

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
(ДИПЛОМНАЯ РАБОТА)

на тему «Разработка отраслевой CRM – системы на примере абстрактного  
малого металлообрабатывающего предприятия.»

студент Копытов В.Г.

специальность Frontend – программист

## **Оглавление.**

1 Введение	3
2 Анализ предметной области.	
2.1 Общая характеристика предприятия и его подразделений	4
2.2 Постановка задачи.	
3 Реализация проекта	
3.1. Пользовательский интерфейс	6
3.2. UML – диаграмма	9
3.3. Код и описание	9
3.3.1 Настройка Vue	9
3.3.2 Декомпозиция	9
3.3.3 Пошаговое исполнение	10
3.4. Реализация	
3.5. Перспективы и доработки	
4. Заключение	

## ВВЕДЕНИЕ

Стремительное развитие и интеграция цифровых технологий во все аспекты повседневной жизни оказало огромное влияние на экономическую составляющую. Сбор и анализ информации, составление отчетности, взаимодействие между производителями, потребителями, контролирующими органами, банками, компонентами финансовой системы, планирование, маркетинг – сложно сейчас представить без использования информационных технологий. Возможности, которые предоставляет программное обеспечение, становятся не просто выгодным отличием для участников рынка, но и зачастую, обязательным условием для того, чтобы оставаться конкурентоспособным, успешным, прибыльным.

Особое место среди информационных продуктов занимают CRM – системы. Они являются одним из основных инструментов для бизнеса, поскольку любой бизнес держится на «трех столпах»: контроль, аналитика, прогнозирование. Крупные и крупнейшие участники рынка давно пользуются благами CRM – систем и постоянно работают над их улучшением.

В среде малого бизнеса и небольших предприятий всё не так однозначно. Зачастую программное обеспечение, применяющееся на предприятии, ограничивается только 1С и планировщиком задач у руководителя, в лучшем случае. Производственные процессы при этом контролируются либо посредством бумажных носителей, либо специально нанятым персоналом в лице мастеров, менеджеров, прочих специально нанятых работников. При этом процент неучтенных потерь на различных этапах производства, может складываться в ощутимые убытки с течением времени.

Причины отказа от CRM могут быть различны: от неосведомленности, до нежелания вкладываться в разработку ПО под свои нужды и специфику. В данной работе, на примере условно – абстрактного металлообрабатывающего предприятия, мы разберемся в этапах производства, и разработаем простую CRM – систему, которая потенциально может существенно помочь в контроле и учете производственного процесса.

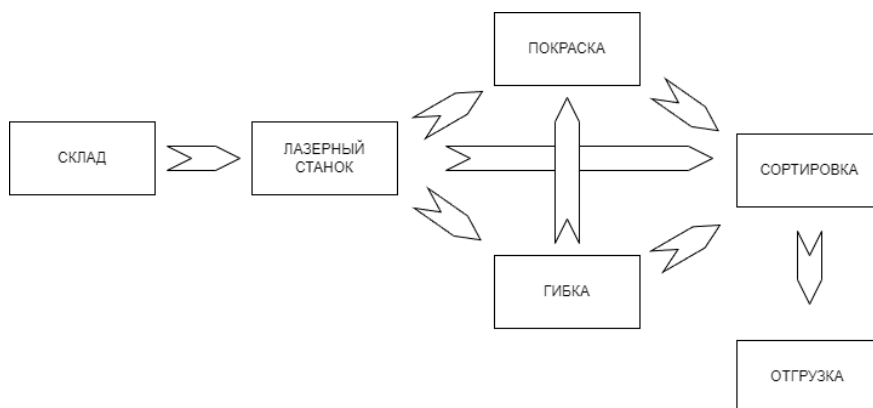
## Анализ предметной области.

### 2.1 Общая характеристика предприятия и его подразделений

Как правило, крупные компании, разрабатывая ПО (или заказывая разработку), стараются учесть все возможные этапы производства и наделить CRM максимальной функциональностью. Действительно, удобнее иметь один продукт, представляющий собой корпоративную экосистему, в которой гармонично существует вся информация от стадии формирования заказа до стадии отгрузки готовой продукции. Однако, разработка такого продукта требует большого штата программистов и соответствующие временные затраты. В данной работе, мы рассмотрим только один возможный программный модуль, отражающий, в нашем случае, конкретно производственный процесс. Для этого конкретно сформируем условия и порядок рассматриваемого производства.

Рассматриваем среднее металлообрабатывающее предприятие (далее П.), которое занимается изготовлением заготовок из листового металла посредством станков лазерной резки на заказ и производит некоторую обработку в виде гибки и покраски некоторых произведенных деталей (в зависимости от заказа). Производственная цепочка выглядит следующим образом:

1. П. закупает листовой металл, который приходит на склад, где сортируется по толщине.
2. Со склада необходимые листы металла передается на лазерный станок.
3. После прохождения этапа «резка» на лазерном станке, детали могут быть переданы на гибку, либо же сразу на покраску или в отгрузку.
4. После гибки детали также могут быть переданы на покраску или в отгрузку.
5. После покраски детали передаются на отгрузку.
6. Отгрузка готовых деталей клиенту.



## 2.2 Постановка задачи

В данной работе нас интересует участок производственной цепи, связанный конкретно с механической обработкой деталей, без складского учета, работы с контрагентами, и прочими дополнительными модулями, которые, могут быть в последствии интегрированы в нашу CRM. Поэтому, опираясь на этапы производства, мы можем четко сформировать интересующие нас задачи для программного продукта:

- Администратор/технолог должен иметь возможность добавления задач на доску заданий;
- Доска заданий формируется для каждого этапа механической обработки отдельно в соответствии с этапом производства;
- Всего существует 4 этапа механической обработки: 1- «Резка», 2-«Гибка», 3 –«Покраска», 4 – «Отгрузка»;
- Добавленная задача может проходить не все этапы обработки, т.е., например, может быть направлена на 1 и 4 этап без включения в маршрут этапов 2 и 3;
- При формировании задачи, должны быть сформированы значения:
  - ID задачи (генерируется случайным образом)
  - Описание задачи (размеры листа металла, толщина металла)
  - Количество листов/деталей
  - Время в работе (когда задача была запущена в работу и завершена)
  - Чертеж деталей (загружается в задачу технологом для последующего открытия чертежа на ПО станков)
- Администратор /Технолог должен иметь возможность просмотра выполненных задач;
- Выполняющие задачу работники на каждом этапе получают свои задачи на доске задач, где отображаются все необходимые для исполнения параметры;
- После выполнения на одном этапе, задача автоматически переходит на другой;
- После этапа «Отгрузка», задача сохраняется в отдельное хранилище;

На основании поставленных задач, приступим к выполнению.

## Реализация проекта

### 3.1 Пользовательский интерфейс

Я начал с проектирования пользовательского интерфейса, для того, чтобы в процессе понять возможные нюансы и сложности в разработке приложения. В моём представлении, пользовательский интерфейс должен включать в себя:

- Окно авторизации пользователя
- Кабинет администратора
- Окно добавления задач
- Кабинет исполнителя

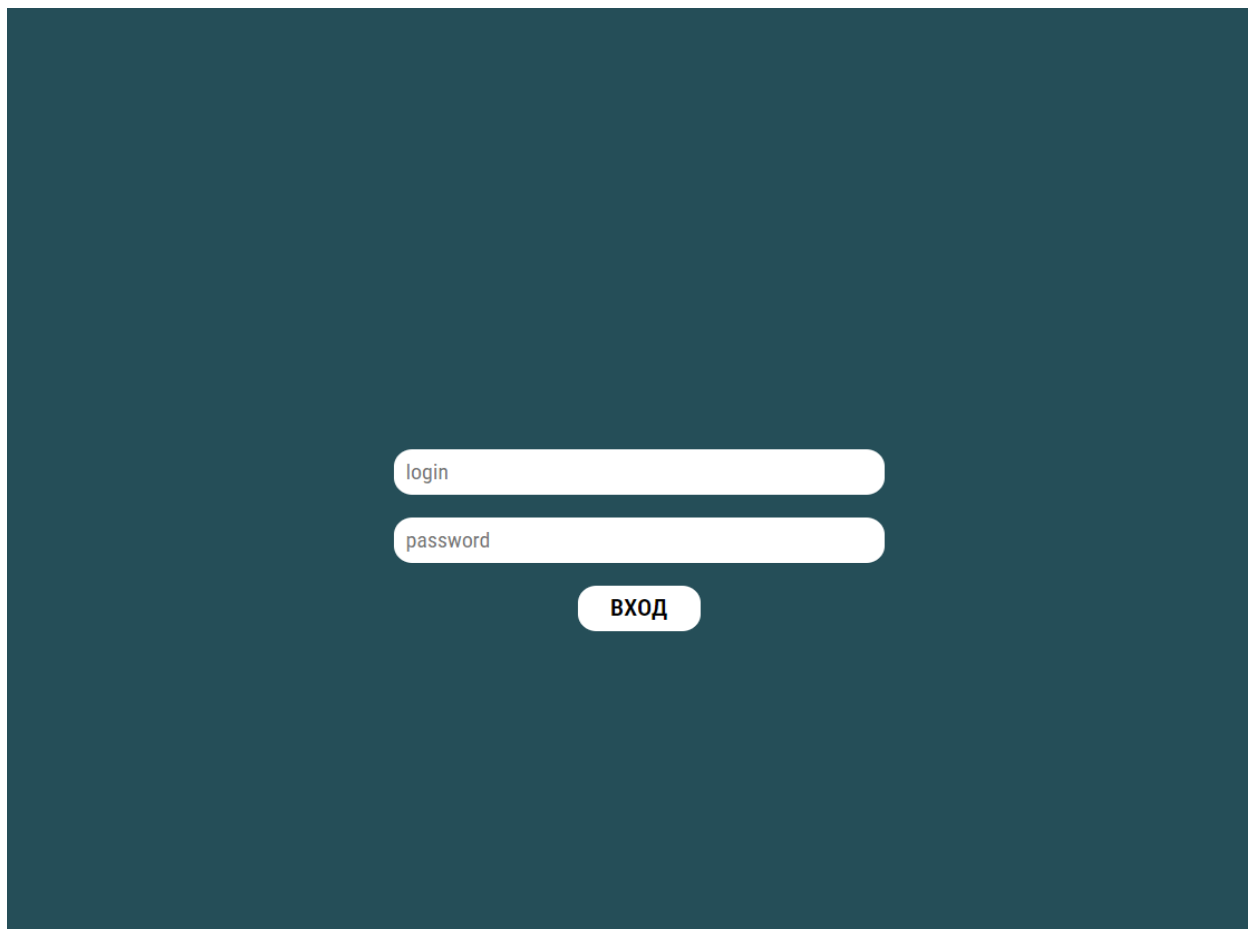
В соответствии с описанными выше задачами, этого достаточно.

Для пользовательского интерфейса на сайте <https://visme.co/> была выбрана схема номер 22, которая использовалась в дальнейшем для оформления приложения.

#### 22 Корпоративный и традиционный

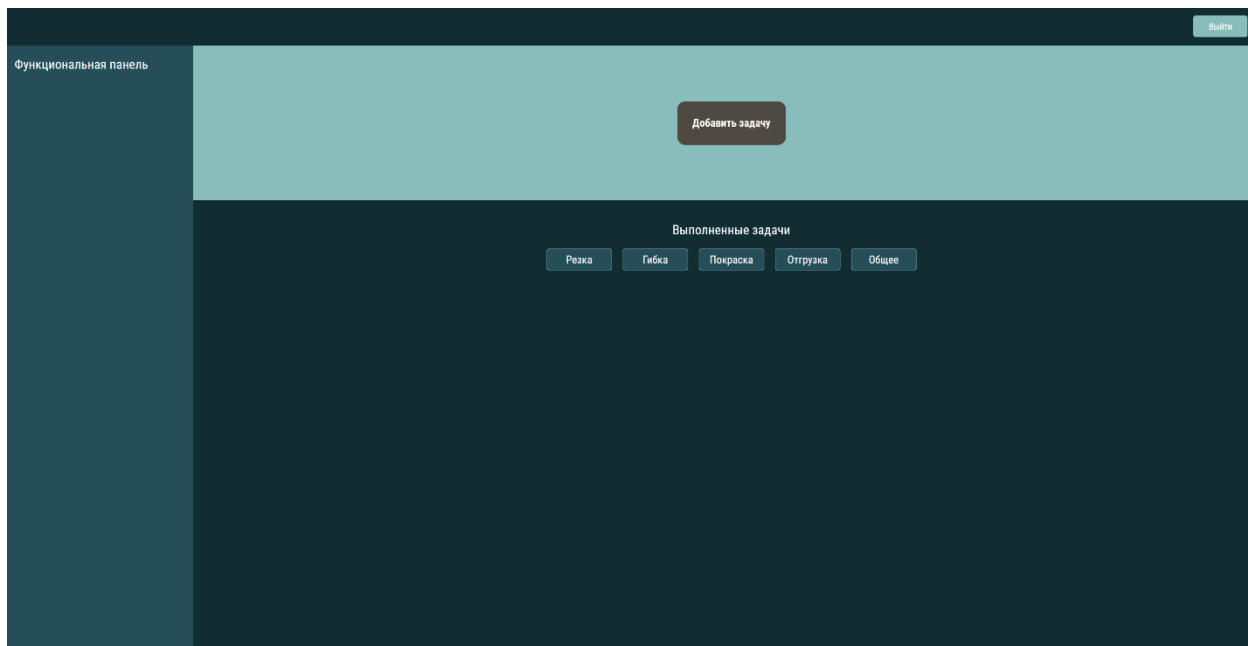


Окно авторизации:



The image shows a dark teal background with a white login form in the center. The form consists of two input fields: the top one is labeled 'login' and the bottom one is labeled 'password'. Below these fields is a white button with the text 'ВХОД' (Login) in black. The entire form is centered horizontally and vertically on the background.

Кабинет администратора:



The image shows a dark teal dashboard layout. On the left is a vertical sidebar labeled 'Функциональная панель'. The main content area has a light teal header bar with a 'Выйти' (Logout) button in the top right corner. Below the header bar is a dark teal section containing a 'Добавить задачу' (Add task) button. Further down is a section titled 'Выполненные задачи' (Completed tasks) which contains five sub-buttons: 'Разка', 'Гибка', 'Покраска', 'Отгрузка', and 'Общее'. The bottom part of the dashboard is a large dark teal area.

Окно добавления задач:

Добавить новую задачу

description

0

Резка

Гибка

Покраска

Отгрузка

Запустить в работу

Заккрыть окно

Кабинет исполнителя:

Выйти

Задачи на стадию: РЕЗКА

Количество: 0

Описание:

Поступило в работу: Технолог:21.3.2024 17:56:0

ID: 2151955e-ecfb-42e7-ac43-b73644a78a95

Скачать файл

Начать задачу

Завершить

Количество: 3

Описание: Test

Поступило в работу: Технолог:21.3.2024 21:15:12

ID: eab4a656-e6c4-4208-8b02-cd5b7ada3fff

Скачать файл

Начать задачу

Завершить



## 3.2 UML-Диаграмма

Перед составлением UML – диаграммы, необходимо сказать об архитектуре приложения. Поскольку данная работа выполняется с целью демонстрации приобретенных навыков во Frontend – разработке, всю логику приложения было решено перенести в хранилище Vue. В нашем случае store и локальное хранилище браузера Local Storage будут выполнять роль стороны сервера, где и будет развернута основная логика. Предоставить полноценное клиент-серверное приложение с учетом всех особенностей и функциональности не представляется возможным ввиду ограниченных ресурсов времени и количества разработчиков. Однако, на представленной ниже диаграмме я обозначил логику полноценного клиент – серверного приложения.

Здесь должна быть UML диаграмма!!!

## 3.3. Код и описание

В данном разделе я постараюсь как можно детальнее описать процесс написания кода и проблемы, с которыми сталкивался в процессе его написания. Проект будет написан в редакторе кода Visual Studio Code с использованием фреймворка Vue.

### 3.3.1 Настройка Vue

Перед началом работы, необходимо настроить рабочее окружение, а именно фреймворк Vue. Для этого введем несколько команд в терминале:

**npm instal vuex@3** – подключение библиотеки для управления состоянием, обеспечивает централизованное хранилище.

**Vue add router** - официальное решение для маршрутизации на стороне клиента. Нужно, чтобы связать URL браузера с контентом, отображаемым пользователю.

### 3.3.2 Декомпозиция

После осознания задач, я пришел к выводу, что необходимо провести декомпозицию, то есть, разделение задач на более простые и понятные составляющие компоненты, в соответствии с которыми становятся понятными дальнейшие шаги в написании кода. После декомпозиции пошаговый план действий и пунктов, необходимых к реализации представлял из себя следующее:

1. Создание карточки задачи, описание необходимых свойств

2. Создание полей ввода, необходимых для заполнения карточки задания
3. Написание метода для изменения карточки задания
4. Создание «хранилища», в которое будут передаваться измененные задачи
5. Написание методов для сохранения и извлечения данных в Local Storage
6. Написание методов (геттеров) для передачи задач из хранилища
7. Создание компонента модального окна
8. Создание страницы администратора
9. Создание компоненты для отображения задач на странице исполнителя
10. Создание страницы исполнителя
11. Написание метода для изменения актуальной стадии исполнения задачи
12. Написание метода для отметки времени начала и завершения задачи
13. Написание метода для передачи актуальных задач соответствующему исполнителю
14. Написание методов сохранения выполненных задач
15. Написание метода для корректного завершения задач
16. Создание простой авторизации
17. Добавление стилей к проекту
18. Тестирование и исправление ошибок приложения.

### 3.3.3 Пошаговое исполнение

#### *1. Создание карточки задачи, описание необходимых свойств*

В произвольной компоненте в `data` я создал объект, а также свойства, теоретически необходимые для изменения объекта:

```
data() {  
  return {  
    openModWin: false,  
    id: 0,  
    description: '',  
    count: 0,  
    productionStages: [],  
    timeInWork:[],  
    file: [],  
    ObjectX: {  
      id: 0,  
      description: '',  
      count: 0,  
      productionStages: [],  
      timeInWork:[],
```

```

    currentStage: 0,
    file: [],
    comments:[],
    date:'',
  },

```

**Id:** поле для присвоения уникального идентификатора

**description:** поле для описание задачи, по умолчанию имеет строковый формат

**count:** количество повторений задачи

**productionStages:** массив для сохранения стадий обработки

**timeInWork:** массив для сохранения времени начала и окончания обработки

**currentStage:** стадия обработки, на которой сейчас находится задача

**file:** массив для сохранения файла задачи

**comments:** предполагался как массив для сохранения комментариев, созданных в процессе обработки

**date:** дата создания задачи

## 2. Создание полей ввода, необходимых для заполнения карточки задания

```

<div class="addTaskWindow">
  <input class="addTaskWindow__input" type="text" v-
model="description" placeholder="description">
  <input class="addTaskWindow__input" type="number" v-
model="count" placeholder="count">
  <div class="addTaskWindow__wrapperCheck">
    <div class="addTaskWindow__wrapperCheck__label">
      <label for="1">Резка</label>
      <label for="2">Гибка</label>
      <label for="3">Покраска</label>
      <label for="4">Отгрузка</label>
    </div>
    <div class="addTaskWindow__wrapperCheck__input">
      <input class = "addTaskWindow__wrapperCheck__checkbox"
type="checkbox" value="1" id="1" v-model="productionStages"/>
      <input class = "addTaskWindow__wrapperCheck__checkbox"
type="checkbox" value="2" id="2" v-model="productionStages"/>

```

```

        <input class = "addTaskWindow__wrapperCheck__checkbox"
type="checkbox" value="3" id="3" v-model="productionStages"/>
        <input class =
"addTaskWindow__wrapperCheck__checkbox" type="checkbox" value="4"
id="4" v-model="productionStages"/>
    </div>
</div>
</div>

```

Добавление этапов обработки задачи было решено реализовать с помощью «checkbox». Очень помогла особенность VUE, которая позволяет присвоить значение чекбоксу и с помощью v-model, в случае, если чекбокс выбран, добавить это значение в массив, чем я и воспользовался.

### 3. Написание метода для изменения карточки задания

```

SetObject(ObjectX) {      //изменение объекта
    ObjectX.id = self.crypto.randomUUID();
    ObjectX.description = this.description;
    ObjectX.count = this.count;
    ObjectX.timeInWork.splice(0, ObjectX.timeInWork.length);
    ObjectX.timeInWork.push('Технолог:' + new
Date().getDate().toString() + '.' + new
Date().getMonth().toString() + '.' + new Date().getFullYear().toString() + '
' + new Date().getHours().toString() + ':' + new
Date().getMinutes().toString() + ':' + new
Date().getSeconds().toString());
    ObjectX.productionStages = this.productionStages.sort();
    ObjectX.currentStage = this.productionStages[0]; // TODO все
поля провалидировать!!!
    ObjectX.date = new Date().getDate().toString() + '.' + new
Date().getMonth().toString() + '.' + new Date().getFullYear().toString();
},

```

При написании метода изменения карточки задания, я использовал следующее:

1. ObjectX.id = self.crypto.randomUUID(); – встроенный метод, генерирующий случайный ID
2. ObjectX.timeInWork.splice(0, ObjectX.timeInWork.length); - обнуление массива, для того, чтобы не допустить добавление присвоенного раньше времени.

3. `ObjectX.timeInWork.push...` - в массив добавляется время создания задачи в полном временном формате.
4. `ObjectX.productionStages = this.productionStages.sort();` - необходимость в сортировке массива с маршрутом задачи обуславливается тем, что выбранные в произвольном порядке чекбоксы, заносятся в массив именно в этом порядке. Нам же важна строгая очередность.
5. `ObjectX.currentStage = this.productionStages[0];` - здесь мы присваиваем актуальную стадию обработки, опираясь на данные, полученные из ранее созданного массива.
6. `ObjectX.date = new Date().getDate().toString()+...` - я решил создать отдельное свойство объекта с датой для дальнейшей удобной сортировки, например для просмотра созданных ранее задач

Остальные свойства мы берем из полей ввода, которые «моделируются» в `date`

#### 4. Создание «хранилища», в которое будут передаваться измененные задачи

Поскольку хранилище и методы, которые должны располагаться на стороне сервера, мы условились расположить в `store`, дальнейшая работа разворачивается в `index.js` в `store` в папке проекта.

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);

const store = new Vuex.Store({
  state: {
    arrayTask:[
    ],
    arrayCuttingComplete:[],
    arrayBendComplete:[],
    arrayPeintingComplete:[],
    arrayTaskCompleted:[],
    tempArrayTaskCompleted: [],
```

Назначение созданных массивов сверху – вниз:

`arrayTask` – основной массив с задачами

`arrayCuttingComplete` – массив для сохранения выполненных задач после этапа резки

arrayBendComplete - массив для сохранения выполненных задач после этапа гибки

arrayPeintingComplete - массив для сохранения выполненных задач после этапа покраски

arrayTaskCompleted:[] - массив для сохранения выполненных задач

tempArrayTaskCompleted: - массив для отсортированных массивов выполненных задач. Нужен для нескольких вариантов отображения выполненных задач на странице администратора, либо других страницах.

## 5. *Написание методов для сохранения и извлечения данных в Local Storage*

```
setArrayTask(state, Object) {  
    let Clone = {};  
    for (let key in Object) {  
        Clone[key] = Object[key];  
    }  
    state.arrayTask.push(Clone);  
},  
saveInLocalStorage(state) {  
    localStorage.setItem("arrayTask",  
JSON.stringify(state.arrayTask));  
},  
loadFromLocalStorage(state) {  
    state.arrayTask =  
JSON.parse(localStorage.getItem("arrayTask") || "[]");  
},  
loadArrayTaskCompleted(state) {  
    state.arrayTaskCompleted =  
JSON.parse(localStorage.getItem("arrayTaskCompleted") || "[]");  
},
```

Во время тестирования приложения, была выявлена ошибка, при которой новые измененные задачи перезаписывали ранее сохраненные. Это происходило потому что массив, в который происходила запись и записываемый объект ссылались на одно и то же. Чтобы это исправить, я реализовал метод setArrayTask(). На изображении выше указаны не все методы для загрузки и сохранения массивов, поскольку они все однотипны.

## 6. Написание методов (геттеров) для передачи задач из хранилища

```
getters : {  
  getStageTasks: (state) => (stageNum) => {  
    return state.arrayTask.filter((item) => item.currentStage  
=== stageNum);  
  },  
  getStageTasksCompleted: state => {  
    return state.arrayTaskCompleted;  
  },  
  getTempArray: state => {  
    return state.tempArrayTaskCompleted;  
  },  
}
```

Из всех представленных выше геттеров, наибольший интерес представляет самый первый, который принимает в себя аргумент stageNum. Этот геттер отправляет на отрисовку отсортированный массив с задачами. Сортировка происходит в нём по актуальному этапу обработки. Пользователь, авторизуясь в приложении попадает в свой кабинет со своей доской задач. С этой страницы приходит stageNum.

## 7. Создание компонента модального окна

```
<template>  
  <div class="modalWindow">  
    <div class="modalWindow__header">  
      <span>{{ windowTitle }}</span>  
      <span>  
          
      </span>  
    </div>  
    <div class="modalWindow__content">  
      <slot></slot>  
    </div>  
    <div class="modalWindow__footer">  
      <button class="modalWindow__footer__leftBtn" @click  
="randomEvent">{{ leftBtnTitle }}</button>  
      <button class="modalWindow__footer__closeBtn"  
@click="closeModal">Заккрыть окно</button>  
    </div>  
  </div>  
</template>
```

```
    </div>
  </div>
</template>

<script>
export default {
  name: 'ModalWindowComponent',
  data() {
    return {

    };
  },
  props: {
    leftBtnTitle: {
      type: String,
      default: 'Кнопка по умолчанию',
    },
    windowTitle: {
      type: String,
      default: 'Заголовок по умолчанию',
    }
  },
  methods: {
    closeModal() {
      this.$emit('closeModal');
    },
    randomEvent() {
      this.$emit('randomEvent');
    }
  }
};
```