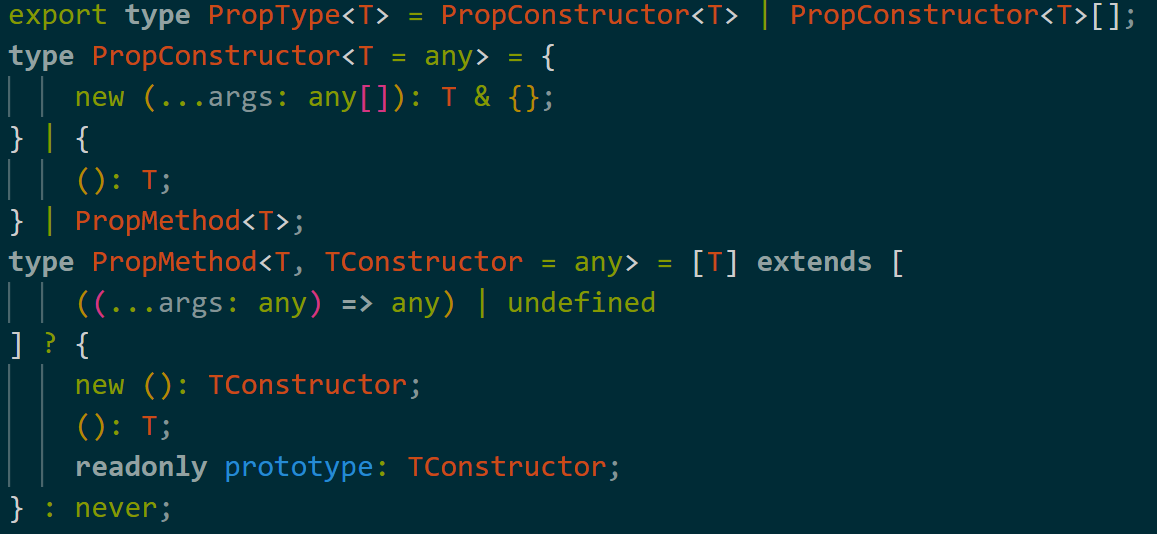
Давайте начнем с малого. Из чего вообще должен состоять каталог? Там точно должны быть товары, товары должны быть заключены в какой-нибудь список/лист/сетку чтобы можно было контролировать структуру выводимых карточек, также нам понадобится фильтр. Все это придется поместить в CatalogView. Очевидно нам придется разбить все это на компоненты. Начну с базовой структуры – карточка продукта.



Думаю, нет смысла объяснять вам что происходит в template. В скрипте мы опять видим отклонения от привычного. Props объявляется через функцию defineProp, а его тип через PropType. Почему мы не можем использовать запись как Object as IProduct или {} as IProduct? Проблема в том, что такая вещь как Object очень большая и абстрактная, у него есть методы, прототипы, проще говоря у нашего IProduct и конструктора Object общих только две фигурные скобки.

Мы могли бы вывернуться из этой ситуации и записать что-то в духе



Но такая запись перезапишет весь Object, чего делать тоже нельзя. Поэтому мы используем любезно заготовленный разработчиками Vue дженерик. Так он выглядит внутри 

Если вчитаться – ничего сложного нет. Но давайте идти далее. Предлагаю сделать список, в который мы будем выводить наши карточки



Здесь уже ничего нового нет, если сравнивать с предыдущим компонентом: получаем проп, типизированный как массив с IProduct внутри и директивой v-for выводим карточки, не забывайте передать карточкам необходимые данные и задать уникальный key.

К фильтру вернемся позже, давайте посмотрим на наш каталог. Предварительно разместите в CatalogView нужные компоненты



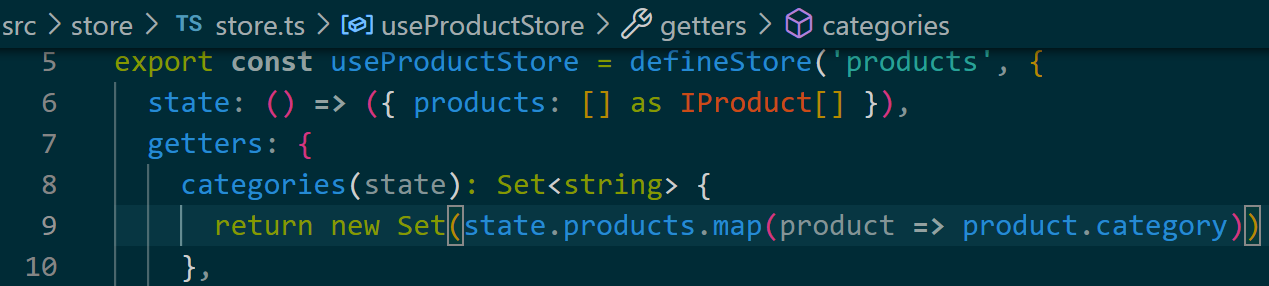
Помните, раньше все переменные как минимум внутри data были реактивными? Теперь это не так, чтобы сделать переменную реактивной, используется следующий синтаксис

Const имя\_переменной = ref(начальное значение). Также вы могли заметить, что рядом с нашим типом красуется | undefined, это говорит о том, что если продукты не дошли (отсутствуют в хранилище), мы принимаем undefined. Чтобы менять значение нашей реактивной переменной, мы обращаемся к полю value. имя\_переменной.value = новое значение.

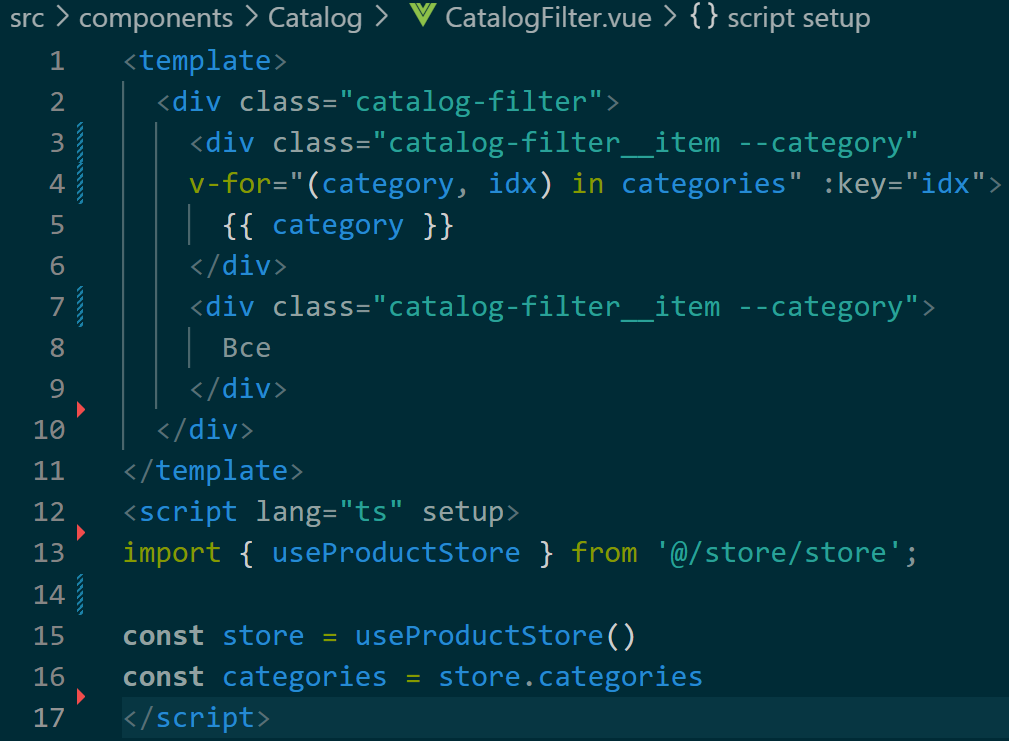
Как я уже говорил, использовать state напрямую чаще всего плохая идея. Для этого придумали [getters](https://pinia.vuejs.org/core-concepts/getters.html)

# Getters

Представьте, что вы пишете фильтр по категориям. Человек жмет на категорию, товары другой категории пропадают, жмет на другую категорию, появляются товары только этой категории. В таком случае странно будет обрезать начальный массив, верно? Если человек выбрал категорию ХХХ у него останутся товары только такой категории, чтобы вывести остальные придется получать данные снова, это еще один запрос к серверу, который может быть не один. Собственно getters и нужны для подобных вещей, во многом они похожи на computed из vue, принимают в себя state, как аргумент и обязательно должны что-то возвращать. Напишем для начала что-то совсем простое – получение категорий. Зачем нам самим статично прописывать категории, если появится еще сотня товаров с сотней новых категорий, будет очень неприятно.



Здесь мы просто пробежались по массиву продуктов и сложили в [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), который уберет дубликаты, категории продуктов. Выводим категории в компоненте фильтра

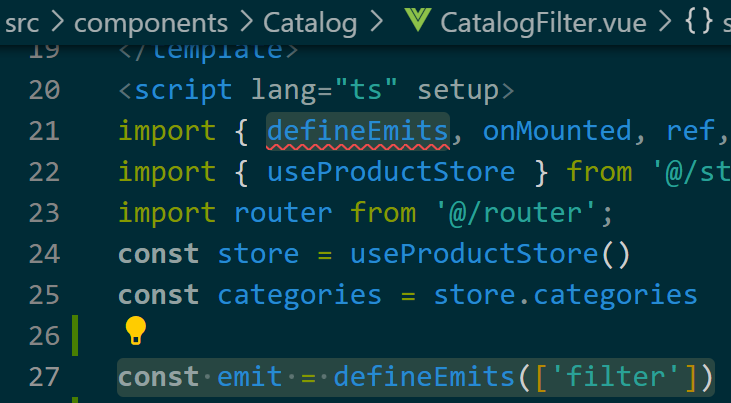


Таким нехитрым образом в хранилище появились категории

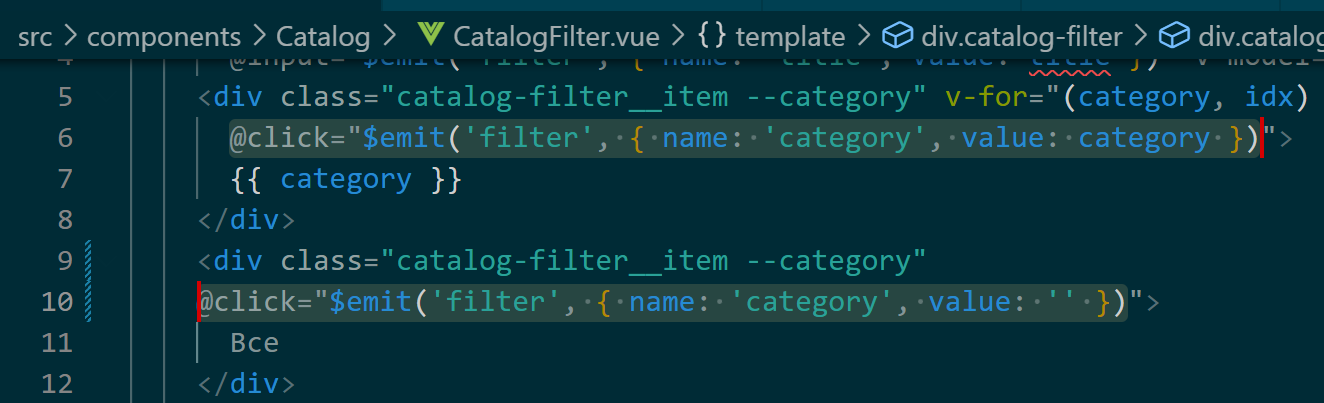
Фильтры можно реализовать разными способами. Можно хранить параметры фильтров в стейт менеджере, в какой-то переменной на странице каталога, но я предлагаю хранить значения фильтров как query параметры, прям в ссылке.

Например так

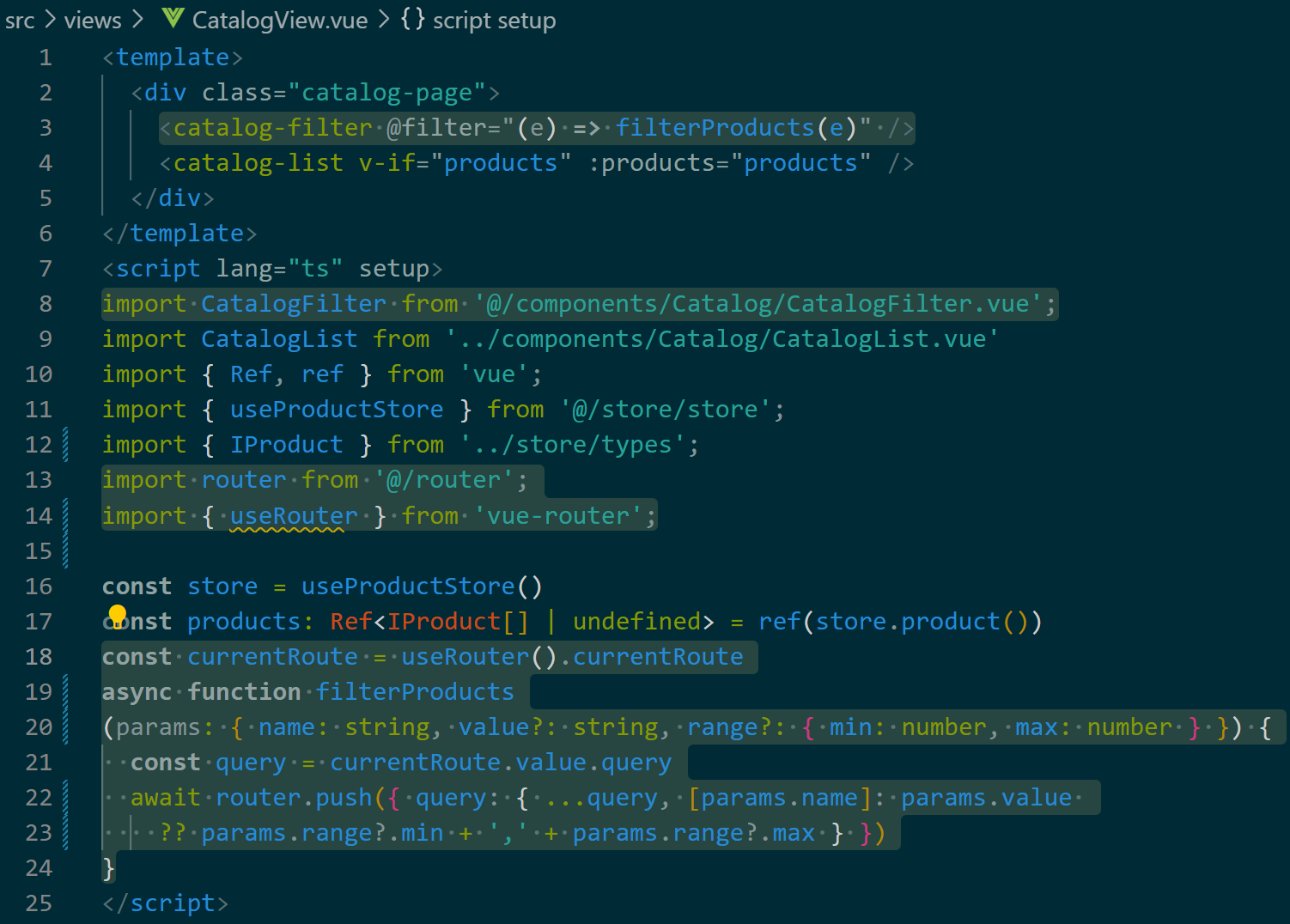


Концепция ясна, query параметр будем доставать непосредственно из route. Но т.к мы будем вызывать действие в компоненте фильтра, а поменять нам нужно будет значение переменной в файле CatalogView, будем использовать [emit](https://vuejs.org/api/sfc-script-setup.html#defineprops-defineemits) 

По клику на категорию будем его активировать. Первым значением передадим имя события, вторым аргументом будет значение



И подпишемся на событие уже на странице каталога



Здесь мы описали функцию, которая принимает в себя объект, содержащий название ключа, по которому фильтруем и необязательные значения value и range, range будет использоваться когда нам нужно будет использовать диапазон, например для цены.

В функции вас может смутить оператор ??, это так называемый nullish оператор, если значение слева от него вернет null или undefined, будет использоваться правая часть выражения, такое короткое тернарное выражение.

Добавим фильтр по цене и начальные значения для них

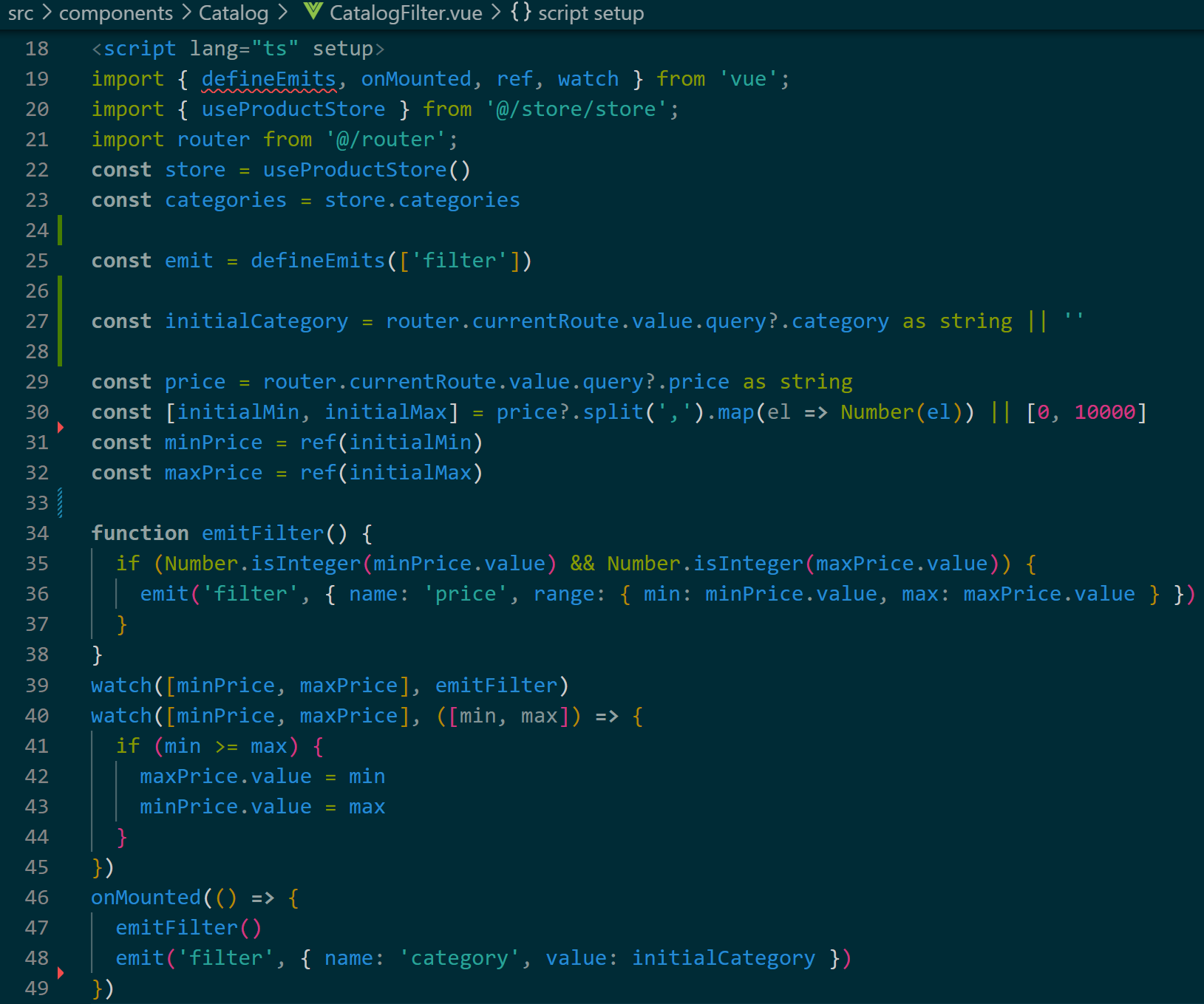


Начальные значения обращаются к роуту, если в роуте значение отсутствует, будем использовать от 0 до 10000, тоже самое для категорий

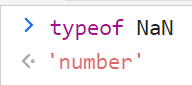
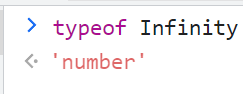
Теперь, сделаем так, чтобы при создании компонента на странице, у нас срабатывало событие



С watch вы уже должны быть знакомы, здесь суть та же – слежка за перинными, только теперь он стал короче, первый аргумент – отслеживаемые переменные, второй – функция, которая выполнится при изменении. Также, если минимум будет введен больше, чем максимум, они должны поменяться местами (46-51 строки)



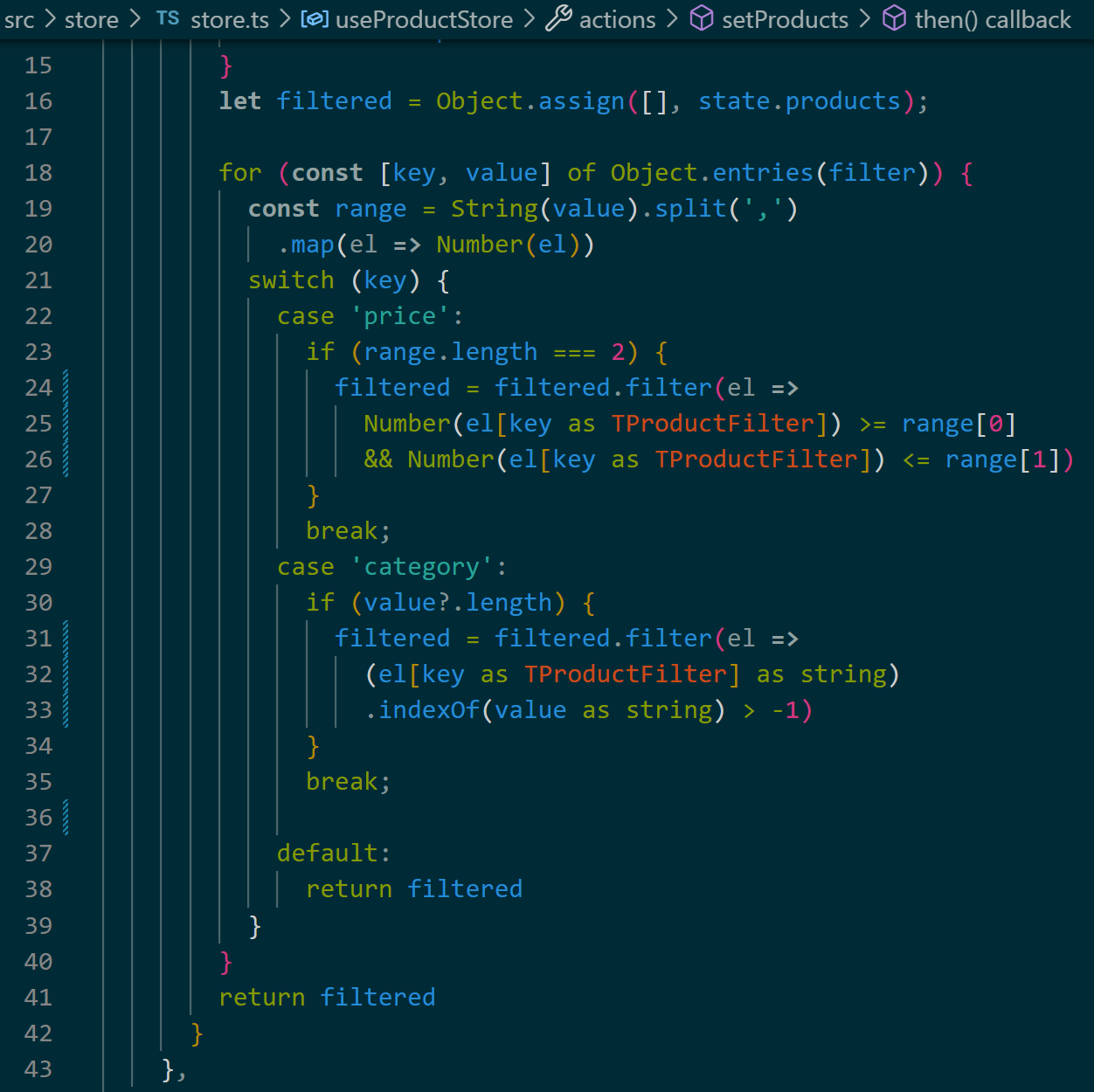
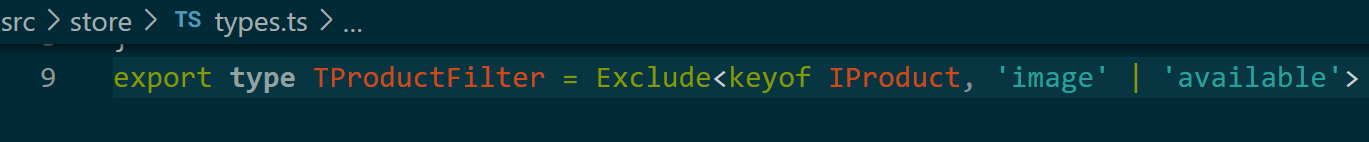
Для емитера, который отдает цену я добавил проверку, являются ли оба значения нормальными числами, как вы знаете например NaN или Infinity тоже числа, но для фильтра они не подходят

Давайте напишем геттер для фильтрованных продуктов

Создадим геттер, но, как мы помним, он может принимать только один параметр – стейт, однако, есть способ передать в него больше значений. Мы передаем необязательный параметр фильтра, при его отсутствии, просто вернем продукты как они есть. Далее, создадим переменную, в которую скопируем наши продукты, ведь сам стейт менять не стоит. Если вы не понимаете, почему нужно именно копировать и почему просто знак = нас не спасет, советую освежить знания JS [тут](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) и [тут](https://learn.javascript.ru/). Далее мы пробежимся по фильтру-объекту циклом for и отфильтруем товары согласно нашим данным

Мы используем switch case для того, чтобы удостовериться какой критерий мы сортируем, не всегда если значение строковое нужно использовать indexOf, или когда прилетает цифра смотреть на цену, такое решение хорошо масштабируется. Также для фильтра мы использовали новый тип, основанный на уже существующем интерфейсе, но без полей image и available, которые в фильтрации не нуждаются, их мы исключили при помощи утилиты [Exclude](https://www.typescriptlang.org/docs/handbook/utility-types.html#excludeuniontype-excludedmembers), которая принимает в себя объединенный тип(UnionType) и поля, которые следует исключить; оператор [keyof](https://www.typescriptlang.org/docs/handbook/2/keyof-types.html) принимает тип объекта и создает строковое или числовое объединение его ключей, что и дает нам тот самый UnionType для Exclude. В цикле я сделал переменную range, которая пробует разбить значение на массив с разделителем в виде запятой.



Если ни один из ключей не совпал, вернем то, что получилось отловить, так приложение не сломается на полпути. Вызовем геттер и присвоим его значение переменной, которую мы отдаем нашему списку товаров



Попробуйте сами расширить фильтр, например, вы можете добавить поиск по названию. Советую выполнить это самостоятельно, ниже будет пример кода, но вы сами прекрасно понимаете, что просто переписать код !== что-то понять.

Вот и моя реализация поиска по названию товара.

