UNIVERSITY OF TARTU
Faculty of Social Studies
Narva College
Study program “Information Technology Systems Development”

Ilja Holmogortsev

DEVELOPMENT OF A WEB APPLICATION FOR VIRTUAL RIM FITMENT SIMULATION ON USER-UPLOADED VEHICLE IMAGES

Final Thesis

Supervisor(s): Deniss Ruder, M. Sc

Narva 2025

Thesis Title in English

Abstract:

This project presents a web application designed to visualize how different rims would appear on a user's vehicle. Users upload images of their vehicle and selected rims via an intuitive interface built with Next.js, which serves as the front-end. These images are then sent to the backend, powered by Nest.js, where they are stored in Cloudflare R2 Storage. The backend generates URLs for the uploaded images and forwards them to FastAPI through RabbitMQ for asynchronous processing. FastAPI, leveraging Python, handles image manipulation, overlaying the rim images onto the vehicle images to create a realistic preview. Data management is supported by PostgreSQL for persistent storage and Redis for session handling. This application demonstrates the integration of modern web technologies to enhance user experience and streamline automotive customization.

Lõputöö pealkiri

Lühikokkuvõte:

See projekt tutvustab veebirakendust, mis on loodud visualiseerima, kuidas erinevad veljed kasutaja sõidukil välja näeksid. Kasutajad laadivad üles oma sõiduki ja valitud velgede pildid intuitiivse liidese kaudu, mis on ehitatud Next.js abil ja toimib esirakendusena. Need pildid saadetakse seejärel rakenduse serveri ossa, mida toetab Nest.js, kus need salvestatakse Cloudflare R2 Storage'is. Serveri osa genereerib üleslaaditud piltide URL-id ja edastab need FastAPI-le RabbitMQ kaudu asünkroonseks töötlemiseks. FastAPI, kasutades Pythoni, tegeleb piltide manipuleerimisega, kattes sõiduki piltidele velgede pildid, et luua realistlik eelvaade. Andmehalduse toetuseks on kasutusel PostgreSQL püsivaks salvestamiseks ja Redis seansside haldamiseks. See rakendus näitab kaasaegsete veebitehnoloogiate integreerimist kasutajakogemuse parandamiseks ja auto personaliseerimise lihtsustamiseks.

License

I, Ilja Holmogortsev,

1. grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis DEVELOPMENT OF A WEB APPLICATION FOR VIRTUAL RIM FITMENT SIMULATION ON USER-UPLOADED VEHICLE IMAGES, supervised by Deniss Ruder.
2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the Dspace digital archives, under the Creative Commons license CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Ilja Holmogortsev

19.05.2025

Table of Contents

Introduction

In recent years, the automotive industry has undergone significant changes, with more people shopping online and using digital tools to customize their cars. Car enthusiasts and potential buyers often want to see how a car will look with new parts, such as different rims, which greatly affect its appearance. However, physically changing rims on a car takes a lot of time and is not always possible.

Problem

Current tools for visualizing rims, such as those offered by car dealers or manufacturers, have limitations. They usually work only with pre-loaded car models and have limited rims collection to try-on, also they do not allow users to upload their own photos of either car or rims. This reduces flexibility, especially for rare or older models, which may not be in their databases. Additionally, many of these tools show only static images, without the option to view the car from different angles. Therefore, there is a need for a more convenient and universal solution.

Goal

This bachelor's thesis presents a web application that addresses these issues. It allows users to upload photos of their cars and photos of their rims, then overlays the rims onto the car image to create a realistic visualization. To achieve this, I used modern technologies: OpenCV for image processing and YOLOv8 for image segmentation to accurately detect rim areas. I also compiled a custom dataset of over 300 car images from auto24.ee and trained a specialized model using inbuilt YOLOv8 training tool to improve the application's accuracy.

The application also includes user-friendly features. Registered users can create accounts, publish their customized images, and add them to favorites. All users, including unregistered ones, can view galleries on the main page and search for images by tags or car brands, making it easier to explore and find content.

Tasks

The primary objectives of this thesis are:

1. To develop an intuitive interface using Next.js for seamless image uploading and visualization.
2. To build a robust backend with Nest.js for efficient image storage and processing.

3. To integrate FastAPI and Python for precise image manipulation, including overlaying rims on car photos.
4. To support a wide range of car models and viewing angles, addressing the limitations of existing tools.

Achieving these goals will result in a practical and innovative tool that enhances the car customization experience and contributes to the advancement of digital visualization technologies.

1. Research

To support the development of a web application designed for virtual rim fitment simulation on user-uploaded vehicle images, the author conducted market research to identify existing visualization tools that tackle a similar challenge. Through this process, the author discovered several main competitors offering solutions that partially address the problem, highlighting opportunities for the new project to deliver a more flexible tool for automotive customization without depending on pre-installed models.

1.1 Existing Solutions

1.1.1 The Wheel Group

The Wheel Group is a leading manufacturer and distributor of aftermarket automotive wheels, tires, and accessories. On their website they have a tool called “Visualizer”. The visualizer is an online tool that allows users to see how different wheel designs, sizes, and finishes look on their specific vehicles. Users can select their car's make, model, and year to virtually “try-on” rims, helping them make informed purchasing decisions.

Pros:

1. Wide variety of options: With multiple proprietary brands users have access to diverse designs, sizes, and finishes.
2. Good quality of Images: The Wheel Group offers pre-loaded car and rim images to virtually try-on rims.
3. User-Friendly Interface: The tool is designed to be intuitive, making it easy for users to navigate and find suitable wheels.

Cons:

1. Limited vehicle and rims selection: Not all vehicles or rims may be available, which could restrict its usefulness for some users.
2. Limited car position: There are only a few positions in which you can preview the car with the rim.
3. No possibility to upload your car or rim: You are unable to try on the available rim on your vehicle.
4. Not a real car: The Wheel Group offers two plane images of the selected car but not the real one.

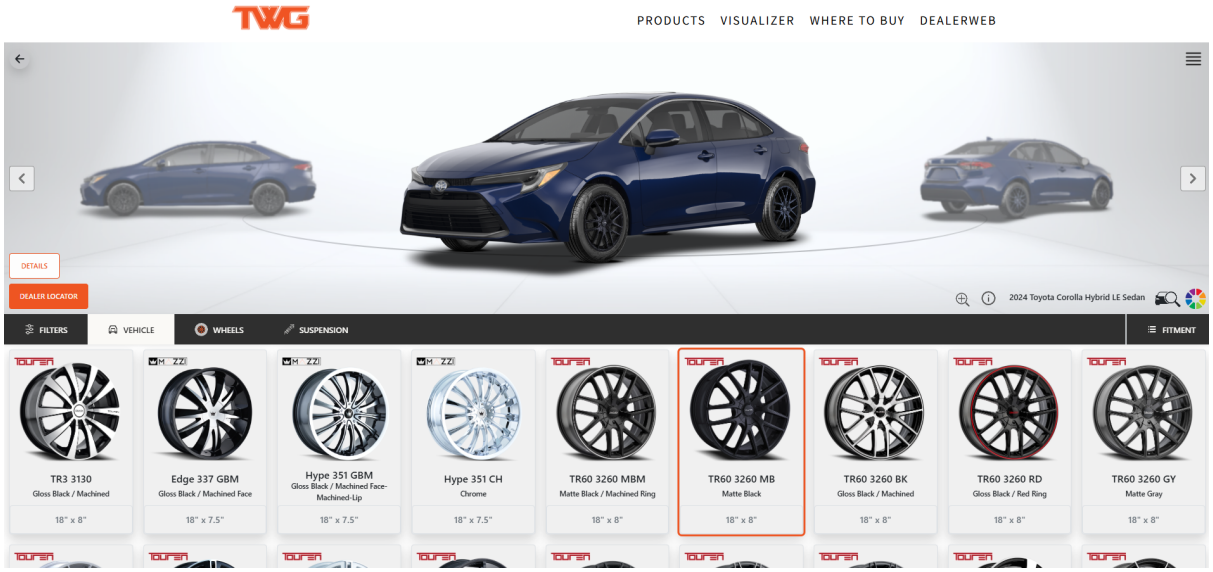


Figure 1. The wheel group image

1.1.2 RAW Wheels + Tires

RAW Wheels + Tires, focuses on making custom wheels and performance tires affordable, offering a vast inventory with flexible payment options like Rent-To-Own. RAW's visualizer tool, accessible at RAW Wheels + Tires website, helps users find the perfect wheel and tire match without guesswork.

Pros:

1. Detailed vehicle matching: Allows selecting color, enhancing visualization accuracy, and uses up-to-date databases including the newest cars.

Cons:

1. Lack of advanced features: Features like 3D rotation or suspension adjustment.
2. Only one position of the car: RAW Wheels + Tires visualizer tool offers only one position at which you can preview selected rim.
3. Limited choice: Only the rims that are on sale on the website are available.

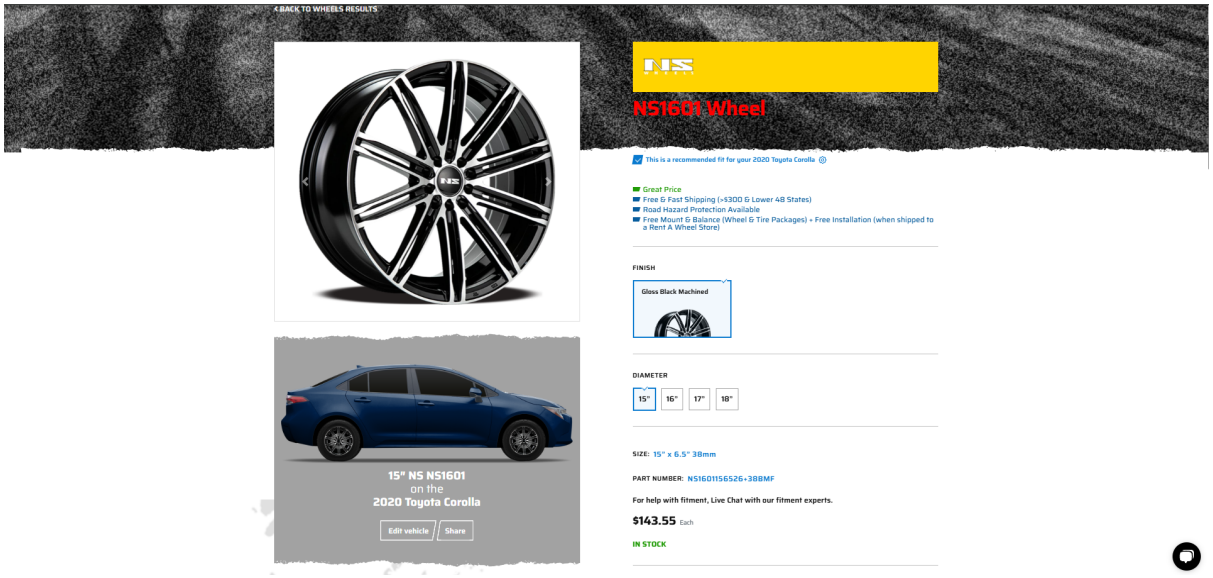


Figure 2. RAW Wheels + Tires image

1.1.3 3Dtuning

The 3DTuning app lets you fully customize cars, trucks, and bikes, from color of the car to front bumpers and tail lights, including wheels, with photorealistic visuals. You can adjust wheel dimensions, offset, and choose from a unique collection of wheels and tires, like off-road or sport types.

Pros:

1. Comprehensive customization: Part of a broader car configurator, allowing for holistic vehicle modifications, including wheels, with photorealistic quality.
2. Advanced wheel adjustments: Users can modify wheel dimensions, offset, and providing detailed control over the appearance.
3. Advanced features: Feature like 3D rotation where you can preview your car almost at any angle.

Cons:

1. Rims selection: May not offer as many rim designs as specialized wheel visualizers.
2. Restricted car selection: There are no older or lesser-known car brands available.

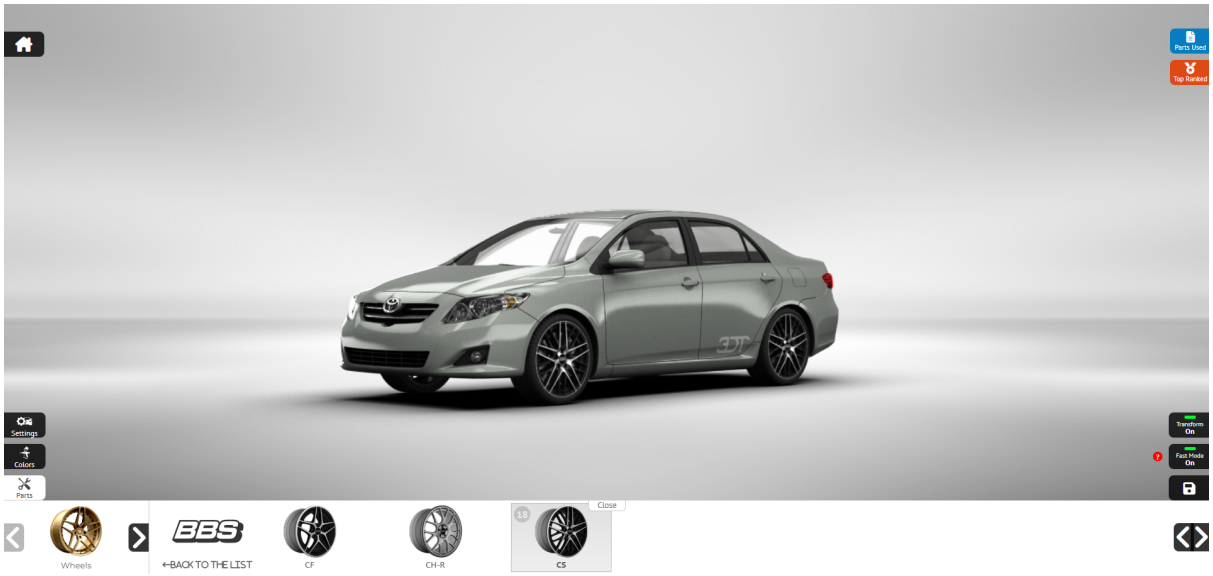


Figure 3. 3DTuning image

1.2 Research Summary

To support the development of a web application for virtual rim fitment simulation using user-uploaded vehicle images, a comprehensive review of existing visualization tools for automotive customization was conducted. The research aimed to identify the strengths and weaknesses of current solutions, revealing a significant gap in tools that allow users to visualize rims on their own vehicles rather than relying on pre-installed models. While several tools offer valuable features for general automotive customization, they lack the flexibility and functionality required for a fully personalized experience. The key findings from the analysis of competitors such as The Wheel Group, RAW Wheels + Tires, and 3DTuning are outlined below:

1. Limited customization and flexibility

Existing tools, such as The Wheel Group and RAW Wheels + Tires, provide pre-loaded car and rim images, restricting users to a fixed selection of vehicles and rim designs. For instance, The Wheel Group's Visualizer offers a wide variety of rim options across multiple brands, but it does not include all vehicles such as rare or modified models like my car the Mazda Familia 323 or allow users to upload their own car images. Similarly, RAW Wheels + Tires limits users to rims available for purchase on their site, reducing options for broader customization. This lack of flexibility hinders users seeking to visualize rims on unique or personal vehicles.

2. Lack of advanced features and personalization

While tools like 3DTuning provide comprehensive customization options such as wheel dimension adjustments, offset, and camber they do not support user-uploaded vehicle images, limiting their applicability for users wanting to see rims on their actual cars. The Wheel Group and RAW Wheels + Tires, despite offering user-friendly interfaces and detailed vehicle matching (e.g., color selection), lack advanced features like 3D rotation or suspension adjustments. Moreover, none of these tools allow users to upload custom vehicle or rim images, meaning the visualizations are not based on real cars but rather generic representations.

3. Restricted viewing angles

Most tools, including The Wheel Group and RAW Wheels + Tires, offer static views of vehicles, typically limited to one or two angles. The Wheel Group's Visualizer provides two positions, while RAW Wheels + Tires restricts users to a single perspective. This constraint prevents users from fully assessing how rims appear from different angles, diminishing the realism and practicality of the visualization. In contrast, 3DTuning offers advanced 3D rotation, allowing previews from multiple angles, but it still relies on pre-set models rather than user-uploaded images.

These findings highlight the need for a dedicated web application that overcomes the limitations of existing tools by enabling users to upload their own vehicle images and visualize rims in a flexible, realistic manner. Such an application should leverage advanced image processing and machine learning techniques to accurately detect and overlay rims onto user-uploaded photos, accommodating diverse vehicle modifications and viewing angles. By addressing these gaps, the proposed tool will provide a more personalized and practical solution, significantly enhancing the automotive customization experience for users.

2. Application requirements

This section outlines the systematic approach and techniques employed to develop the web application for virtual rim fitment simulation. The methodology was carefully designed to ensure that the project met its objectives of creating an intuitive, efficient, and reliable tool for users to visualize rims on their vehicles. An iterative development process was adopted, which allowed for continuous improvement and adaptation based on testing. This approach ensured that both functional requirements (e.g., image uploading, rim overlay, user authentication) and non-functional requirements (e.g., performance, scalability, usability) were addressed effectively throughout the project.

The development process was divided into key phases: planning, design, implementation, and testing. In the planning phase, the project's goals and requirements were defined, and the most suitable technologies were selected, including Next.js for the frontend, Nest.js for the backend, and FastAPI for image processing. The design phase focused on creating a user-friendly interface and a robust system architecture, while the implementation phase involved building the application's features incrementally, allowing for regular integration and testing.

For testing, the Nest.js API was thoroughly validated using Postman, a widely used tool for API testing. Postman was employed to create comprehensive test suites that simulated various user interactions, such as image uploads, user authentication, and data retrieval. Each endpoint was rigorously tested to ensure it processed requests accurately, returned appropriate responses, and handled errors effectively. This testing process confirmed the backend's reliability and robustness, ensuring it could support real-world usage scenarios.

This methodology not only facilitated the successful development of the application but also ensured that it was scalable, secure, and user-centric, meeting the needs of both casual users and automotive enthusiasts.

2.1 Functional requirements

These requirements describe what the application should do.

User Interface and Interaction:

- The application must provide an intuitive interface for users to upload images of their vehicles and rims.

- It must display a gallery of generated images (cars with overlaid rims) shared by registered users.
- Registered users must be able to publish their customized images.
- Registered users must be able to add generated images to their favorites list.
- The application must include a search functionality to find images by tags or car brands.

Image processing:

- The system must process uploaded images to accurately overlay the rim images onto the vehicle images.
- It must use computer vision techniques, specifically YOLOv8 for rim detection and OpenCV for image manipulation.

User authentication:

- The application must allow users to register and log in.
- It must manage user sessions and permissions, restricting certain actions (e.g., publishing images, adding to favorites) to registered users.

Data management:

- The system must store user data, uploaded images URL, and generated images URL in a database.
- It must use Cloudflare R2 Storage for image storage and PostgreSQL for other data.

2.2 Non-Functional requirements

These requirements describe how the system should be performed.

Performance:

- The image processing must be efficient and not cause significant delays, even with multiple users.
- The application must load quickly and respond promptly to user actions.

Scalability:

- The system must handle multiple users uploading and processing images simultaneously without significant performance degradation.

Usability:

- The interface must be user-friendly and easy to navigate.
- It must be responsive and work well on different devices, such as desktops and mobiles.

Security:

- User data must be protected, and authentication must be secure.
- Images must be stored securely in cloud storage.

3. Development tools

To create the web application for virtual rim fitment simulation, a range of modern technologies and tools were used. This section explains the key components involved in the development, why they were chosen, and how they contribute to the project. The technologies are divided into two main parts: front-end development and back-end development.

3.1 Frontend development tools

The frontend is the part of the application that users see and interact with. It was built using the following tools:

3.1.1 Next.js

Next.js is an open-source framework built upon React, a widely adopted JavaScript library for crafting dynamic user interfaces. This framework was chosen for its advanced capabilities, notably server-side rendering (SSR) and static site generation (SSG). These features enhance the application's performance by reducing page load times and improve its discoverability by search engines, offering a significant advantage over standalone React implementations, which require additional configuration to achieve similar routing and rendering functionalities.

In this project, Next.js proved instrumental in developing the application's core structure, comprising three primary pages: the homepage, which showcases a gallery of user-generated images; the image generation page, where users upload car and rim photos to produce customized visuals; and the favorites page, restricted to registered users for managing preferred images. The SSR capability was particularly valuable on the homepage, where gallery content is fetched and rendered server-side, ensuring rapid initial display for users. A basic example of a Next.js page component is illustrated below:

```
// create/page.tsx
import {UploadGrid} from "@/components/upload-section";

export default function Page() : Element { no usages  iljah
  return (
      <UploadGrid/>
  )
}
```

Figure 4. Next.js page component

3.1.2 Tailwind CSS and Shadcn UI

For styling the application, Tailwind CSS was employed. Tailwind is a utility-first CSS framework that facilitates rapid and flexible design through the application of pre-defined classes directly within HTML or JSX markup, eliminating the need for extensive custom CSS. This methodology accelerates development while maintaining adaptability in the design process.

Complementing Tailwind CSS, Shadcn UI was integrated to provide a collection of pre-designed, accessible components such as buttons, forms, and modals that align seamlessly with Tailwind's styling system. Together, these tools ensure a modern, consistent aesthetic across the application. For example, the image upload form on the image generation page utilizes Shadcn UI components styled with Tailwind classes, creating an intuitive and visually appealing interface. The decision to adopt Tailwind CSS and Shadcn UI over traditional CSS approaches stemmed from their ability to expedite development and minimize styling inconsistencies, which are more prevalent in manually crafted stylesheets.

Here's a simple example and comparison between traditional CSS and Tailwind CSS:

```
<div className="grid grid-cols-4 items-center gap-4">
    <Label htmlFor="username" className="text-right">
        Username
    </Label>
    <Input
        id="username"
        defaultValue="@username"
        className="col-span-3"
    />
</div>
```

Figure 5. Tailwind CSS with Shadcn

Here is an example of how the exact the same code would look like in traditional CSS and HTML:

```html
<div class="container">
    <label for="username" class="label">Username</label>
    <input id="username" value="@username" class="input">
</div>

<style>
    .container {
        display: grid;
        grid-template-columns: repeat(4, 1fr);
        align-items: center;
        gap: 1rem;
        margin: 1rem 0;
    }

    .label {
        text-align: right;
        padding-right: 1rem;
    }

    .input {
        grid-column: span 3;
        padding: 0.5rem;
        border: 1px solid #ccc;
        border-radius: 4px;
        width: 100%;
    }
</style>
```

Figure 6. Traditional CSS and HTML

3.1.3 App Router

App Router is a feature in Next.js that handles navigation between pages. In this project, there are main four: the homepage, the image generation page, the favorites page and browsing tags. App Router makes it simple to set up and manage these routes, providing a smooth experience for users as they move between pages. App Router approach handles routes based on folder name inside app folder, so each folder inside app/ folder becomes a route below is an example of how my routes look like:

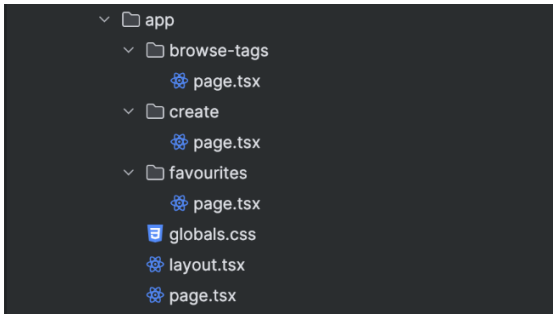


Figure 7. Next.js Routers image

3.1.4 Zustand for State Management

I also utilized Zustand for state management in the Next.js frontend. Zustand is a lightweight and efficient library that works seamlessly with React applications. It was chosen over alternatives like Redux because of its simplicity and minimal setup requirements, which saved time during development without losing functionality. In this project, Zustand managed global states, such as opening and closing modal dialogs like signup modal dialog.

For example, Zustand modifies the global state to adjust the appearance of the modal dialog when a user interacts with a "Settings" button in the menu bar on the left or a button in the top navigation bar. This approach eliminates the need for prop drilling or intricate context setups, keeping the codebase clean, maintainable, and easy to scale. Below is an example of how Zustand was implemented to manage the signup and sign in modal's state:

```
type AuthDialogState = {
    isOpen: boolean;
    toggle: () => void;
}

export const useAuthDialogStore = create<AuthDialogState>((set) => ({
    isOpen: false,
    toggle: () => set((state) => ({isOpen: !state.isOpen})),
}))
```

Figure 8. Zustand auth dialog store

3.1.5 React TanStack Query

React TanStack Query is a powerful library designed to manage server-state in React applications. It provides a collection of hooks that simplify data fetching, caching, and synchronization with external APIs, making asynchronous operations more efficient and manageable. This library was chosen for its ability to streamline server-state management, its built-in caching capabilities, and its seamless integration with React and Next.js, aligning perfectly with the application's front-end architecture.

In this project, React TanStack Query is essential for handling dynamic content, such as retrieving gallery images displayed on the homepage and managing the list of favorite images for registered users. For example, the useQuery hook is used to fetch gallery data, automatically

caching the results to reduce redundant network requests and ensure fast content delivery. Additionally, when a user triggers an image generation process, the useMutation hook handles the submission and updates the gallery in real-time once the operation completes, providing immediate feedback without requiring manual state updates.

Here's an example of how my custom hooks with React TanStack Query are implemented to fetch gallery images and generate car image with overlaid rim:

```
export const useGalleryQuery = (tag?: TagQuery) => {
    return useQuery({
        queryKey: ['gallery', tag?.tagName],
        async queryFn(){
            const r = await getGalleries(tag)
            return r && 'data' in r ? r.data : []
        },
        enabled: true,
    })
}
```

Figure 9 Custom hook to fetch galleries with TanStack Query

```
export const useCarGenerationImage = (): UseMutationResult<
    GeneratedImage,
    Error,
    FormData
> => {
    return useMutation({
        mutationFn: generateCarImage,
    })
}
```

Figure 10 Custom hook to generate car image with TanStack Query

3.2 Backend development tools

The backend is the behind-the-scenes part of the application that processes data and handles logic. It was developed using these technologies:

3.2.1 Nest.js

Nest.js is a progressive Node.js framework designed to build efficient and scalable server-side applications. It leverages TypeScript for type safety and adopts a modular, Angular-inspired architecture that promotes clean code organization.

Nest.js was selected for its structured approach, using modules, controllers, and services to manage complex logic, such as image uploads and authentication. Its built-in TypeScript support reduces runtime errors, and its decorator-based syntax simplifies dependency injection compared to lighter frameworks like Express.js. While Koa offers flexibility, it lacks the robust organization Nest.js provides, making it less suitable for this project's scale.

Nest.js serves as the backbone of the backend, handling HTTP requests and orchestrating workflows like image uploads to Cloudflare R2 and task queuing with RabbitMQ. For instance, an image upload endpoint uses a controller to process files and trigger subsequent actions:

```
@Controller('car-rim')  Show usages  iljah *
export class CarRimController {
    constructor(private carRimService: CarRimService) {}  no usages  iljah

    @Post('/generate')  no usages  iljah
    @UseInterceptors(FileFieldsInterceptor([
        { name: 'car', maxCount: 1 },
        { name: 'rim', maxCount: 1 },
    ]))
    async generateCarImage(@UploadedFiles(
        new ParseFilePipe({
            fileIsRequired: true,
        })
    ) files: UploadFilesDto){
        const carImages: ImageUrlsDto = {
            car: files.car[0],
            rim: files.rim[0],
        }
        return await this.carRimService.generateImage(carImages)
    }

}
```

Figure 11. Image upload endpoint controller

3.2.2 FastAPI and Python

FastAPI is a modern Python web framework optimized for building high-performance APIs with native support for asynchronous operations, paired with Python's rich ecosystem for tasks like image processing.

FastAPI's async capabilities enable concurrent image processing, critical for handling multiple uploads without delays. Unlike Django, which requires additional tools for async tasks, or Flask, which lacks native async support, FastAPI offers simplicity, speed, and automatic API documentation via OpenAPI, aligning with the project's performance needs.

FastAPI processes image URLs from RabbitMQ, using Python libraries like OpenCV and YOLOv8 to overlay rims onto car images. Here's an example of a FastAPI endpoint:

```python
@app.on_event("startup")  iljah *
async def startup():
    connection = await aio_pika.connect_robust("amqp://guest:guest@localhost/")
    channel = await connection.channel()

    queue = await channel.declare_queue(
        name: "fastapi_queue",
        durable=False
    )

    async def on_message(message: aio_pika.IncomingMessage):  iljah *
        async with message.process():
            if message.reply_to:
                try:
                    payload = json.loads(message.body.decode())
                    request = ImageRequest(**payload['data'])
                    generated_car_url = await asyncio.to_thread(process_image_request, *args: request)
                except (ValidationError, json.JSONDecodeError) as e:
                    print("Ошибка обработки сообщения:", e)

                await channel.default_exchange.publish(
                    aio_pika.Message(
                        body=generated_car_url.encode(),
                        correlation_id=message.correlation_id
                    ),
                    routing_key=message.reply_to,
                )
            else:
                print("Нет reply_to, не можем ответить автоматически")

    await queue.consume(on_message)
```

Figure 12. FastAPI message queue endpoint

3.2.3 YOLOv8

YOLOv8 is a modern model used for object detection and image segmentation, important tasks in computer vision. Object detection means finding and locating objects in an image, while image segmentation divides an image into parts to understand it better. In my project, I used YOLOv8 to find car rims in vehicle photos, making it a useful tool for analyzing car images.

To make the model work well for this task, I trained a custom YOLOv8 model with a dataset of over 300 car images from the website auto24.ee. I labeled these images carefully using Roboflow, a platform that helps prepare data for machine learning. Unlike general pre-trained models, which may not work well for specific tasks like detecting rims in different lighting or angles, my custom model was adjusted to fit my project's needs. This improved the model's ability to find rims accurately in car photos.

I chose YOLOv8 because it is fast and accurate, which is important for real-time image processing. Speed helps the model process images quickly, and accuracy ensures good results. For example, in a web app where users upload car and rim photos to see the overlaid rims on the car photo, YOLOv8's fast and precise performance creates a smooth experience. This makes it a great choice for interactive apps where speed and quality matter.

3.2.4 OpenCV

OpenCV, a library for computer vision tasks, plays a crucial role in this image processing algorithm. After the rims are detected on the original car photo using a YOLO model, OpenCV is used to add the new rim images onto the car photos. Its extensive functions and flexibility allow for precise image manipulation, making it an ideal choice over simpler tools.

The algorithm is designed to overlay a new rim design onto the rims of a car in a photograph, making the rims look natural and well-integrated. Below is a step-by-step explanation of how it works:

Rim image preparation:

- The rim image is downloaded from a provided URL.
- Using OpenCV, the white background is removed to make it transparent by setting nearly white pixels' alpha channel to zero.
- The rim is resized to 1080x1080 pixels while maintaining its aspect ratio.
- A black ellipse is added behind the rim using OpenCV to enhance realism.

Car image loading:

- The car image is downloaded and loaded using OpenCV.

Rims detection:

- A pre-trained YOLO model detects the rims in the car image.
- For each detected rim, OpenCV fits an ellipse to the rim's mask to determine its size, position, and angle.

Rim overlay:

- For each rim, OpenCV's perspective warping adjusts the rim image to match the rim's perspective.
- Multi-Sample Anti-Aliasing (MSAA) is applied to smooth the edges.
- OpenCV's alpha blending overlays the warped rim onto the car image naturally.

Result generation:

- The modified car image with the new rims is returned.

3.2.5 RabbitMQ

RabbitMQ is a message broker that helps different parts of the application talk to each other. It sends image URLs from Nest.js to FastAPI for processing. Unlike direct HTTP requests, RabbitMQ can queue tasks if one part is busy, improving scalability and reliability. Nest.js queues image URLs in RabbitMQ for FastAPI to process:

```
@Injectable()  Show usages  iljah
export class FastApiClientService {
    private client: ClientProxy;

    constructor() {  no usages  iljah
        this.client = ClientProxyFactory.create({
            transport: Transport.RMQ,
            options: {
                urls: ['amqp://localhost:5672'],
                queue: 'fastapi_queue',
                queueOptions: {
                    durable: false,
                },
            },
        });
    }
    async forwardImageUrlsToFastApi(image_urls: ImageUrlsDto) : Promise<string | undefined> {  Show usages  iljah
        return this.client.send<string, ImageUrlsDto>('fastapi_pattern', image_urls).toPromise();
    }

}
```

Figure 13. Nest.js RabbitMQ setup

3.2.6 Cloudflare R2 Storage

Cloudflare R2 Storage is a cloud service used to store all images in the application. It was chosen because it's free for basic use and easy to connect with the backend by using AWS S3 client. Compared to AWS S3, another storage option, R2 provides similar features at a lower cost with 10 GB free per month, which fits the project's budget. Images are uploaded to R2 via its S3 client:

```
async uploadCarRimImages({car, rim}: ImageUrlsDto) {  Show usages
    try{

        const car_key = `${uuidv4()}`
        const car_command = new PutObjectCommand({
            Bucket: this.bucketName,
            Key: `${car_key}`,
            Body: car.buffer,
            ContentType: car.mimetype,
            ACL: 'public-read',
            Metadata: {
                originalName: car.originalname,
            },
        })

        const rim_key = `${uuidv4()}`
        const rim_command = new PutObjectCommand({
            Bucket: this.bucketName,
            Key: `${rim_key}`,
            Body: rim.buffer,
            ContentType: rim.mimetype,
            ACL: 'public-read',
            Metadata: {
                originalName: rim.originalname,
            },
        })

        const upload_car = await this.s3.send(car_command)
        const upload_rim = await this.s3.send(rim_command)

        const car_url = this.getFileUrl(car_key)
        const rim_url = this.getFileUrl(rim_key)

        return {rim: {url: rim_url, key: rim_key }, car: {url: car_url, key: car_key}}

    }catch(err){
        throw new InternalServerErrorException(err);
    }
}

getFileUrl(key: string) {  Show usages
    return `https://pub-217c18ec8e20403390bdf42dcf6fe580.r2.dev/${key}`;
}
```

Figure 14. Upload image to Cloudflare R2 storage

3.2.7 Roboflow

Roboflow is a tool for preparing datasets. It was used to label and organize the car images for training the YOLOv8 model. Roboflow simplifies this process, ensuring the dataset is high-quality and ready for machine learning tasks.

3.2.8 Passport.js for Authentication

Passport.js is a flexible and powerful authentication middleware for Node.js, designed to support a wide variety of authentication methods, such as username-password logins, OAuth, and more. It's widely used due to its modularity and seamless integration with Node.js frameworks.

Passport.js was selected for its simplicity and extensive ecosystem of authentication strategies, making it ideal for implementing secure user authentication. In this project, it was used for session-based authentication, which maintains user state across requests in a straightforward and secure way. Compared to token-based systems like JWT, which require managing token expiration and refresh cycles, Passport.js paired with session storage (e.g., Redis) offered a simpler solution that integrates effortlessly with Nest.js, the project's backend framework. Its strong community support and documentation also made it a reliable choice.

Passport.js was integrated into the Nest.js application to manage user login and session handling. When a user logs in with valid credentials, a session is created and stored in Redis for quick access across requests. Below is an example of how Passport.js was set up with a local strategy for email and password authentication:

```
@Injectable()  Show usages  iljah *
export class LocalStrategy extends PassportStrategy(Strategy){
  constructor(private authService: AuthService) {  no usages  iljah
    super({
      usernameField: 'email',
      passwordField: 'password',
    });
  }
  async validate(email: string, password: string){  no usages  iljah *
    const dto: AuthDto = {
      email: email,
      password: password
    }

    const user = await this.authService.singin(dto)

    if (!user){
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

Figure 15. Passport.js image

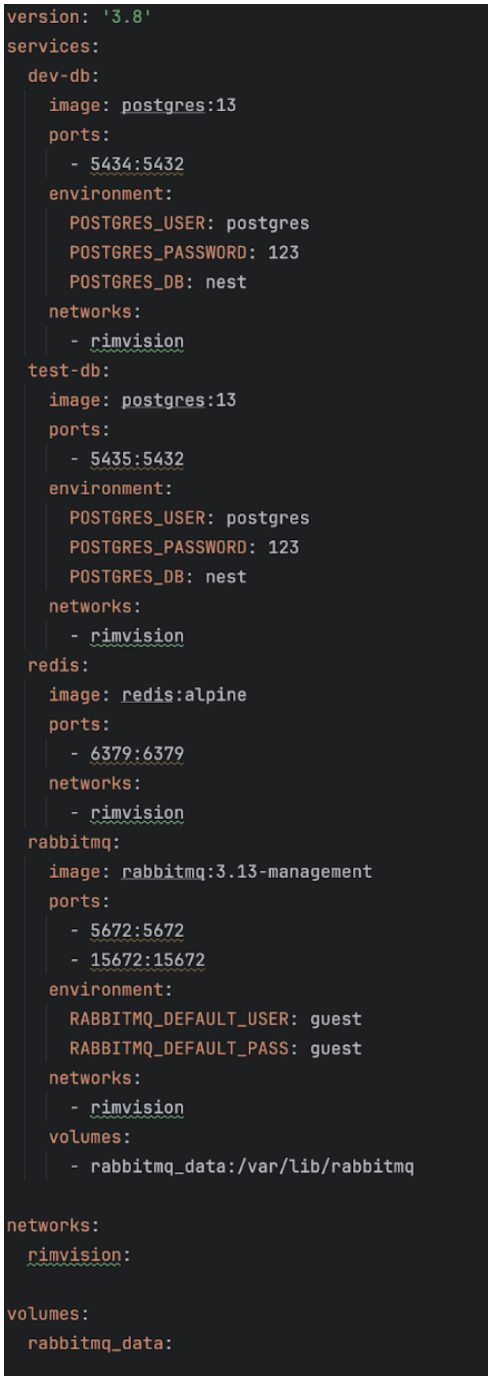
3.2.9 Docker for deployment

Docker is a containerization platform that packages applications and their dependencies into portable, isolated containers, ensuring consistency across different environments.

Docker was selected for its ability to eliminate environment-specific issues, providing a consistent runtime for all services whether in development, testing, or production. Unlike traditional setups, which can suffer from configuration mismatches, Docker's containerized approach simplifies deployment and dependency management.

Docker was used to containerize and deploy essential services, including PostgreSQL (database), Redis (session storage), and RabbitMQ (message queuing).

A docker-compose.yml file was created to define and manage these services, enabling them to be started with a single command:



```yaml
version: '3.8'
services:
  dev-db:
    image: postgres:13
    ports:
      - 5434:5432
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: 123
      POSTGRES_DB: nest
    networks:
      - rimvision
  test-db:
    image: postgres:13
    ports:
      - 5435:5432
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: 123
      POSTGRES_DB: nest
    networks:
      - rimvision
  redis:
    image: redis:alpine
    ports:
      - 6379:6379
    networks:
      - rimvision
  rabbitmq:
    image: rabbitmq:3.13-management
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest
    networks:
      - rimvision
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq

networks:
  rimvision:

volumes:
  rabbitmq_data:
```

Figure 16. docker-compose image

3.2.10 Postman for API testing

Postman is a widely used tool for API development and testing, enabling developers to send requests, analyze responses, and automate testing workflows. In this project, Postman played a crucial role in validating the backend API endpoints, which were built using frameworks like Nest.js and FastAPI.

I have included an image of my API endpoints for testing Nest.js below:

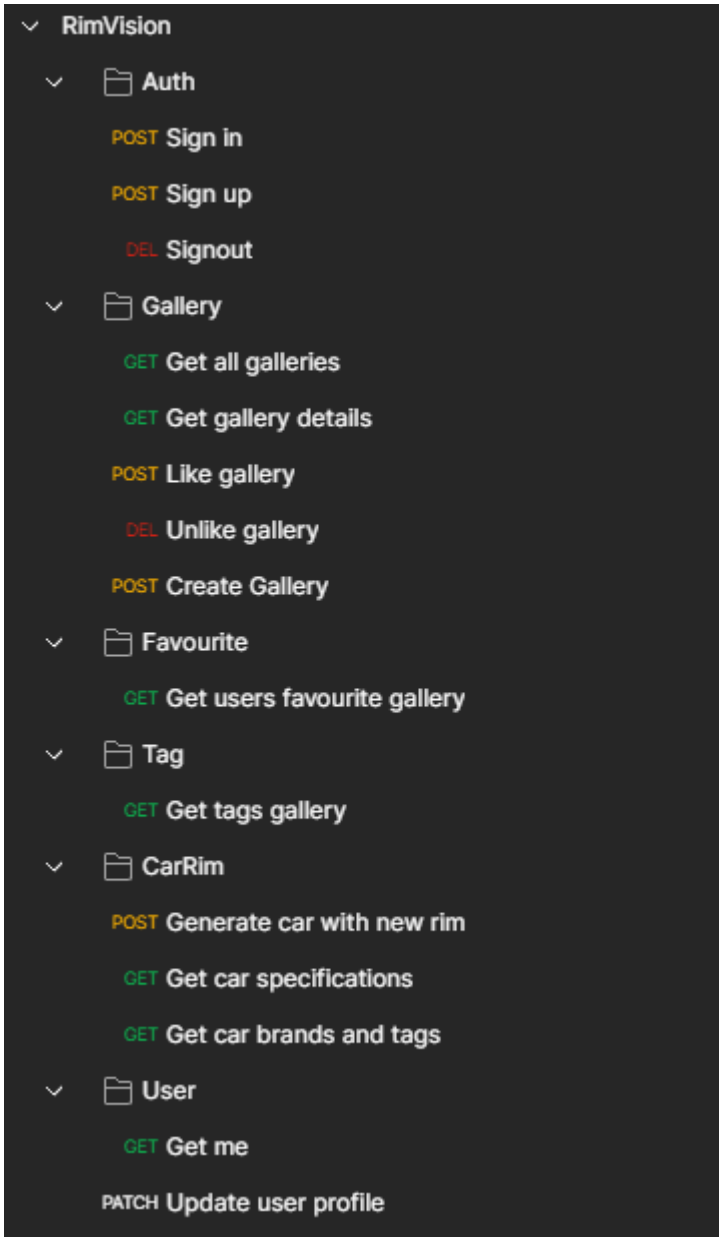


Figure 17. Postman API endpoints image

3.2.11 PostgreSQL for managing data

PostgreSQL is an open-source, relational database management system (RDBMS) known for its robustness, extensibility, and support for complex queries and data integrity. It provides a structured way to store, manage, and retrieve data using SQL, ensuring reliability and scalability for applications.

PostgreSQL was chosen for its ability to efficiently handle structured data and enforce data integrity through features like foreign keys, unique constraints, and transactions. Unlike NoSQL databases, which may sacrifice consistency for flexibility, PostgreSQL's relational

model ensures predictable and reliable data operations, making it ideal for the application's needs.

PostgreSQL was used to store and manage core application data, including user information, gallery content, and user interactions such as favorites. Key tables include:

- **User:** Stores user details such as id, email, password, and username, with unique constraints on email and username to prevent duplicates.
- **Gallery:** Represents gallery items with fields like id, carUrl, views, downloadCount, and userId (foreign key referencing the User table), establishing a one-to-many relationship where a user can own multiple galleries.
- **Favourite:** Tracks user favorites with userId and galleryId (foreign keys referencing User and Gallery tables, respectively), using a composite unique constraint (userId, galleryId) to ensure a user can favorite a gallery only once.

A picture of the database schema diagram can be found below:

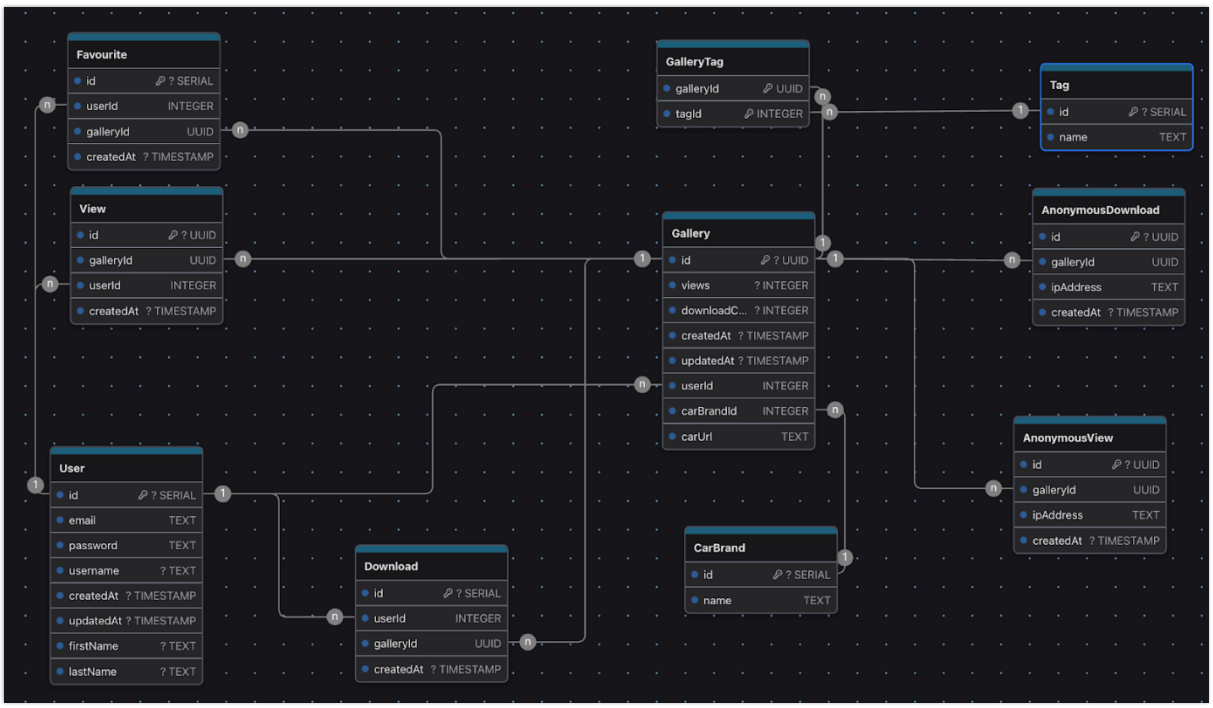


Figure 18. Database schema diagram

4. Practical Development

This section explains the practical development of a web application designed for virtual rim fitment simulation. The application allows users to upload a car photo and a rim photo, overlay the rims onto the car's rims, and view the result. It uses modern technologies like Next.js, Nest.js, FastAPI, Python, YOLOv8, OpenCV, RabbitMQ, and Cloudflare R2 Storage. Here, I describe the application's architecture, the image processing workflow, the features available to users, and the reasons behind the chosen technologies.

4.1 Image Processing Workflow

The image processing workflow is the core feature that allows users to see rims overlaid on their car photos. Here's how it works step-by-step:

1. Image Upload: On the image generation page, a user uploads two photos - one of a car and one of rims. The front-end (Next.js) sends these images to the Nest.js backend.
2. Storage: Nest.js receives and then uploads both images to Cloudflare R2 Storage, a cloud storage service. After uploading, it generates URLs for the car photo and the rim photo.
3. Queueing: Nest.js sends these URLs to a RabbitMQ queue. RabbitMQ acts as a message broker, holding the tasks until FastAPI is ready to process them.
4. Processing: FastAPI listens to the RabbitMQ queue, retrieves the URLs, and downloads the images. Using Python, it processes the images:
    - A custom-trained YOLOv8 model detects the rims on the car photo. I created this model by building a dataset of about 300 car photos from auto24.ee, manually labeling the rims using Roboflow, and training the model to identify rims positions accurately.
    - OpenCV, along with numpy library, overlays the rim photo onto the car photo at the detected rim locations. This step adjusts the rim size and angle to match the car's perspective.
5. Result Storage: After processing, FastAPI uploads the final image (car with overlaid rims) to Cloudflare R2 Storage and generates a new URL.
6. Delivery: FastAPI sends this URL back to Nest.js via RabbitMQ. Nest.js then forwards it to the frontend, where the user sees the end result.

This workflow ensures that image processing happens efficiently and does not slow down the user experience, even with many users uploading images at once.

An illustration of a sequence diagram that illustrates the data flow between my application is shown below:

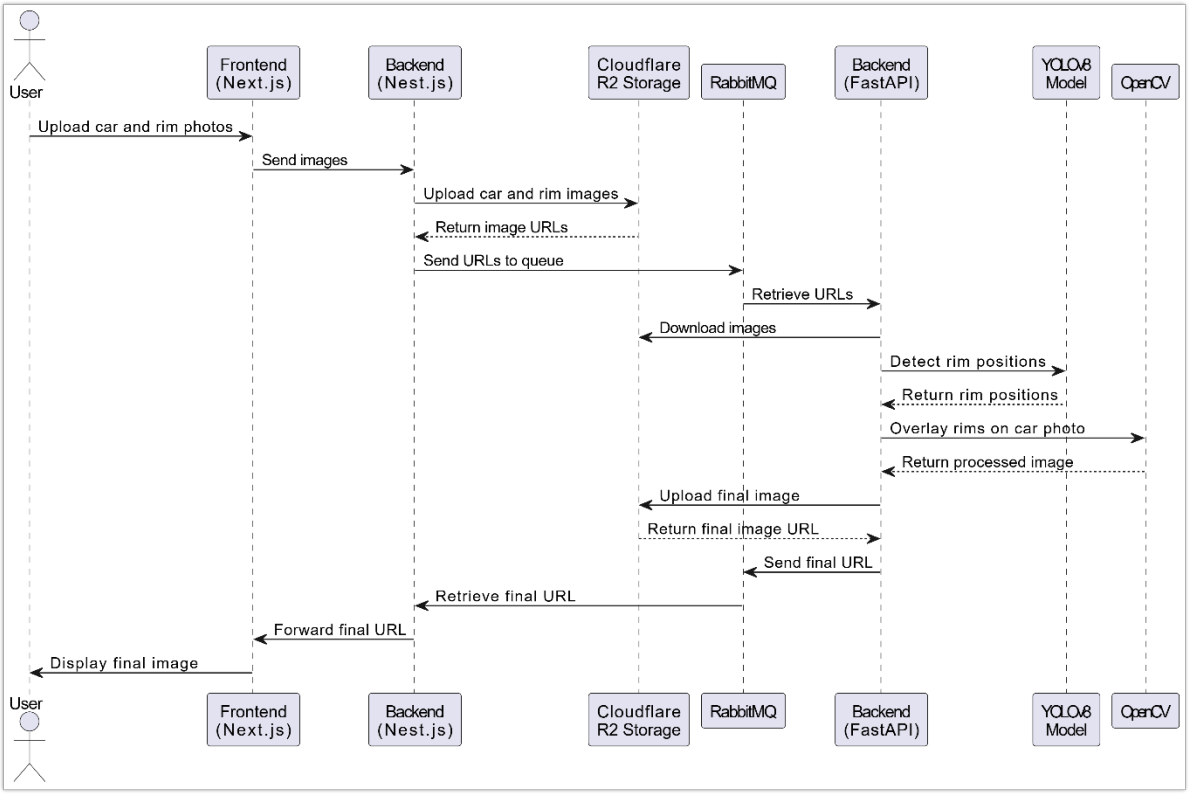


Figure 19. Sequence diagram

4.2 User Features and Authentication

The application has four main pages, each designed to provide a smooth and engaging experience:

Homepage: This page shows a gallery of generated images (cars with overlaid rims) shared by registered users. It acts as a community space where users can get inspiration. The image below shows the homepage:

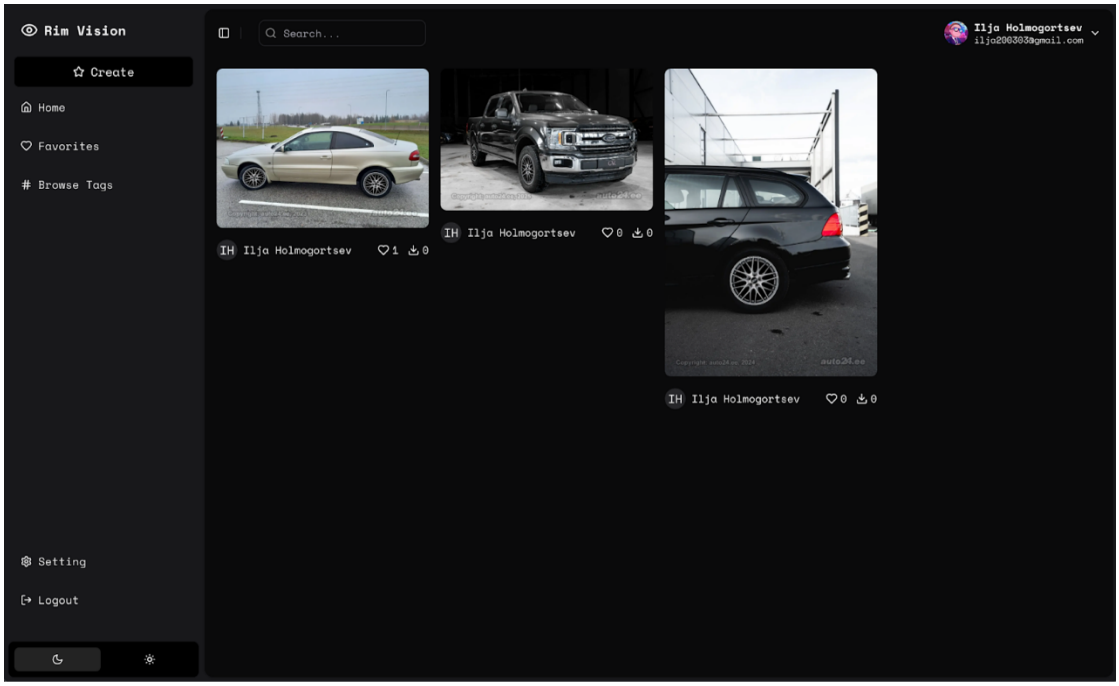


Figure 20. Homepage

Image generation page: Available to all users registered or not this page lets users upload a car photo and a rim photo to create a new image with rim overlaid on the original car's rim. However, only registered users can share their generated images with others and select tags and car brand for generated image. The following is an illustration of car with overlaid rim generation page:

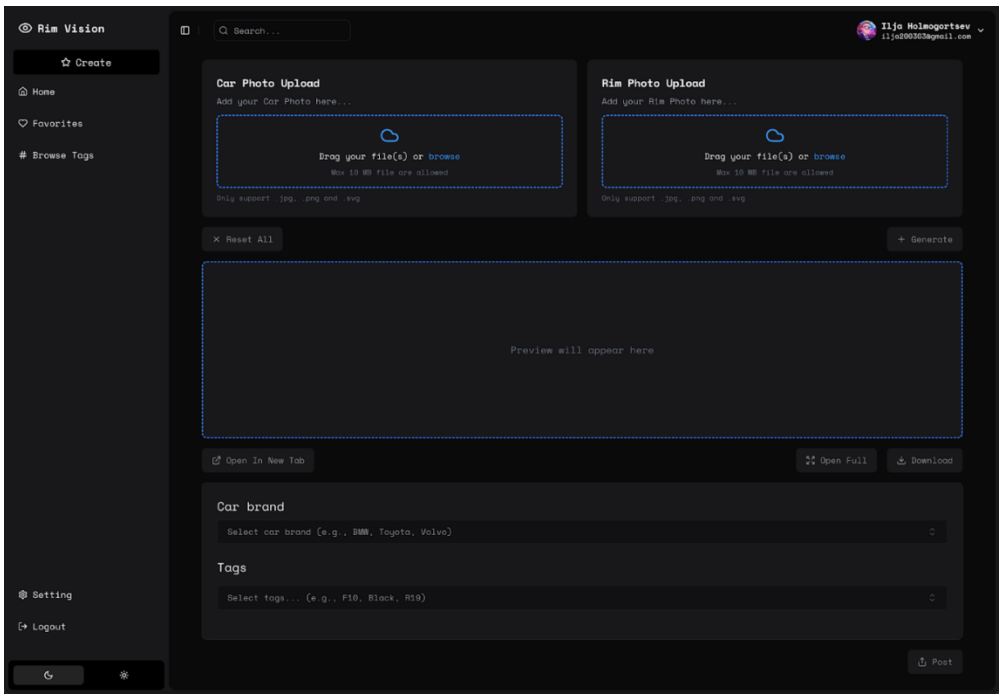


Figure 21. Image generation page

Favorites page: Exclusive to registered users, this page allows them to like images they enjoy and view their saved favorites later.

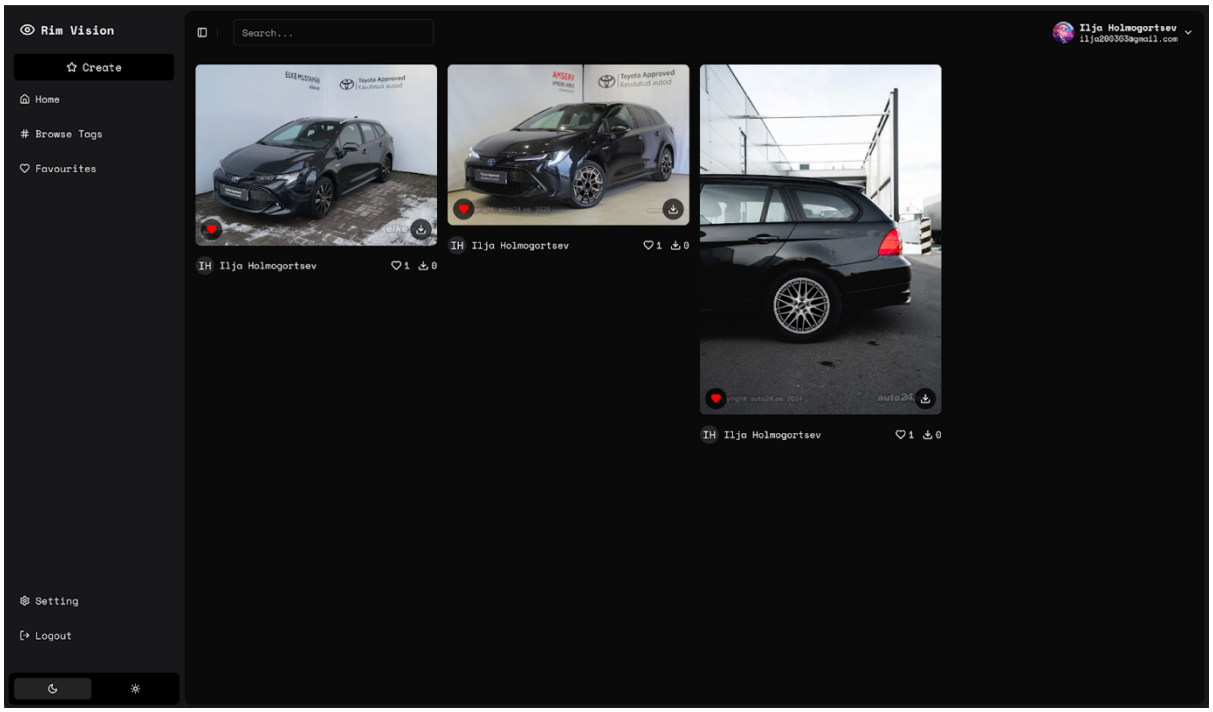


Figure 22. Favorites page

Gallery details modal popup: The purpose of this modal popup is to view gallery data, including tags, views, downloads, and the name of the image creator.

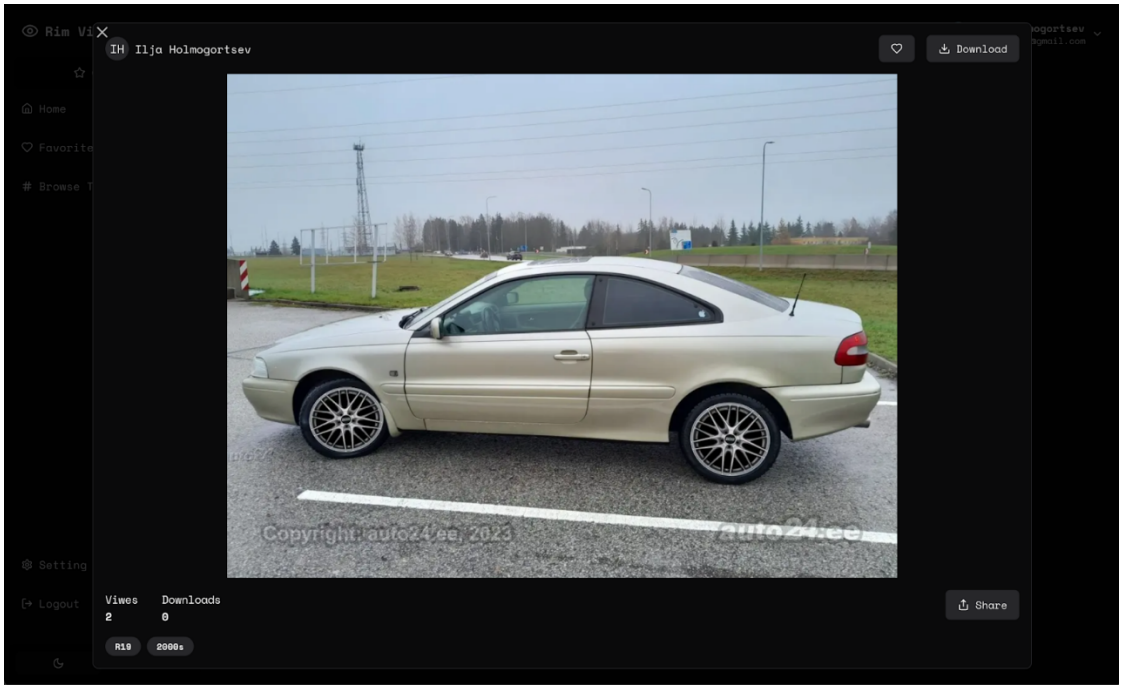


Figure 23. Gallery image dialog

Browse tags: This specific page allows users to browse all available tags that were created by registered users in the Image generation page.

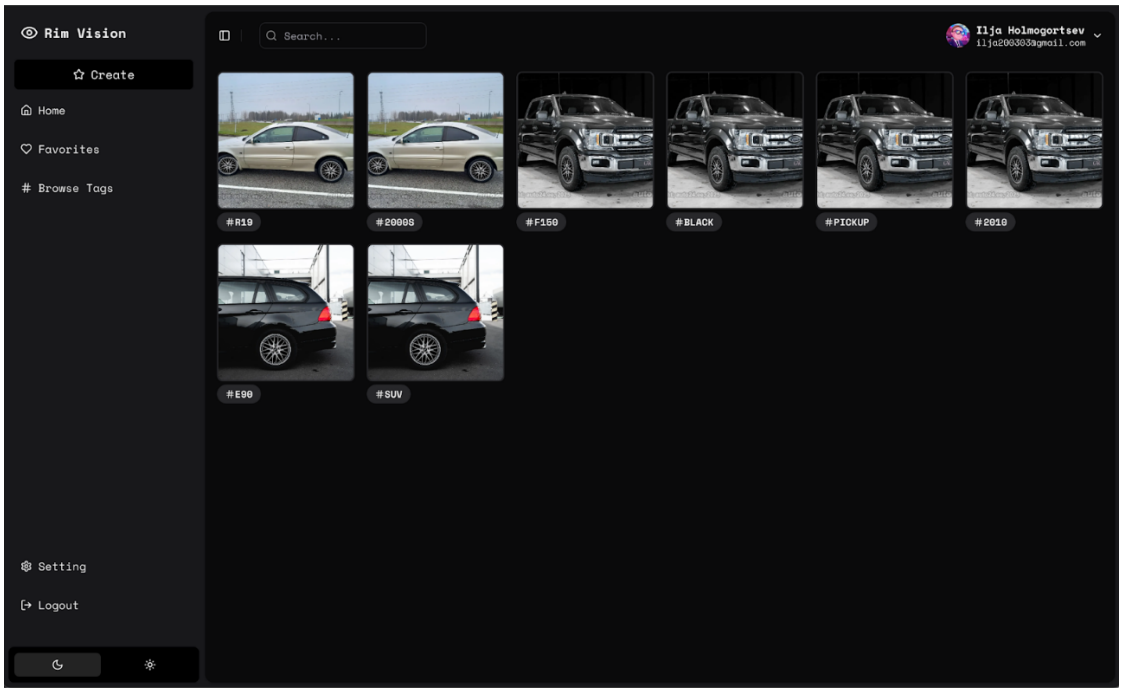


Figure 24. Browse tags page

4.3 Validation of requirements

This section describes the validation of the application in order to determine whether it satisfies the functional and non-functional requirements.

4.3.1 Functional requirements validation

User interface and interaction:

- **Met:** Section 4.2 describes the implementation of an intuitive interface using Next.js, with pages for uploading images, viewing a gallery, managing favorites, and search functionality by tags or car brands.

Image processing:

- **Met:** Section 4.1 details the image processing workflow using YOLOv8 for rim detection and OpenCV for overlaying rims onto car images. Section 3.2.3 covers the training of a custom YOLOv8 model.

User authentication:

- **Met:** Section 4.2 and Section 3.2.8 outline the use of Nest.js for user registration, login, and permissions, restricting certain actions to registered users.

Data management:

- **Met:** Section 3.2.6 and Section 3.2.11 describe the use of Cloudflare R2 Storage for image storage and PostgreSQL for managing other data.

4.3.2 Non-functional requirements validation

Performance:

- **Met:** Section 3.1.1 covers server-side rendering with Next.js for fast page loads, Section 3.2.5 details asynchronous processing with RabbitMQ for efficient task handling, and Section 3.2.2 describes FastAPI and Python's high-performance API capabilities, which enable concurrent image processing without delays.

Scalability:

- **Met:** Section 3.2.5 describes task queuing with RabbitMQ, and Section 3.2.1 highlights Nest.js's modular architecture for scalability.

Usability:

- **Met:** Section 3.1.2 covers the use of Tailwind CSS and Shaden UI for a modern interface, and Section 3.1.1 details Next.js for responsiveness across devices.

Security:

- **Met:** Section 3.2.8 describes secure authentication with Nest.js, and Section 3.2.6 covers secure storage with Cloudflare R2.

Conclusion

This thesis presents a new web application designed to help users see how different rims would look on their cars by simply uploading photos of their vehicle and the rims. What makes this app special is its ability to work with any car and rim, unlike other tools that only support specific models or designs. This flexibility is a big advantage for people with unique or custom cars who want to experiment with their own ideas.

The app was created using a mix of modern tools. The frontend, built with Next.js, makes it fast and easy to use, while the backend, powered by Nest.js and Fast API, handles all the data and images work smoothly. For the image processing, advanced technologies like YOLOv8 and OpenCV were used to accurately place rims onto car photos, ensuring the results look realistic and clear. Together, these tools make the app both effective and simple for users.

Building the app came with some difficulties. One challenge was getting the image processing to work well no matter the lighting or angle of the photos. Another was making sure the app stayed quick even when many people were using it at once. These problems were solved by tweaking the algorithms and improving how the app manages tasks, leading to a smooth and dependable experience for everyone.

The app also includes features that make it fun and useful. Users who sign up can save their designs and share them with others, while everyone can explore a gallery full of designs from the community and generate car images with overlaid rims. This gallery lets users search by car brand or tags, helping them find ideas easily. These additions make the app more than just a tool - they turn it into a place where car fans can connect and get inspired.

For the future, there are many ways to make the app even better. Adding a similar technology like Snapchat's mask which is called augmented reality (AR), that would allow the app to offer a more realistic rim fitting experience. More options, like picking different car parts, could be included too. The app might also connect with other car-related tools or suggest rims that match a user's car automatically. These ideas could take the app to the next level and attract even more users.

In the end, this project has built an original and practical tool that fills a gap for car enthusiasts. It shows how today's technology can make customizing a car easier and more creative. The work done here sets the stage for exciting updates and improvements down the road.

References

Nestjs. Developing Rest API with Nestjs. Retrieved April 20, 2025, from https://docs.nestjs.com/

Nextjs. Creating front-end application with Nextjs. Retrieved April 20, 2025, from https://nextjs.org/docs

Fast API. Building high-performance API with Python. Retrieved April 20, 2025, from https://fastapi.tiangolo.com/

Tailwind. Build modern websites without ever leaving your HTML. Retrieved April 20, 2025, from https://tailwindcss.com/docs

Shadcn. Build your component library. Retrieved April 20, 2025, from https://ui.shadcn.com/docs/installation/next

OpenCV. Learn Computer Vision for Free. Retrieved April 20, 2025, from https://opencv.org/

Roboflow. The Fastest Way to Label Computer Vision Datasets. Retrieved April 20, 2025, from https://roboflow.com/

Docker. Containerizing application. Retrieved April 20, 2025, from https://docs.docker.com/compose/

Passportjs. Simple, unobtrusive authentication for Node.js. Retrieved April 20, 2025, from https://www.passportjs.org/packages/passport-local/

Zustand, A small, fast, and scalable state management solution. Retrieved April 20, 2025, from https://zustand.docs.pmnd.rs/getting-started/introduction

Session authentication. Adding Session-Based Authentication with Nest.js, Retrieved April 20, 2025, from https://www.youtube.com/watch?v=0Mv3-Soi-UQ

RabbitMQ. One broker to queue them all. Retrieved April 20, 2025, from https://www.rabbitmq.com/docs

YOLOv8. A computer vision model architecture for detection, classification, segmentation, and more. Retrieved April 20, 2025, from https://yolov8.com/

TanStack Query. Powerful asynchronous state management for TS/JS, React, Solid, Vue, Svelte and Angular. Retrieved April 20, 2025, from https://tanstack.com/query/latest/docs/framework/react/overview

Redis. Develop with Redis. Retrieved April 20, 2025, from https://redis.io/docs/latest/develop/

Cloudflare R2. Object storage for all your data. Retrieved April 20, 2025, from https://developers.cloudflare.com/r2/get-started/

PostgreSQL. What Is PostgreSQL? Retrieved April 20, 2025, from https://www.postgresql.org/docs/current/intro-whatis.html

Prisma. Next-generation & fully type-safe ORM for NestJS. Retrieved April 20, 2025, from https://www.prisma.io/nestjs