



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Казанский национальный исследовательский технологический
университет»**

(ФГБОУ ВО «КНИТУ»)

Кафедра «Интеллектуальных систем и управления информационными
ресурсами»

Направление «02.03.03 – Математическое обеспечение и администрирование
информационных систем»

Профиль «Информационные системы и базы данных»

Группа 4311-21

КУРСОВОЙ ПРОЕКТ

по дисциплине «Разработка пользовательского интерфейса»
на тему «Разработка клиентской части для инструмента анализа покрытия
автоматизированными тестами»

Исполнитель	_____	Козлов Илья Валерьевич
	(дата, подпись)	(Ф.И.О.)
Руководитель	_____	д-т кафедры ИСУИР, к.ф.-м.н.
	(дата, подпись)	Мангушева А.Р.
Нормоконтролер	_____	должность, (Ф.И.О.)
	(дата, подпись)	_____
		должность, (Ф.И.О.)

Проект защищен с оценкой _____

Руководитель _____

Казань, 2024

СОДЕРЖАНИЕ

ЗАДАНИЕ	3
ЛИСТ НОРМОКОНТРОЛЕРА	4
ВВЕДЕНИЕ.....	5
1 Анализ требований, выдвигаемых к продукту	7
2 Разработка макета веб-приложения.....	10
3 Верстка сайта по макету	14
4 Внедрение интерактивности	20
5. Связь функционала с серверной частью	25
ЗАКЛЮЧЕНИЕ	28
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	29
ПРИЛОЖЕНИЕ А	30

ЗАДАНИЕ

на курсовой проект студенту кафедры «Интеллектуальных систем и управления информационными ресурсами»

Тема проекта: «Разработка клиентской части для инструмента анализа покрытия автоматизированными тестами»

Исходные данные к проекту: _____

Содержание расчетно-пояснительной записки (включая перечень подлежащих разработке вопросов, включая вопросы стандартизации и контроля качества):

ВВЕДЕНИЕ

1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

2

Перечень графического материала (схемной документации) рисунки, таблицы

Консультанты по проекту (с указанием относящихся к ним разделов) _

Дата выдачи задания «_____» _____ 20__ г.

Руководитель проекта _____ (Мангушева А.Р.)

ЛИСТ НОРМОКОНТРОЛЕРА

1. Лист является обязательным приложением к пояснительной записке курсового проекта.

2. Нормоконтролер имеет право возвращать документацию без рассмотрения в случаях:

- нарушения установленной комплектности,
- отсутствия обязательных подписей,
- нечеткого выполнения текстового и графического материала.

3. Устранение ошибок, указанных нормоконтролером, обязательно.

Перечень

замечаний и предложений нормоконтролера по курсовому проекту, студента

(группа, инициалы, фамилия)

Л(Лист (страница))	Условное обозначение (код ошибок)	Содержание замечаний и предложений со ссылкой на нормативный документ, стандарт или типовую документацию

Дата «___» _____ 20__ г.

Нормоконтролер _____
(подпись) (инициалы, фамилия)

ВВЕДЕНИЕ

В современном информационном обществе, где технологические инновации становятся неотъемлемой частью повседневной жизни, понижается порог входа в профессии, связанные с информационными технологиями (ИТ). В связи с этим возрастает потребность в простых, удобных и функциональных интерфейсах не только со стороны конечного пользователя, но и со стороны команды, разрабатывающей продукт. К примеру, чтобы претендовать на должность стажера по ручному тестированию, достаточно пройти курсы продолжительностью в 1 месяц. Специалисту с подобным уровнем подготовки необходим наглядный и понятный интерфейс, отображающий статистику и позволяющий производить верхнеуровневый анализ покрытия автоматизированным тестами.

Создание веб-интерфейса, адаптированного к уникальным требованиям узконаправленных специалистов, может не только обеспечить удобство работы с данными о покрытии тестами, но также повысить производительность команды, снизить количество ошибок при интерпретации данных и упростить процесс принятия решений.

Целью данного курсового проекта является создание веб-интерфейса для инструмента, специально адаптированного под анализ покрытия автоматизированными тестами, который будет интуитивно понятен и функционален.

Для достижения поставленной цели выдвигаются следующие задачи:

1. Сформулировать требования, выдвигаемые к продукту.
2. Разработать макет клиентской части, отвечающий сформулированным функциональным требованиям.
3. Создать визуальную структуру сайта, используя простые языки разметки.

4. Внедрить интерактивность с жестко прописанными в программном коде данными.
5. Добавить связку клиентской части с серверной.

1 Анализ требований, выдвигаемых к продукту

1.1 Функциональные требования

Разрабатываемое веб-приложение должно предоставлять пользователю возможность увидеть сводную статистику по покрытию автоматизированными тестами, которая передается с бэкенд части посредством Application Programming Interface (API). Каждая метрика должна быть представлена прогресс-баром для упрощения визуального анализа.

Кроме общей статистики, необходимо дать возможность просматривать статистику по каждому запуску инструмента для подсчета покрытия. Информация о каждом запуске хранится в базе данных (БД) и отправляется на клиентскую часть сервером с помощью API.

Должна быть возможность увидеть полную информацию о файле, хранящем метрики, а также необходимо учесть, что возможно потребуется скачать этот файл на компьютер пользователя.

В статистике должны отображаться следующие данные:

- покрытие строк кода;
- покрытие функций;
- покрытие ветвлений;
- покрытие операторов.

Для упрощения навигации по результатам запусков необходимо добавить фильтрацию данных по уровню покрытия по метрикам и по дате загрузки файла.

Также необходимо предоставить возможность загружать файл с данными о покрытии тестами в формате JSON.

Интерфейс должен корректно отображаться на устройствах с различными разрешениями экрана, включая мобильные телефоны, планшеты и настольные компьютеры.

Все элементы интерфейса должны быть легко доступными и понятными даже для пользователей с минимальной подготовкой.

1.2 Пользовательские истории

1. Пользователь открывает приложение и видит визуально отделенные друг от друга секции загрузки файла, сводной статистики, фильтрации данных, информации по всем файлам.
2. Пользователь локально запускает инструмент по подсчету покрытия, у него должна быть возможность загрузить файл с данными на сервер. Обновленные данные должны отображаться в интерфейсе.
3. Пользователь просматривает сводную статистику, которая подсчитывается на сервере.
4. Пользователь фильтрует результаты прогонов по минимальному покрытию и по дате генерации отчета с помощью блока фильтрации.
5. Пользователь просматривает детальную информацию о файле, кликнув по интересующей его строке.
6. Пользователь скачивает детальную информацию о прогоне в виде JSON файла на локальный компьютер.

1.3 Технические требования

- Используемые технологии: HTML, CSS, JavaScript.
- Структура данных файла:

```
{  
  "file": "file 1",  
  "lines": 82,  
  "functions": 87,  
  "branches": 5,  
  "statements": 98,  
  "uploadTime": 1729774461458,  
  "id": "1"  
}
```


- Загруженные данные должны сохраняться на сервере через соответствующий API-эндпоинт.
- Веб-приложение должно взаимодействовать с внешним API для получения и обновления данных о покрытии тестами.
- Приложение должно корректно работать в современных браузерах (Chrome, Firefox, Safari).

2 Разработка макета веб-приложения

Для верстки веб-страницы необходимо компоновать текстовые и графические элементы – создать макет. Для проектирования макета выбран графический редактор Figma. Он имеет понятный, легкий в освоении интерфейс, а также позволяет отрисовать элементы сайта и веб-приложения, иллюстрации и векторную графику.

После создания нового проекта открывается редактор, позволяющий отрисовать будущий сайт (рис. 2.1).

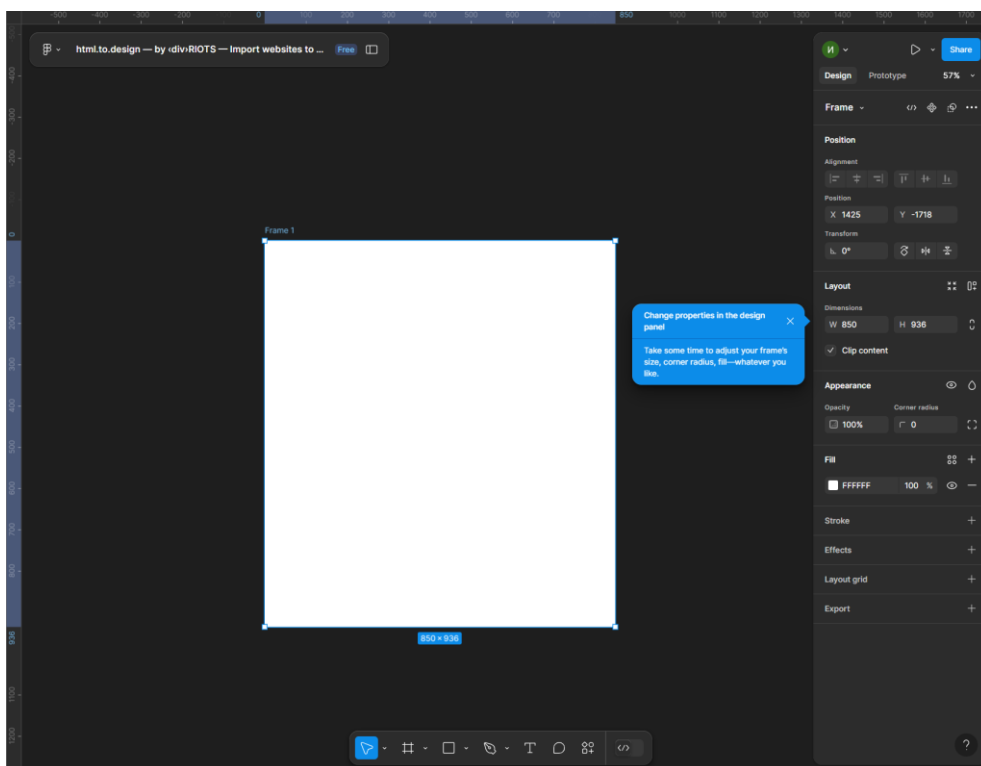


Рисунок 2.1 – Редактор Figma.

При создании макета необходимо опираться на проведенный ранее анализ требований. Все элементы необходимо располагать таким образом, чтобы среднестатистическому пользователю было очевидно, к чему они относятся. Блоки необходимо отделять друг от друга визуально и озаглавливать.

Исходя из функциональных требований и с учетом пользовательских сценариев была разработана следующая структура сайта:

1. Хедер, содержащий информацию о названии и назначении веб-приложения.

2. Блок загрузки файла, имеющий поле типа `input` для загрузки файла и кнопку для подтверждения действия.

3. Секция с суммарной статистикой по покрытию тестов. Для каждой из метрик необходим прогресс-бар для лучшего визуального восприятия.

4. Блок с фильтрами для облегчения навигации. Фильтр по минимальному покрытию, фильтр по датам и кнопка для подтверждения действия.

5. Таблица, содержащая детали о каждом файле.

Разработанный макет сайта содержит стандартные шрифты, которые будут читаться в разных браузерах. Для поддержки адаптивности отрисованы две версии сайта:

- десктопная с шириной 1920 пикселей;
- мобильная с шириной 390 пикселей.

Кроме того, в процессе создания макета отрисованы элементы прогресс-бара и кнопки в уникальном стиле, они представлены на рисунке 2.2. Также на макете представлены все размеры в пикселях, чтобы при разработке было проще сверстать сайт.

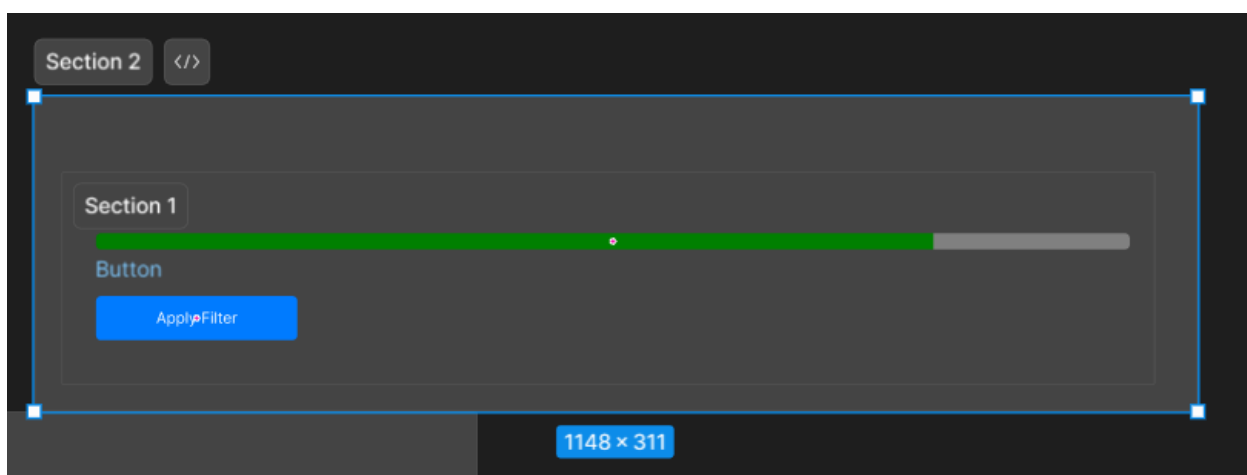


Рисунок 2.2 – Прогресс бар и кнопка.

На рисунке 2.3 представлена окончательная версия макета сайта для десктопа, а на рисунке 2.4 - окончательная версия макета сайта для мобильного веба.

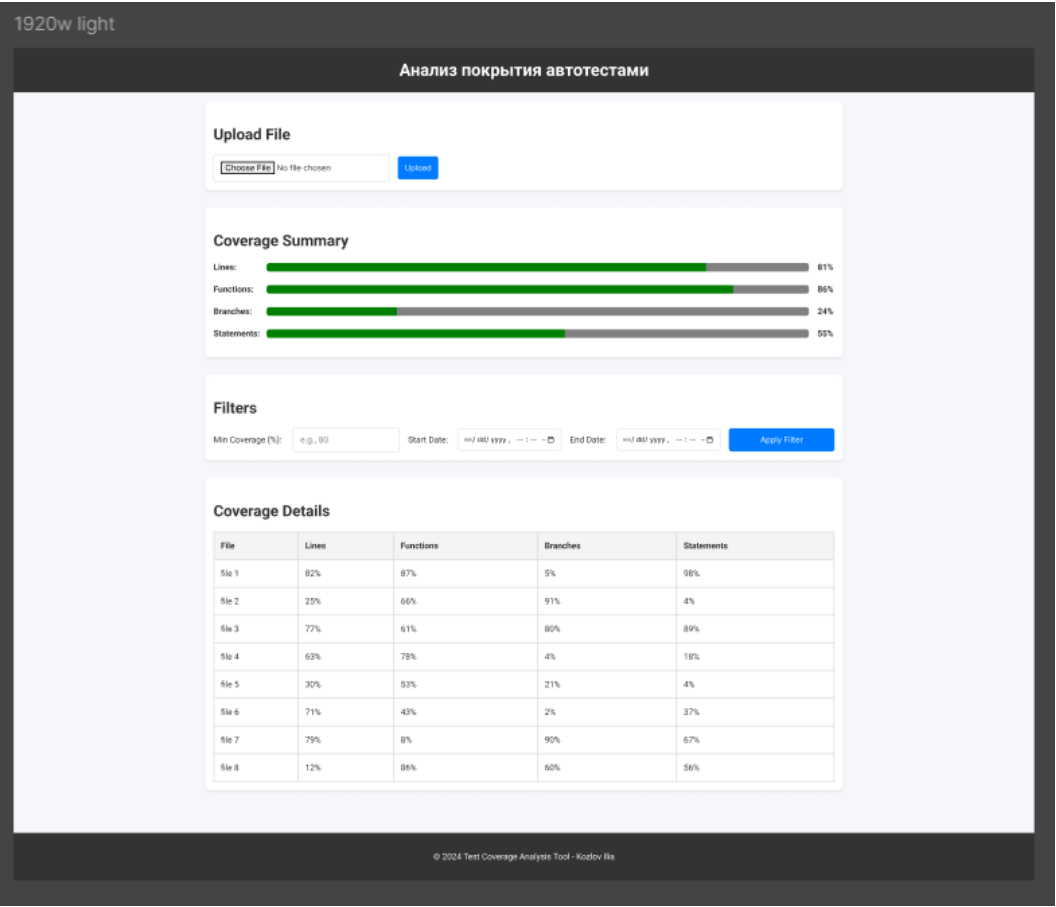


Рисунок 2.3 – Макет сайта для десктопа.

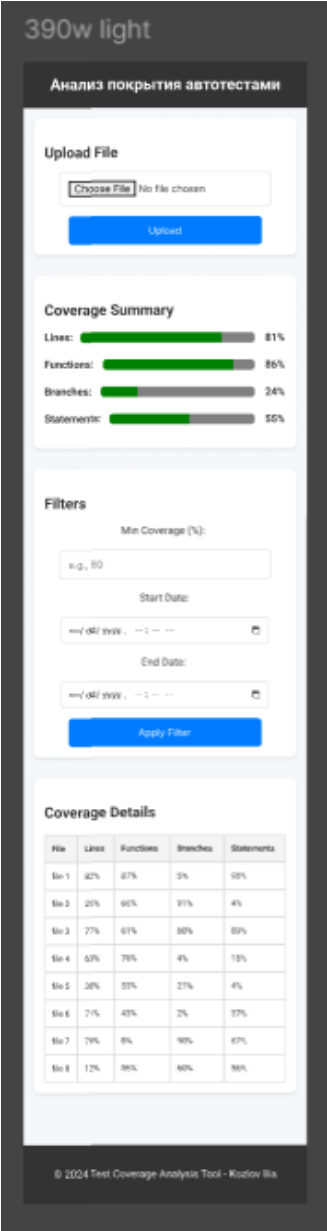


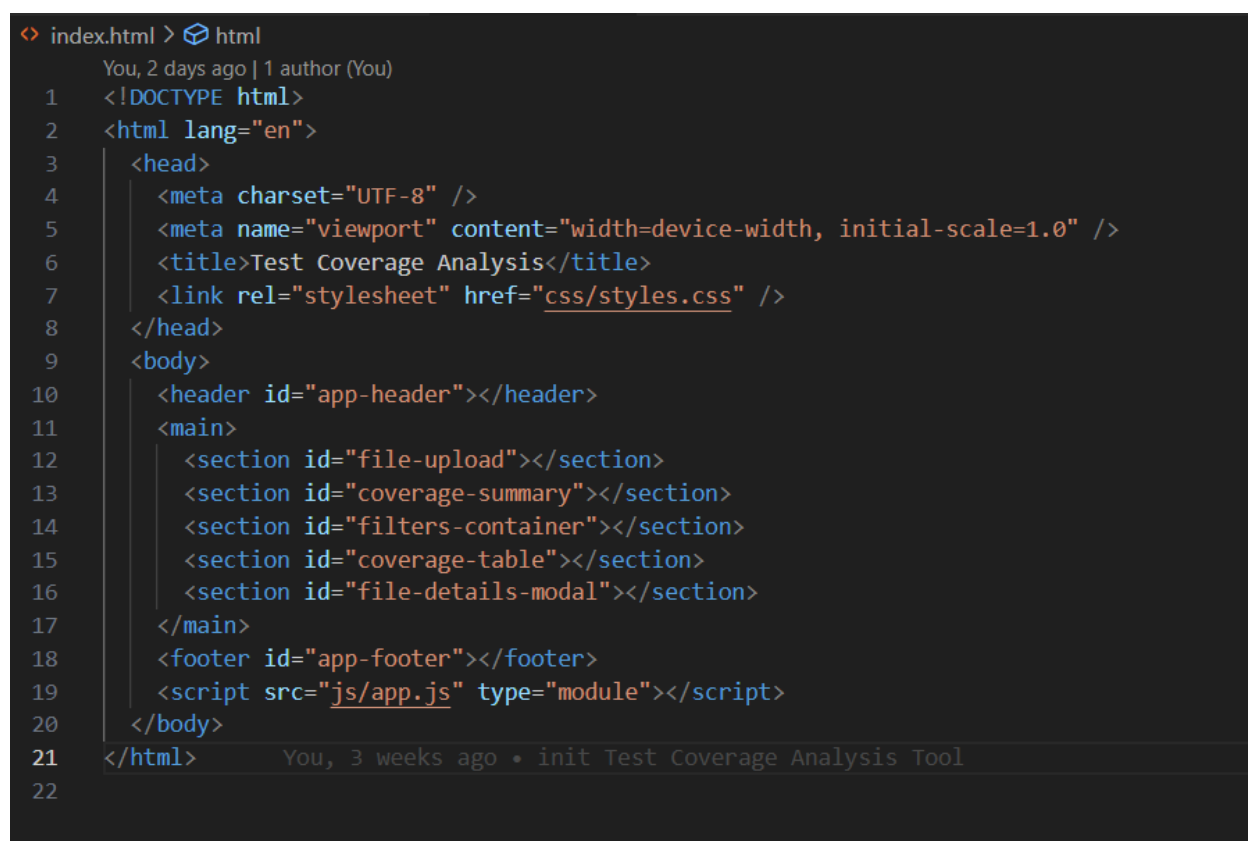
Рисунок 2.4 – Макет для мобильного веба.

3 Верстка сайта по макету

3.1 Создание структуры сайта

Для создания каркаса сайта, основанного на готовом макете, необходимо использовать язык разметки HTML (HyperText Markup Language — «язык гипертекстовой разметки»). Этот язык поддерживается всеми браузерами и позволяет создать блочную структуру с элементами, заранее определенными при проектировании макета.

Изначально необходимо создать «скелет» будущего сайта — исходя из определенной ранее структуры, с помощью тегов разметки выделить блоки будущего веб-приложения (рис. 3.1).



```
index.html > html
You, 2 days ago | 1 author (You)
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>Test Coverage Analysis</title>
7     <link rel="stylesheet" href="css/styles.css" />
8   </head>
9   <body>
10    <header id="app-header"></header>
11    <main>
12      <section id="file-upload"></section>
13      <section id="coverage-summary"></section>
14      <section id="filters-container"></section>
15      <section id="coverage-table"></section>
16      <section id="file-details-modal"></section>
17    </main>
18    <footer id="app-footer"></footer>
19    <script src="js/app.js" type="module"></script>
20  </body>
21 </html> You, 3 weeks ago • init Test Coverage Analysis Tool
22
```

Рисунок 3.1 – Создание структуры сайта на с использованием HTML.

Далее с использованием стандартных тегов HTML формируется полная структура клиентской части (рис. 3.2). Полный код файла index.html приведен в Приложении А. Результат проделанной работы представлен на рисунке 3.3.

```

<body>
  <header id="app-header">
    <h1>Анализ покрытия автотестами</h1>
  </header>
  <main>
    <section id="file-upload">
      <h2>Upload File</h2>
      <form id="file-upload-form">
        <input type="file" id="file-input" accept="application/json">
        <button type="submit">Upload</button>
      </form>
    </section>
    <section id="coverage-summary">
      <h2>Coverage Summary</h2>
      <div class="stats-bar">
        <label>Lines:</label>
        <progress id="lines-coverage" max="100" value="81"></progress>
        <span id="lines-percentage">81%</span>
      </div>
      <div class="stats-bar">
        <label>Functions:</label>
        <progress id="functions-coverage" max="100" value="86"></progress>
        <span id="functions-percentage">86%</span>
      </div>
      <div class="stats-bar">
        <label>Branches:</label>
        <progress id="branches-coverage" max="100" value="24"></progress>
        <span id="branches-percentage">24%</span>
      </div>
      <div class="stats-bar">
        <label>Statements:</label>
        <progress id="statements-coverage" max="100" value="55"></progress>
        <span id="statements-percentage">55%</span>
      </div>
    </section>
    <section id="filters-container">
      <h2>Filters</h2>
      <form id="filter-form">
        <label for="min-coverage">Min Coverage (%):</label>
        <input type="number" id="min-coverage" name="min-coverage" min="0" max=

```

Рисунок 3.2 – Стандартные HTML теги, составляющие структуру сайта.

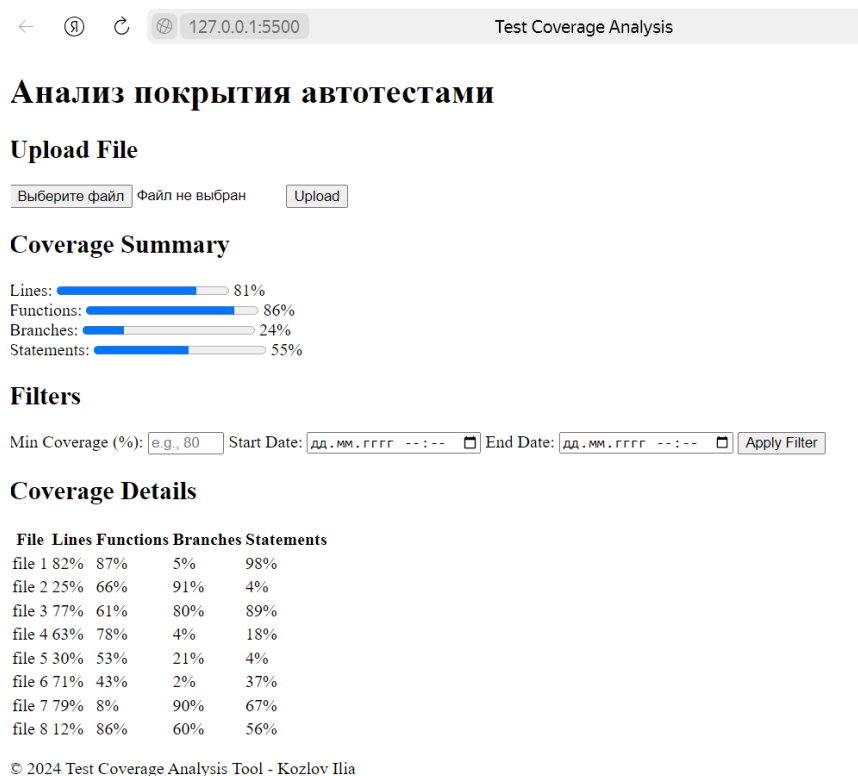


Рисунок 3.3 – Веб-интерфейс, построенный с помощью HTML.

3.2 Добавление стилизации

Для формирования внешнего вида страницы, написанной с использованием языка разметки HTML необходимо использовать формальный язык CSS (Cascading Style Sheets, «каскадные таблицы стилей»). Объявление стиля состоит из следующих частей:

- селектор – например, селектор `h1` обозначает, что описанный стиль будет применен к любому тегу `h1` на странице;
- свойство – например, `font-size` отвечает за размар текста;
- значение – например, `2em` означает, что размер текста будет в 2 раза больше стандартного, используемого браузером.

Полностью стиль заголовка будет выглядеть следующим образом:

```
h1 {  
    margin: 0;  
    font-size: 2em;  
}
```


Аналогичным образом необходимо стилизовать остальные элементы интерфейса (рис. 3.3). Полученный результат представлен на рисунке 3.4.

```
148
149 .modal.hidden {
150   display: none;
151 }
152
153 .modal-content {
154   background: #fff;
155   padding: 2em;
156   border-radius: 10px;
157   width: 90%;
158   max-width: 500px;
159   box-shadow: 0 5px 15px rgba(0, 0, 0, 0.3);
160   position: relative;
161 }
162
163 .close-btn {
164   position: absolute;
165   top: 10px;
166   right: 15px;
167   font-size: 1.5em;
168   color: #333;
169   cursor: pointer;
170 }
171
172 .close-btn:hover {
173   color: #ff0000;
174 }
175
176 .modal button {
177   padding: 0.8em;
178   font-size: 1em;
179   border: 1px solid #ddd;
180   border-radius: 5px;
181 }
182
183 .modal button {
184   background-color: #007bff;
185   color: #fff;
186   cursor: pointer;
187   border: none;
188 }
189
190 .modal button:hover {
191   background-color: #0056b3;
192 }
193
194 /* Анимация модального окна */
195 .modal.show {
196   animation: fadeIn 0.3s ease-out forwards;
197 }
```

Рисунок 3.3 – Пример стилизации элементов интерфейса.

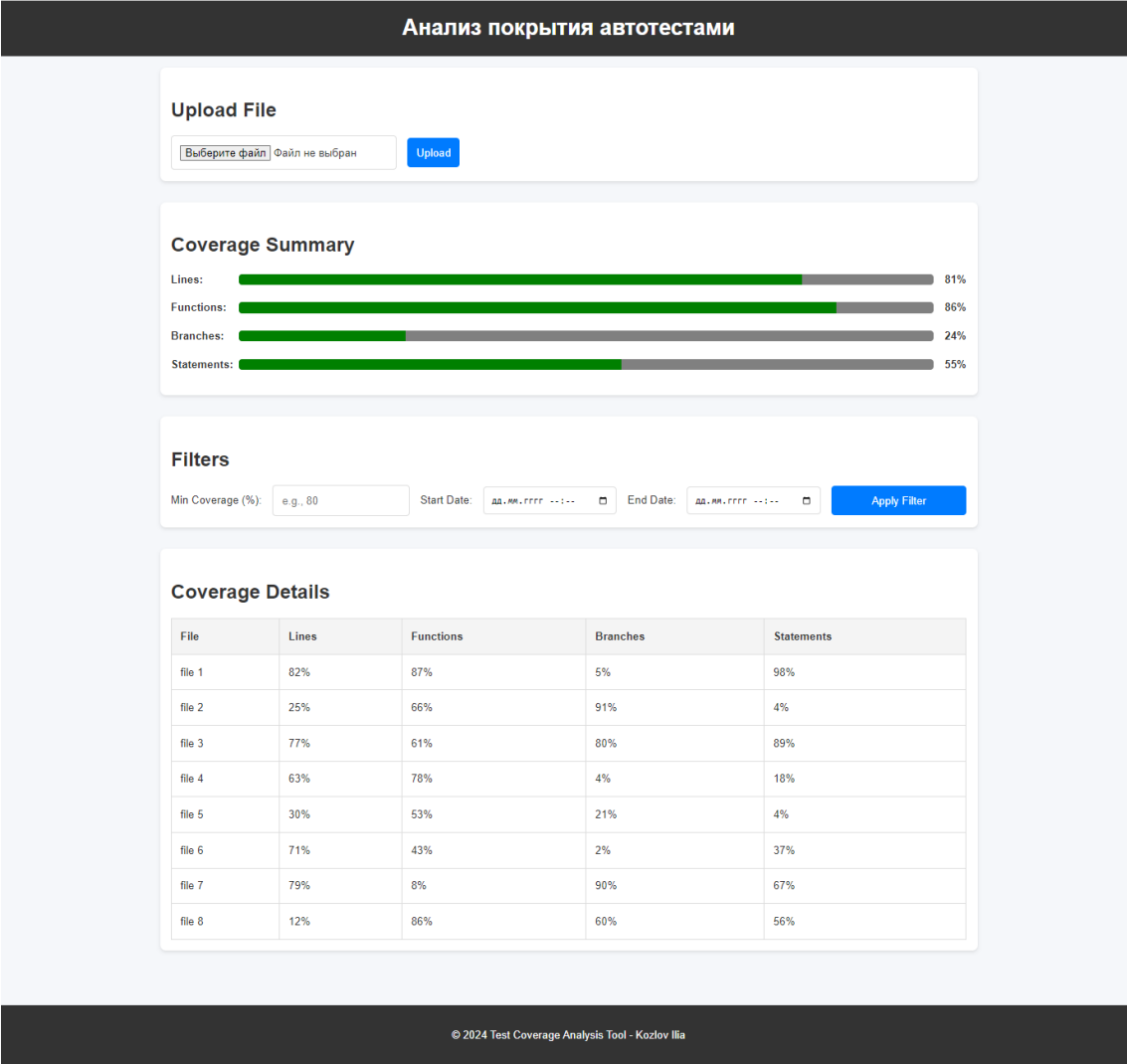


Рисунок 3.4 – Сайт с добавленной стилизацией.

Также согласно проведенному анализу требований и составленному макету, необходимо добавить адаптивность для получившегося сайта (рис.3.5).

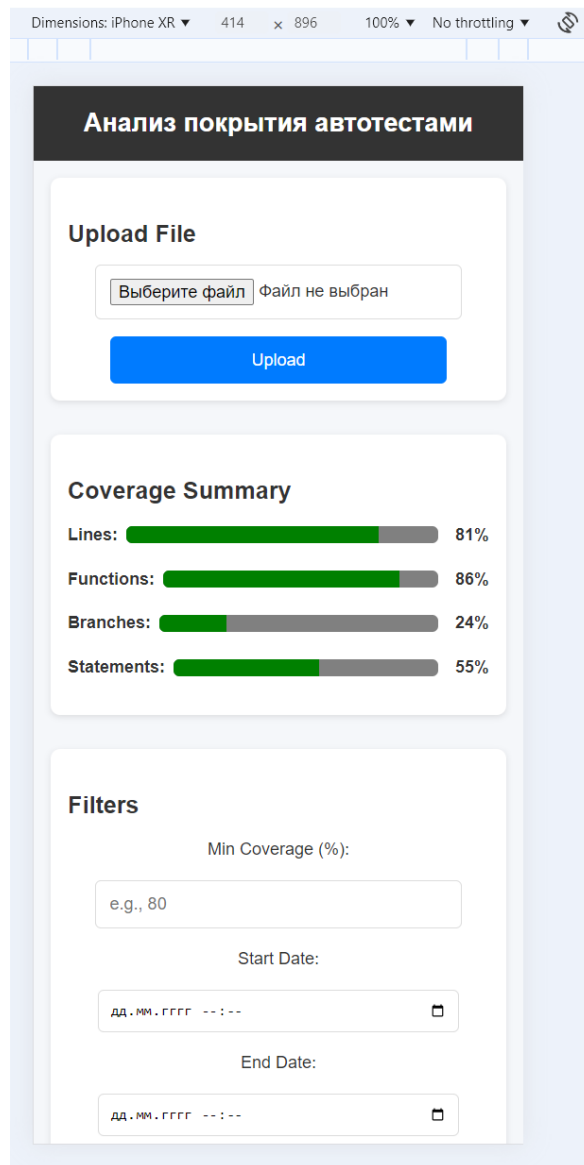


Рисунок 3.5 – Адаптивная верстка.

4 Внедрение интерактивности

Процесс внесения изменений в HTML документ крайне трудозатратен и неэффективен. Полученная ранее версия веб-приложения не зависит от данных на сервере и не поддерживает взаимодействие с пользователем. По этой причине необходимо внедрить интерактивность в приложение.

Для отделения логики приложения от пользовательского интерфейса и упрощения разработки был выбран язык программирования JavaScript и архитектура MVP (Model-View-Presenter). Данная архитектура позволяет легко добавлять новые функции или изменять существующие, не затрагивая другие части приложения, что обеспечивает простоту тестирования, поскольку каждый модуль изолирован.

Модель MVP предполагает разделение ответственности:

- model отвечает за данные и логику их обработки;
- view отвечает только за отображение данных;
- presenter управляет взаимодействием между model и view.

Для проекта была разработана структура директорий и файлов, представленная на рисунке 4.1

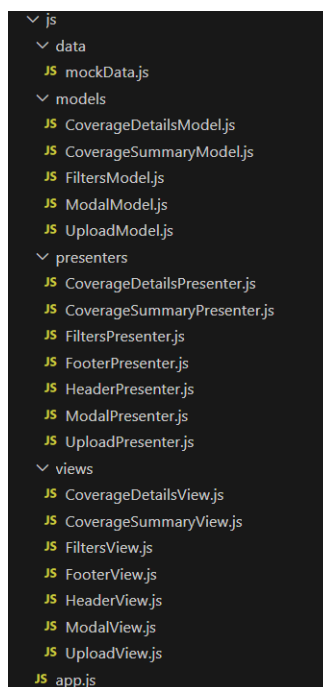


Рисунок 4.1 – Структура директорий и файлов.

В папке data временно хранятся жестко закодированные данные в формате JSON, необходимые для построения таблиц.

В папке models хранятся все модели для управления данными приложения. Также в модели можно выполнять операции с API, такие как загрузка или сохранение данных.

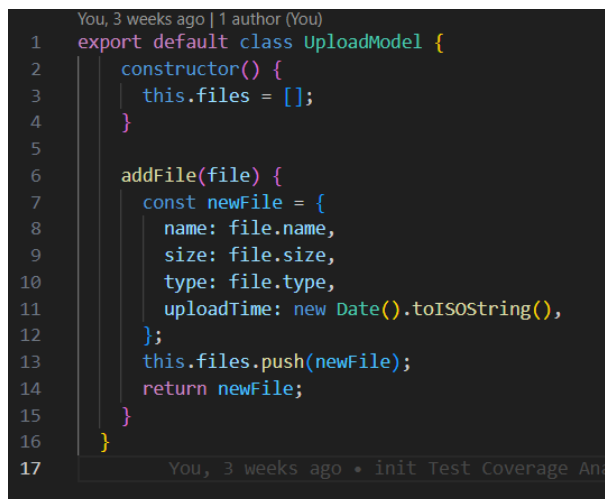
В папке views лежат файлы, генерирующие HTML на основе предоставленных данных, а также обрабатывающие взаимодействие пользователя с интерфейсом.

В папке presenters хранятся файлы, реализующие бизнес-логику приложения. Также в них обрабатываются пользовательские действия посредством запросов данных у models и передачи их во views.

App.js – это центральная точка приложения, в ней инициализируются все компоненты и связываются между собой.

Рассмотрим пример выноса секции загрузки файлов из index.html:

- в классе UploadModel, приведенном на рисунке 4.2 содержится переменная files, хранящая массив загружаемых файлов, а также функция addFile, которая добавляет файл в этот массив;

A screenshot of a code editor with a dark background. It shows the code for a class named UploadModel. The code is as follows:

```
1 export default class UploadModel {
2   constructor() {
3     this.files = [];
4   }
5
6   addFile(file) {
7     const newFile = {
8       name: file.name,
9       size: file.size,
10      type: file.type,
11      uploadTime: new Date().toISOString(),
12    };
13    this.files.push(newFile);
14    return newFile;
15  }
16 }
17
```

Line numbers 1 through 17 are visible on the left side of the code block. The status bar at the bottom shows "You, 3 weeks ago • init Test Coverage An".

Рисунок 4.2 – Код класса UploadModel.

- в классе UploadView (рис.4.3) содержится функция render(), позволяющая отрисовать элемент в интерфейсе. За контейнер берется родительский элемент всего блока. Далее в контейнер с помощью изменения

свойства `innerHTML` помещается код разметки HTML, написанный при верске сайта по макету. Также в классе есть функция `bindFileUpload`, в которой добавляется реакция на событие `submit`, которое возникает при нажатии кнопки;

```
You, 3 weeks ago | 1 author (You)
1 export default class UploadView {
2   constructor() {
3     this.container = document.getElementById("file-upload");
4   }
5
6   render() {
7     this.container.innerHTML = `
8       <h2>Upload Test Coverage Report</h2>
9       <form id="upload-form">
10         <input type="file" id="coverage-file" accept=".json" />
11         <button type="submit">Upload</button>
12       </form>
13     `;
14     this.uploadForm = document.getElementById("upload-form");
15     this.fileInput = document.getElementById("coverage-file");
16   }
17
18   bindFileUpload(handler) {
19     this.uploadForm.addEventListener("submit", (event) => {
20       event.preventDefault();
21       const file = this.fileInput.files[0];
22       if (file) {
23         handler(file);
24         this.fileInput.value = ""; // Сброс input после загрузки
25       }
26     });
27   }
28 }
29
```

Рисунок 4.3 – Код класса UploadView.

- в классе `UploadPresenter` реализована связь между моделью и представлением. При загрузке файла и последующем клике по кнопке `Submit` происходит вывод в консоль информации о файле. В дальнейшем содержимое файла будет отправляться на сервер для обновления таблицы;

```

You, 1 second ago | 1 author (You)
1  export default class UploadPresenter {
2      constructor(model, view) {
3          this.model = model;
4          this.view = view;
5
6          this.view.render();
7          this.view.bindFileUpload(this.handleFileUpload.bind(this));
8      }
9
10     handleFileUpload(file) {
11         const newFile = this.model.addFile(file);
12         console.log("Uploaded file:", newFile);
13     }
14 }
15
You, 3 weeks ago • init Test Coverage Analysis Tool

```

Рисунок 4.4 – Код класса UploadPresenter.

- инициализация всех ранее описанных классов происходит в файле app.js (рис. 4.5);

```

65  // --- Upload ---
66  const uploadModel = new UploadModel();
67  const uploadView = new UploadView();
68  const uploadPresenter = new UploadPresenter(uploadModel, uploadView);
69

```

Рисунок 4.5 – Инициализация классов.

- после проделанной работы можно убрать содержимое секции file-upload из файла index.html. Обновленный тег выглядит следующим образом: `<section id="file-upload"></section>`.

Проделав аналогичные шаги с остальными секциями, можно добавить интерактивность для всех блоков из index.html. Финальный код этого файла представлен на рисунке 4.6. Теперь при изменении блока нет необходимости менять данный файл. Отображение веб-интерфейса при этом осталось прежним.

```
You, 2 days ago | 1 author (You)
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>Test Coverage Analysis</title>
7     <link rel="stylesheet" href="css/styles.css" />
8   </head>
9   <body>
10    <header id="app-header"></header>
11    <main>
12      <section id="file-upload"></section>
13      <section id="coverage-summary"></section>
14      <section id="filters-container"></section>
15      <section id="coverage-table"></section>
16      <section id="file-details-modal"></section>
17    </main>
18    <footer id="app-footer"></footer>
19    <script src="js/app.js" type="module"></script>
20  </body>
21 </html>
22
```

Рисунок 4.6 – Оптимизированный код файла index.html.

5. Связь функционала с серверной частью

В данный момент все данные прописаны заранее (замокированы), что не позволяет динамически взаимодействовать с веб-приложением. Чтобы получать актуальную информацию с сервера и сохранять изменения, необходимо связать клиентскую часть с серверной по следующему алгоритму:

1. Создание и использование эндпоинтов API. Задействован сервер MockAPI для хранения данных о покрытии тестов. Основной эндпоинт: /coverage-details – для работы с данными о покрытии.

2. Запросы к API. Функции, реализующие общение с сервером, представлены в файле ApiService.js (рис. 5.1). Обмен с сервером с помощью методов HTTP:

- GET для получения текущих данных с сервера
- POST для добавления новых файлов.

```
You, 2 days ago | 1 author (You)
1 export const BASE_URL = "https://6721179d98bbb4d93ca76b6f.mockapi.io/api/todo-list";
2
3 export async function fetchCoverageDetails() {
4   try {
5     const response = await fetch(`${BASE_URL}/coverage-details`);
6     if (!response.ok) {
7       throw new Error(
8         `Failed to fetch coverage details: ${response.statusText}`
9       );
10    }
11    return await response.json();
12  } catch (error) {
13    console.error("Error fetching coverage details:", error);
14    throw error;
15  }
16 }
17
18 export async function fetchInitialCoverageData() {
19   try {
20     const response = await fetch(`${BASE_URL}/initial-coverage-data`);
21     if (!response.ok) {
22       throw new Error(
23         `Failed to fetch coverage details: ${response.statusText}`
24       );
25     }
26     const data = await response.json();
27     return data.length > 0 ? data[0] : null;
28   } catch (error) {
29     console.error("Error fetching coverage details:", error);
30     throw error;
31   }
32 }
33
```

Рисунок 5.1 – Состав файла ApiService.js

4. Интеграция с функционалом. При загрузке файла из интерфейса он обрабатывается презентером и сохраняется на сервер через модель (рис. 5.2). После отправки данных из файла на сервер страница перезагружается и обновленные данные запрашиваются повторно, а далее интерфейс рендерится на основе полученного ответа.

```
You, 2 days ago | 1 author (You)
1 import { BASE_URL } from "../api/ApiService.js";
You, 2 days ago | 1 author (You)
2 export default class UploadModel {
3   async addFile(fileContent) {
4     try {
5       const response = await fetch(`${BASE_URL}/coverage-details`, {
6         method: "POST",
7         headers: {
8           "Content-Type": "application/json",
9         },
10        body: JSON.stringify(fileContent),
11      });
12      if (!response.ok) {
13        throw new Error(`Failed to upload file: ${response.status}`);
14      }
15      return await response.json();
16    } catch (error) {
17      console.error("Error uploading file:", error);
18      return null;
19    }
20  }
21 }
22
```

Рисунок 5.5 – Обработка загружаемого файла.

```
You, 2 days ago | 1 author (You)
1 export default class UploadPresenter {
2   constructor(model, view) {
3     this.model = model;
4     this.view = view;
5
6     this.view.render();
7     this.view.bindFileUpload(this.handleFileUpload.bind(this));
8   }
9
10  async handleFileUpload(fileContent) {
11    const newFile = await this.model.addFile(fileContent);
12    if (newFile) {
13      location.reload();
14    } else {
15      console.error("Failed to upload file.");
16    }
17  }
18 }
19
```

Рисунок 5.6 – Перезагрузка страницы после отправки данных на сервер.

Пример потока данных

1. Пользователь загружает файл через форму.
2. Презентер вызывает метод модели для отправки данных на сервер через POST-запрос.
3. Сервер добавляет новые данные и возвращает результат.
4. Приложение получает обновленные данные и рендерит их на странице.

Благодаря реализации динамического взаимодействия появилась возможность разграничить доступ к данным — клиент отвечает только за отображение данных, а сервер за их обработку и хранение. Также удалось добиться централизации данных, что обеспечивает согласованность и возможность их обновления в реальном времени.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

