Template

Фильтр разбит на 3 div’а с классом dropdown. Первый блок – список турниров, второй блок – список клубов, принимающих участие в выбранном в первом блоке турнире в текущем сезоне. В третьем блоке – список игроков, заявленных за выбранный клуб в розыгрыше выбранного турнира.

Внутри каждого рендерится список. В блоке соревнований он всегда стандартен, так как передается в компонент сервером во время рендера страницы. Список в блоке клубов и игроков варьируется в зависимости от выбранного пункта в списке предыдущего блока. Каждый элемент списка хранит в себе название турнира/клуба либо имя игрока и соответствующие им так называемые слаги (по которым происходит поиск в базе данных)

Логика фильтра заключается в поэтапном выборе параметров фильтрации. Т.е. выбрать клуб мы может только после выбора турнира, а игрока только после выбора соревнования и клуба. Соответственно, выбирая нулевой элемент списка (т.е. передающий слаг 'all'), мы сбрасываем все последующие блоки. Т.е. выбирая нулевой элемент списка в блоке турниров, сбрасывается весь фильтр целиком.

У каждого блока забинжен класс **active**. Бинд класса срабатывает, если из отрендеренного списка выбран любой пункт, отличающийся от нулевого.

У второго и третьего блока есть бинды на классы **disabled** и **loading**. Класс **disabled** присутствует у второго блока, если в первом не выбран турнир или загрузка списка клубов еще не закончилась. У третьего, если не выбраны клуб либо соревнование либо загрузка списка игроков не завершилась. Проверка выбора турнира/клуба/игрока производится по значению соответствующих им параметрам запроса (co, club, pl. Подробнее о них в разделе Computed Properties). Класс disabled снимается у блока только в случае удачной обработки запроса и рендера списка. В случае, если запрос списка клубов или

игроков вернет ошибку, он будет отправляться заново, пока не вернет успешный ответ. Класс **loading** тригеррится в момент запроса списка клубов и игроков.

На каждый блок вешается слушатель события клик, который обращается к методу isOpened, регулирующим управление классом "open" у блоков, передавая в качестве аргумента методу нативное событие $event.

Блок турниров:

```
@click.self="isOpened($event)"
```

.self обращается к методу isOpened только при клике непосредственно на родительский div отменяя всплывание события click с дочерних элементов списка.

Блок клубов:

```
@click.self="isOpened($event) && co.length"
```

Вызываем метод только если блок разблокирован (выбран турнир).

Блок игроков:

```
@click.self="isOpened($event) && (co.length && club.length)"
```

По аналогии с предыдущим: вызываем метод, если выбраны турнир и клуб (в противном случае, в параметрах co и club будет записано пустое значение и условие не выполнится)

Формирование списка внутри каждого блока разберем на примере блока клубов.

```
<ul v-if="co.length">
<li v-for="club in clubsList"
                    @click="pickClub($event, club.slug, club.name)"
                    :class="{picked: pickedClubSlug == club.slug}">{{
club.name }}</li>
</ul>
```

В директиве v-if проверяем, что выбран турнир.

Событие @click вызывает метод выбора клуба. В метод вы передаем нативное событие, слаг и название клуба в качестве аргументов

Также мы биндим класс **picked** элементу списка, который в данный момент выбран.

Кнопка сброса фильтра:

```
<div class="btn reset"
             tabindex="0"
      :class="{disabled: ( errorsCount > 0 && errorsCount <= 15 ) &&
!co.length}"
             @click="resetFilter"
             v-if="(co.length || !reseted">
                {{ this.resetBtnValue }}
            <div class="icn reset"></div>
</div>
```

Событие @click вызывает функцию сброса фильтра.

В директиве **v-if** проверяем выполнение условий, при которых кнопка монтируется в DOM (если произведен выбор элемента параметра фильтрации (т.е. выбран как минимум турнир) самим пользователем или фильтр не сброшен (т.к. либо страница отдана сервером по url, содержащему валидные параметры фильтрации, либо пользователь непосредственно сам уже отфильтровал контент)). Бинд на класс **disabled** срабатывает, если фильтр сброшен, но данные не получены в связи с ошибкой отправки/обработки

запроса (параметр **co** должен быть пустым, а количество вернувшихся с ошибкой запросов подряд больше 0 но не превышает лимитированные15).

Кнопка поиска:

```
<div class="btn filter"
             tabindex="1"
      :class="{disabled: fetching}"
             v-if="picked && co.length"
             @click="fetchData(false)">{{ this.searchBtnValue }}
</div>
```

В директиве **v-if** проверяем, что переменная **picked** возвращает true и фильтр активирован (выбран как минимум турнир).

Бинд на класс **disabled** срабатывает в момент, когда обрабатывается запрос, отправляемый методом fetchData.

Событие @click вызывает функцию **fetchData**, отправляющую запрос на получение контента, соответствующего выбранным параметрам. В качестве параметра функции передается false. Подробнее, для чего надо передавать этот параметр, написано в описании самой функции.

```
<span class="error" v-if="fetchClubsError || fetchPlayersError || errorsCount
> 0">
   <p v-if="fetchClubsError">{{ fetchClubsErrMsg }}</p>
   <p v-if="fetchPlayersError">{{ fetchPlayersErrMsg }}</p>
   <p v-if="errorsCount > 0 && errorsCount <= 15">{{ fetchDataErrMsg }}</p>
   <p v-if="errorsCount > 15">{{ exceededRequestsMsg }}</p>
</span>
```

Блок, в котором рендерятся оповещения о возникших ошибках.

Data

`fetching:`

Флаг состояния процесса загрузки данных с сервера, отвечающий за отображение прелоадера. По умолчание значение **false**. Значение **true** принимает в момент вызова метода **fetchData**.

`picked:`

Флаг, регулирующий видимость кнопки поиска. По умолчанию присваивается **false**. Кнопка видна, когда переменная принимает значение **true**. Происходит это в момент выбора варианта параметра фильтрации, отличающегося от того, по которому в данный момент отфильтрован контент. Проверка этого происходит в watcher'е метода computed properties **requestURL**.

`reseted:`

Флаг, регулирующий видимость кнопки сброса фильтра. По умолчанию принимает значение **true**, отменяя рендер кнопки в DOM. Значение **false** переменная получает в момент вызова метода **fetchData** либо же если страница отдана сервером по url, содержащему валидные параметры фильтрации, проверка чего осуществляется в методе **getSchedule** (который вызывается в хуке **created**) на основании значения параметра **isFiltered**. Если **isFiltered** возвращает **true**, то **reseted** принимает **false** и наоборот.

```
fetchClubsError: false,
fetchPlayersError: false
```

Флаги состояния ошибки, которая возвращается при неудачной отправке/обработке сервером запроса на получение списка клубов и списка игроков соответственно.

`errorsCount: 0`

Свойство, которое хранит текущее кол-во вернувших ошибку запросов контента подряд.

`competitionsList, clubsList, playersList:`

Три переменные, хранящие массивы турниров, клубов и игроков соответственно. Массивы двумерные и каждый элемент состоит из слага и названия турнира/клуба/игрока.

Массив соревнований компонент всегда получает от сервера при рендере страницы через параметр competitions (далее массив этот парсится и передается в переменную competitionsList в методе **getSchedule**).

Массивы клубов и игроков могут быть либо переданы таким же способом (если страница отдана сервером по url, содержащему валидные параметры фильтрации) либо получены аякс-запросом от сервера в методах **fetchClubsList** и **fetchPlayersList** соответственно.

```
loadingClubs, loadingPlayers:
```

Флаги, которые регулируют отображение прелоадеров в блоках со списками клубов и игроков. По умолчанию принимают значение **false**. Значение **true**, которое активирует рендер прелоадеров в DOM, они принимают во время ожидания ответа от сервера при аякс-запросе массивов клубов и игроков в методах **fetchClubsList** и **fetchPlayersList** соответственно.

```
pickedCoSlug: this.competitionSlug,
pickedCoName: this.competitionName;
pickedClubSlug: this.clubSlug,
pickedClubName: this.clubName;
pickedPlSlug: this.playerSlug,
pickedPlName: this.playerName;
```

Переменные, хранящие слаг и название выбранного в данный момент времени турнира / клуба / полное имя игрока. Изначально принимают значения указанных свойств из props.

Props

```
competitions: String
clubs: String
players: String
```

Массивы соревнований / клубов / игроков, которые передаются компоненту. Массив с перечнем соревнований всегда одинаков. Массивы же клубов и игроков по умолчанию имеют всего один стандартный элемент (со слагом ‘all’, по клику на который происходит сброс следующего блока). Но если страница отдана сервером по url-адресу, содержащему валидные параметры фильтрации, то в зависимости от значения параметров, на сервере формируется массив клубов / игроков и передается в компонент, где в дальнейшем парсится в методе **getSchedule**. Т.е. если в url есть параметр **co=epl**, то на сервере сформируется массив из клубов, которые в текущем сезоне принимают участие в розыгрыше соревнования, слагу которого соответствует значение параметра (в данном случае – АПЛ). По аналогии: если в url присутствует параметр **club=mun**, то сервер формирует массив из игроков, которые в текущем сезоне заявлены за клуб, слагом которого является **mun**. И небольшой нюанс: если страница отдана по url, содержащему только параметр **club** с валидным значением, то будет отдан массив из игроков, которые заявлены за этот клуб в этом сезоне. А если помимо параметра **club** есть и параметр **co** с валидным значением, то сервер сформирует массив из игроков, которые заявлены в текущем сезоне за указанный клуб в указанном соревновании. Более детально описано в разделе по организации серверной части.

```
competitionSlug: String,
competitionName: String,
clubSlug: String,
clubName: String,
playerSlug: String,
playerName: String
```

Слаги и названия (имена) турнира / клуба / игрока, по которым отфильтрована отданная сервером страница. Значения передаются в зависимости от значений параметров **co**, **club** и **pl** в url или их отсутствия. Подробнее о формировании этих значений в разделе по организации серверной части.

```
unpickedCompetition: String,
unpickedClub: String,
unpickedPlayer: String
```

Наименования для дефолтных вариантов списка, которые используются для присваивания их свойствам data pickedCoName, pickedClubName и pickedPlName соответственно в методах сброса блоков фильтра (resetClubsBlock, resetPlayersBlock и resetFilter).

Эти свойства нужны в целях поддержки компонентном мультиязычности.

```
apiPath: String
```

URL-адрес на страницу api, которая занимается генерацией динамического контента.

```
model: String
```

Название модели данных Laravel, для фильтрации контента которой используется данный экземпляр компонента.

```
container: String
```

id контейнера (элемента DOM), в который помещаются данные, получаемые в результате запроса в методе fetchData.

```
resetBtnValue: String,
searchBtnValue: String
```

Текст для кнопок сброса фильтра и поиска.

Данные свойства нужны в целях поддержки компонентном мультиязычности.

```
fetchClubsErrMsg: String,
fetchPlayersErrMsg: String,
fetchDataErrMsg: String,
```

```
exceededRequestsMsg: String
```

Свойства, передающие текст ошибок, сообщающий об их наличии пользователю.

```
isFiltered: String
```

Свойство, передающее true или false, в зависимости от того, отдана страница по url с валидными параметрами фильтрации или нет. Используется в методе **getSchedule**.

Computed

```
co() {
 return ( this.pickedCoSlug !='all' ? '?co='+this.pickedCoSlug : ' ' )
}
```

Функция, составляющая параметр **co** для url, на который отправляется аякс-запрос отфильтрованных данных в методе **fetchData**.

В случае, если пользователем выбран турнир, возвращающий слаг, отличный от 'all', то параметр принимает вид **'?co='+this.pickedCoSlug**. Если же выбран нулевой элемент списка, возвращающий слаг 'all', то функция возвращает пустой ответ, таким образом удаляя этот параметр из url-запроса.

По такому же принципу работают и функции, составляющие параметры **club** и **pl**.

```
requestUrl() {
        return ( this.apiPath + '/' + this.model + this.co + this.club
+ this.pl)
}
```

Функция сборки url-адреса, который используется в методе **fetchData** для отправки на него аякс-запроса.

Methods

```
getSchedule() {
                this.competitionsList = JSON.parse(this.competitions);
                this.clubsList = JSON.parse(this.clubs);
                this.playersList = JSON.parse(this.players);

                if (this.isFiltered) {
                    this.reseted = false;
                    this.$root.$emit( 'filtered', this.requestUrl,
this.reseted );
                }
}
```

Метод, который вызывается в хуке created. В нем парсятся получаемые от сервера через свойства props массивы со списком турниров, клубов и игроков. Так же, в случае, если страница отдана по url с валидными параметрами фильтрации (т.е. с отфильтрованным контентом), то флаг **reseted** принимает значение false (и кнопка сброса фильтра монтируется в DOM) и на корневой элемент, с которым связан данный компонент, имитируется пользовательское событие под именем filtered, которое прослушивается другими компонентами, связанными с этим элементом DOM (в данном случае – компонентном пагинации, импорт которого объявляется перед свойством data строчкой:)

```
import infinitePagination from '../components/infinitePaginate' )
```

Помимо имени, событие так же передает два параметра – url-адрес, по которому будет производится пагинация и значение свойства reseted, в зависимости от которого будет формироваться вид параметра **page**.

Далее разберем функции выбора турнира / клуба / игрока на примере одной из них, т.к. их архитектура в целом идентична и для каждого следующего блока упрощается.

```
pickCompetition(self, slug, name) {
                if(this.pickedCoSlug != slug ) {
                    this.pickedCoSlug = slug;
                    this.pickedCoName = name;
                    if ( slug != 'all' ) {
                        this.resetClubsBlock();
                        this.resetPlayersBlock();
                        this.fetchClubsList(slug);
                    } else {
                        this.resetFilter()
                    }
                } else {
                   self.stopPropagation()
}
```

Функция выбора соревнования, которая вызывается по клику на элемент списка блока соревнований. В качестве аргументом функция принимает нативное событие клика, слаг и название турнира соответственно.

Если переданный слаг равен тому, который до клика хранился в свойстве pickedCoSlug, то мы предотвращаем всплытие события строчкой self.stopPropagation(), что позволяет оставлять блок со списком открытым при клике на пункт, который соответствует уже выбранному турниру. Если же переданный слаг не равен значению, хранящемуся в свойстве **pickedCoSlug**, то мы их приравниваем и в свойство **pickedCoName** записываем переданный параметр name с именем выбранного турнира. Если передаваемый слаг не равен дефолтному 'all', то мы вызываем методы сброса блоков клубов и игроков и метод **fetchClubsList(slug)**, в который передаем полученный в качестве параметра слаг. Если же был передан слаг 'all', то вызывается метод resetFilter, сбрасывающий весь блок фильтрации целиком (это следует из логики работы фильтра, описанной в самом начале – в разборе template).

```
fetchClubsList(slug) {
                this.loadingClubs = true;
                this.$http.get( this.apiPath + '?getclubslist=' + slug
).then( response => {
                    this.loadingClubs = this.fetchClubsError = false;
                    this.clubsList = response.data
                }, error => {
                 console.log(error);
                 this.fetchClubsError = true;
                 this.loadingPlayers = this.fetchPlayersError = false

                 setTimeout( () => { this.clubsListRecursion(slug) },
3000 )
                })
}
```

Функция, отправляющая запрос на получение массива клубов, участвующих в выбранном турнире, слаг которого передается в функцию в качестве параметра.

Непосредственно перед запросом присваиваем флагу **loadingClubs** значение true, в следствие чего в div, содержащий список клубов, монтируется прелоадер, оповещающий о процессе загрузки.

Затем отправляется запрос на url страницы api с параметром **getclubslist**, значение которого равно передаваемому в функцию параметру **slug**.

В случае успешного выполнения запроса, свойствам **loadingClubs** и **fetchClubsError** присваиваются значение false, что влечет за собой демонтирование прелоадера и сообщения об ошибке, и в свойство data **clubsList** записываются полученные от сервера данные.

В случае возникновения ошибки при отправке или обработке запроса, тело ошибки выбрасывается в консоль и вызывается метод settimeout, рекурсивно вызывающий каждые 3 секунды эту же функцию при помощи метода

clubsListRecursion. Перед вызовом метода settimeout свойствам **fetchClubsError** и **fetchPlayersError** (на случай, если в момент смены выбранного турнира была зафиксирована ошибка получения списка игроков) присваиваются значения false, а свойству **fetchClubsError** – true.

Метод, с помощью которого рекурсивно вызывается предыдущий, выглядит следующим образом:

```
clubsListRecursion(slug) {
                if (this.pickedCoSlug != slug) {
                    return false
                } else {
                    return this.fetchClubsList(slug)
                }
            },
```

В нем присутствует та же проверка, что описана выше. Если значение реактивного свойства pickedCoSlug равно значению передаваемому параметром slug – рекурсивно (относительно контекста вызова) вызывается метод fetchClubsList. Если же они не равны – метод возвращает false, прекращая рекурсивные запросы, которые уже не соответствуют новому выбранному параметру фильтрации.

Функция, отправляющая запрос на получение массива игроков:

```
fetchPlayersList(slug) {
   this.loadingPlayers = true;
   this.$http.get(this.apiPath + '?getplayerslist=' +
slug).then(response => {
      this.loadingPlayers = this.fetchPlayersError = false;
      this.playersList = response.data
      }, error => {
```

```
        console.log(error);
        this.fetchPlayersError = true;
        setTimeout( () => { this.playersListRecursion(slug) }, 3000 )})
}
```

работает по тому же принципу, что и предыдущая. Это же касается и ее рекурсивного метода:

```
playersListRecursion(slug) {
                if (this.pickedClubSlug != slug) {
                    return false;
                } else {
                    return this.fetchPlayersList(slug)
                }
}
```

Функции сброса блоков со списками клубов и игроков соответственно:

```
resetClubsBlock() {
                this.pickedClubSlug =  'all';
                this.pickedClubName = this.unpickedClub;
}
```

```
resetPlayersBlock() {
                this.pickedPlSlug = 'all';
                this.pickedPlName = this.unpickedPlayer;
}
```

Свойствам data **pickedClubSlug** и **pickedPlSlug** присваивается значение нулевого элемента списка, а свойствам **pickedClubName** и **pickedPlName** присваиваются отданные сервером значения, зависящие от выбранной пользователем локализации.

```
resetFilter() {
                this.pickedCoName = this.unpickedCompetition;
                this.pickedCoSlug = 'all';
                this.resetClubsBlock();
                this.resetPlayersBlock();
                this.loadingClubs = this.loadingPlayers =
this.fetchClubsError = this.fetchPlayersError = false;
                this.errorsCOunt = 0;
                if ( !this.reseted) {
                    this.fetchData(true)
                }
}
```

Функция сброса фильтра целиком. Первые 2 строки идентичны контексту предыдущих двух методов, которые вызываются следом.

Затем присваивается значение false свойствам loadingClubs, loadingPlayers, fetchClubsError и fetchPlayersError (в случае, если фильтр сбрасывается в момент, когда запросы на получение списка клубов/игроков вернули ошибку) и обнуляем счетчик ошибок получения контента.

Если свойство data **reseted** в момент вызова функции имеет значение **false**, вызывается метод **fetchData**, отправляющий запрос на получение неотфильтрованного контента. Проверка эта нужна для того, чтобы в случае, если контент и так не отфильтрован, на сервер не отправлялся лишний ненужный запрос.

Функция, отправляющая запрос на получение контента и обрабатывающая ответ сервера:

```
fetchData(ifReseting) {
        this.fetching = true;
        if( this.errorsCount > 15 ) {
           this.errorsCount = 0
```

```
        };
        this.$http.get( this.requestUrl ).then( response => {
            this.reseted = ifReseting;
            var container = document.getElementById( this.container );
            container.innerHTML = '';
            container.insertAdjacentHTML( 'beforeEnd',
response.data.content );

            this.fetching = this.picked = false;
            this.errorsCount = 0;
             window.history.replaceState(null, null, '/' + this.model +
this.co + this.club + this.pl);
            var lastpage = response.data.lastpage;
            this.$root.$emit( 'filtered', this.requestUrl,
this.reseted, lastpage );
             }, error => {
            console.log( error );
            let url = this.requestUrl;
            this.picked = true;
            this.errorsCount++;
            setTimeout( () => { this.fetchDataRecursion(ifReseting,
url) }, 2000 )
             })
}
```

В начале устанавливаем свойству fetching значение true. Затем, если количество вернувших ошибку запросов превысило 15 – обнуляем их (т.к. предполагается, что в этой ситуации пользователь в ручную повторил попытку запроса).

После этого отправляем get-запрос на адрес, который рассчитан в функции **requestUrl**. Если запрос выполнен и обработан сервером удачно:

- Сперва присваиваем значение свойству **reseted**.

```
this.reseted = ifReseting;
```

Для этого во время вызова метода и передавались в качестве параметров true / false. Если в качестве параметра был передан false, значит метод вызван с целью получения отфильтрованного по выбранным пользователем параметрам контента. Если же был передан true, значит метод вызван в контексте сброса фильтра.

- Затем записываем в переменную container элемент DOM, в котором содержится контент, с которым взаимодействует данный компонент. После этого мы этот элемент очищаем и вставляем полученные от сервера данные:

```
var container = document.getElementById(this.container);
container.innerHTML = '';
container.insertAdjacentHTML( 'beforeEnd', response.data.content
);
```

- Помимо этого, устанавливаем свойствам **fetching**, и **picked** в качестве значения false, обнуляем счетчик ошибок и меняем url в адресной строке браузера, добавляя к нему get-параметры, по которым отфильтрован контент:

```
this.fetching = this.picked = false;
this.errorsCpunt = 0;
window.history.replaceState(null, null, '/' + this.model +
this.co +    this.club + this.pl)
```

- Ну и в конце мы записываем в переменную **lastpage** номер последней страницы пагинации полученного в результате запроса содержимого и имитируем пользовательское событие 'filtered', с которым в качестве параметров для прослушивающего компонента передаем свойства requestUrl и reseted, а также переменную lastpage:

```
var lastpage = response.data.lastpage;
```

```
this.$root.$emit( 'filtered', this.requestUrl, this.reseted,
lastpage );
```

Если в результате отправки или обработки заброса возникла ошибка, то выбрасываем в консоль тело ошибки, устанавливаем свойству

rror значение true. Следом записываем в переменную url значение свойства requestUrl в данный момент и инкриминируем значение свойства errorsCount. Затем по аналогии с методам fetchClubsList и fetchPlayersList вызываем функцию рекурсивно каждые 2 секунды при помощи метода fetchDataRecursion, которому в качестве параметров передается полученное самой функцией значение ifReseting и значение переменной url.

```
console.log( error );
this.picked = true;
let url = this.requestUrl;
this.errorsCount++;
setTimeout( () => { this.fetchDataRecursion(ifReseting, url) }, 2000 )
```

Внутри метода fetchDataRecursion описана проверка, в результате которой либо вызывается рекурсивно (в контексте обращения к данному методу) функция fetchData либо возвращается false. Второй вариант выполняется, когда значение переданного параметра url не равно значению реактивного свойства requestUrl (Т.е. в промежутке между предыдущим и текущим вызовом метода значение requestUrl изменилось (изменилось значение как минимум одного параметра фильтрации либо же фильтр был сброшен)) либо запрос вернул ошибку 15 раз подряд. В этом случае мы обрываем рекурсию, устанавливаем свойству fetching значения false и обнуляем счетчик ошибок (только в случае изменения пользователем параметров фильтрации).

```
fetchDataRecursion(ifReseting, url) {
```

```
                if(url != this.requestUrl ) {
                    this.fetching = false;
                    this.errorsCount = 0;
                    return false
                } else if(this.errorsCount > 15) {
                    this.fetching = false;
                    return false
                } else {
                    return this.fetchData(ifReseting);
                }
}
```

Функция, управляющая классом 'open' у блоков фильтрации:

```
isOpened(e) {
                var self = e.target;
                var handleClickOutside = (e) => {
                    if( self != e.target &&
self.classList.contains('open') ) {
                        self.classList.remove('open');
                        document.removeEventListener('click',
handleClickOutside)
                    }
                }

                if( !self.classList.contains('disabled') ) {
                    self.classList.toggle('open');
                    document.addEventListener('click',
handleClickOutside)
                }
}
```

В качестве параметра функция принимает нативное событие click.

Далее в переменную **self** записывается ссылка на элемент DOM, который вызвал событие. Затем в переменную **handleClickOutside** записываем функцию, которую затем вызываем при регистрации обработчика события click. В качестве параметра мы передаем в эту функцию уже другое событие click, которое непосредственно прослушивается зарегистрированным обработчиком.

Если ссылка на элемент DOM, которая записана в **self**, не равна ссылке на элемент DOM, который вызвал переданное в функцию событие **e**, то мы удаляем класс 'open' у элемента, на который ссылается **self** и удаляем зарегистрированный обработчик события:

```
if( self != e.target ) {
    self.classList.remove('open');
    document.removeEventListener('click', handleClickOutside)
}
```

Суть этой функции в том, чтобы открытый блок со списком закрывался по клику вне его самого.

Ниже объявлена функция, которая 'открывает' блок, на который ссылается **self** и регистрирует обработчик события click, принимающий в качестве аргумента функцию, записанную в переменную **handleClickOutside**. Выполняется она в случае, если у элемента, на который ссылается **self**, отсутствует класс 'disabled'. Эта проверка необходима для того, чтобы не плодить обработчики кликом на заблокированные блоки.

Watchers

```
requestUrl: function(newVal, oldVal) {
     this.picked = ( this.co !=' ' && newVal != ( this.apiPath + '/' + this.model + window.location.search ) ) ? true : false;
}
```

Данный watcher следит за изменением ответа, который возвращает функция **requestUrl**. Так как эта функция из данных типа computed, то она автоматически реагирует на изменение параметров, от которых зависит возвращаемый ею ответ. Соответственно, как только мы выбираем турнир, клуб или игрока, отличные от тех, по которым в данный момент отфильтрован контент – в свойство **picked** записывается значение true, что делает кнопку поиска видимой (монтирует ее в DOM). А если в результате взаимодействия с компонентом пользователь возвращается к тем вариантам значений параметров, по которым отфильтрован контент в данный момент времени – свойство получает значение false, демонтируя кнопку поиска из DOM страницы. Это сделано для того, чтобы ограничить количество бесполезных запросов на получение данных, которые и так отображены на странице.

```
fetching: function() {
 document.getElementById( this.container ).classList.toggle('loading')
}
```

Watcher следит за значением свойства fetching в случае изменения которого управляет классом loading элемента DOM, в котором содержится контент.

Server-side declaration

Рендер компонента вызывается добавлением в шаблон экземпляра компонента <three-stage-content-filter></three-stage-content-filter>

```
<three-stage-content-filter api-path="{{ URL::route('api') }}"
                model="(название модели, с содержимым которого
взаимодействует компонент)"

                container="content
                reset-btn-value="{{ $resetBtnValue }}"
                search-btn-value="{{ $searchBtnValue }}"
                competitions="{{ $competitions }}"
                clubs="{{ $clubsList }}"
                players="{{ $playersList }}"
                competition-slug="{{ $competitionSlug }}"
                competition-name="{{ $competitionName }}"
                club-slug="{{ $clubSlug }}"
                club-name="{{ $clubName }}"
                player-slug="{{ $playerSlug }}"
                player-name="{{ $playerName }}"
                unpicked-competition="{{ $unPickedCompetitionValue }}"
                unpicked-club="{{ $unpickedClubValue }}"
                unpicked-player="{{ $unpickedPlayerValue }}"

                fetch-clubs-err-msg="{{ $fetchClubsErrMsg }}"
                fetch-players-err-msg="{{ $fetchPlayersErrMsg }}"
                fetch-data-err-msg="{{ $fetchDataErrMsg }}"

                exceeded-requests-msg="{{ $exceededRequestsMsg }}"
                is-filtered="{{ $isFiltered }}">
</three-stage-content-filter>
```

Все аттрибуты экземпляра соответствуют свойствам props. Те, которые написаны в позвоночном-регистре, соответствуют одноименным свойствам props в верблюжемРегистре.

Значение аттрибута **api-path** автоматически возвращает ссылку на страницу api, основываясь на описание роутинга под названием 'api' в файле App/routes/web.php.

Значением аттрибута **model**, как указано выше, является название модели, с содержимым которое взаимодействует компонент. **Container** – id элемента

DOM, который содержит в себе изначальные данные и куда помещаются уже отфильтрованные.

Значения всех остальных аттрибутов являются переменными, которые при формировании страницы на сервере предоставляются композером threeStageFilterComposer, который вызывается при компоновке определенных представлений Laravel, перечень которых объявляется в методе **composeThreeStageFilter** сервис-провайдера ViewComposerServiceProvider.