RESIZABLE API

Sometimes, Smaller is Better

An Image Resizer Microservice

---

1. Introduction

The Student Organization and Management System for King's College of the Philippines (SOAMS.KCP) faces challenges with efficiently managing storage due to uncertain image file size uploads. Existing image compressors or size reducers are either bundled as packages or dependencies in codebases, which adds complexity, or are provided as paid APIs, limiting accessibility. This creates a significant barrier for SOAMS.KCP and similar organizations seeking cost-effective and straightforward solutions for image size reduction.

Resizable API is a dedicated microservice designed to address these challenges by offering efficient image resizing functionality. This API resizes image files uploaded to the system, ensuring significant storage optimization. By providing easily accessible API endpoints, Resizable API eliminates the need for integrating additional packages or dependencies into codebases, making it a versatile and developer-friendly solution.

Resizable API will directly support SOAMS.KCP by optimizing its storage usage, reducing operational costs, and improving system efficiency. Additionally, the API will be made freely available to other developers, encouraging its adoption as a lightweight, dependency-free solution for image size reduction across various applications.

This project sought to achieve the following objectives:

1. Gather Requirements for the Image Resizer Microservice
2. Construct the System Design of the Image Resizer Microservice
3. Develop and Test the Image Resizer Microservice
4. Deploy the Image Resizer Microservice

---

2. Methodology

Resizable API will be developed using the **RAD (Rapid Application Development)** methodology, which emphasizes rapid prototyping, iterative development, and user feedback. RAD is particularly suited for projects requiring quick delivery and adaptability to changing requirements [1]. This approach ensures that the API is developed efficiently while aligning with user needs. The methodology consists of four stages, each with specific activities and deliverables.

2.1 Gathering Requirements for the Image Resizer Microservice

This stage focuses on gathering and analyzing requirements to establish a clear foundation for the project. This stage ensures that the development team understands the needs of stakeholders and defines the scope of the API.

Key activities in this stage include the following:

1. Create and conduct a self-survey as a developer and part of the SOAMS system stakeholders.
2. Research existing image compression libraries to determine the best-fit technologies [2].
3. Define the technology stack.

2.2 Designing the Image Resizer Microservice

In this stage, the focus shifts to designing the API and its components based on the gathered requirements. The goal is to create a user-centric design that is both functional and easy to integrate.

Key activities in this stage include the following:

1. Develop flow diagrams to visualize key interactions within the API, such as file upload, compression and file retrieval.
2. Design detailed API specifications, including endpoints (e.g., /resize), request payloads, response formats, and error handling mechanisms.
3. Design a frontend prototype for developer reference documentation to facilitate integration.

2.3 Develop and Test the Image Resizer Microservice

This stage involves the iterative development of the API, focusing on building a functional MVP (Minimum Viable Product) that can be tested and refined iteratively.

Key activities in this stage include the following:

1. Set up the development environment, including version control (e.g., GitHub) to ensure smooth integration and deployment.
2. Build the MVP which includes the API endpoint with core functionality for handling basic image compression.
3. Develop test cases and conduct integration testing to ensure smooth interaction between endpoints and the compression pipeline.
4. Develop the frontend for the developer documentation.

2.4 Deploy the Image Resizer Microservice

The final stage focuses on deploying the API to a live environment and ensuring it performs as expected under real-world conditions.

Key activities in this stage include the following:

1. Deploy the API to a live environment with monitoring tools to track usage.
2. Conduct end-to-end testing in the live environment to confirm that the API functions as expected.
3. Continuously optimize compression algorithms and scalability based on user feedback and performance data [5].

References

[1] Steve McConnell. 1996. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press.

[2] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

[3] OWASP Foundation. 2021. *API Security Top 10*. https://owasp.org/www-project-api-security/

[4] Tom Johnson. 2018. *Documentation as Code*. https://idratherbewriting.com/learnapidoc/

[5] Google Cloud. 2023. *Best Practices for API Design*. https://cloud.google.com/apis/design

[6] Microsoft Azure. 2023. *API Management and Monitoring*. https://azure.microsoft.com/en-us/services/api-management/

3. Results and Discussion

This chapter discusses the results of the development process of the Resizable API, aligning with the objectives and methodology outlined in the previous chapter. Each stage of the Rapid Application Development (RAD) approach—requirement gathering, system design, development and testing, and deployment—is analyzed to assess its effectiveness in achieving the project's goals.

3.1 Gathering Requirements for the Image Resizer Microservice

Before developing the Image Resizer API, a structured approach was taken to define its core requirements. This involved assessing the challenges faced by the SOAMS.KCP system, researching existing image compression libraries, and identifying the best technology for seamless integration.

3.1.1 Create and Conduct a Self-Survey as a Developer and Part of the SOAMS System Stakeholders

To ensure that the Resizable API effectively meets the requirements of SOAMS.KCP, a self-survey was conducted by the developer, who is also a stakeholder in the system. The survey aimed to identify key challenges, technical requirements, API preferences, and security considerations that would shape the development of the microservice.

TODO: Insert the reference on the appendix here showing the self-survey

The survey results highlighted a critical issue in SOAMS.KCP—the unpredictability of image file sizes uploaded by users. Since the system does not currently have an automatic resizing mechanism, storage optimization becomes a challenge. This validates the need for an image resizer that can process images before storing them, ensuring efficient use of resources.

Furthermore, the survey revealed that no existing image compression tools or libraries had been used in the system before. This suggests that the Resizable API must be easy to integrate with minimal additional dependencies.

Based on the survey results, the following functional and non-functional requirements were identified:

Requirement Type	Description
Functional Requirements	**(Define what the API must do)**
Image resizing before storage	The API should accept uploaded images, resize them, and return the resized version before storing.

Support for common image formats	The API must handle commonly used image formats (e.g., PNG, JPEG).
Retain original format	The output image should maintain the same file type as the original upload.
Provide JSON responses	The API should return responses in JSON format.
Stateless processing	The API must not store images, ensuring user privacy.
Non-Functional Requirements	**(Define quality attributes and constraints)**
Performance efficiency	The API should process and return resized images with minimal delay.
Scalability	The API should support multiple concurrent requests efficiently.
Ease of integration	The API should be easily usable with Next.js and other modern frameworks.
Deployment on Vercel	The API should be optimized for deployment on Vercel's serverless environment.

For authentication, the survey indicated no preference for a specific security model, and an open API could also be acceptable if security is not compromised. This highlights the need to balance security and accessibility.

Since Next.js is the primary development framework, the API should be optimized for seamless integration with Next.js-based applications. Additionally, Vercel was identified as the preferred deployment environment, aligning with the goal of serverless and scalable deployment.

A key security requirement was ensuring that the API does not store uploaded images. This aligns with user privacy best practices, reinforcing the decision to implement a stateless processing model where images are resized and immediately returned without storage.

This self-survey provided a clear foundation for the project, ensuring that the Resizable API aligns with the specific needs of SOAMS.KCP while also remaining useful for other potential users.

3.1.2 Research on Existing Image Compression Libraries

Various libraries offer image compression functionalities with different trade-offs in terms of speed, quality, and ease of integration. This section aims to

evaluate existing image compression libraries and determine the best-fit technology for the Resizable API.

To identify the most suitable image compression library, the following criteria were considered:

1. Compression Efficiency: The ability to reduce file size while maintaining image quality.
2. Performance: Speed of processing and memory usage.
3. Format Support: Compatibility with common image formats (JPEG, PNG, etc.).
4. Ease of Integration: Availability of API support and documentation.
5. Customizability: Ability to set compression parameters such as quality and resolution.

The following table summarizes the descriptions, key features, and limitations of different image compression libraries:

Library	**Description**	**Features**	**Limitations**
Sharp	Sharp is a high-performance image processing library for Node.js, built on the libvips library. It is widely used for resizing, cropping, and compressing images efficiently [2].	Fast processing (libvips-based), supports JPEG, PNG, WebP, AVIF, GIF, low memory usage, customizable compression. [2]	Requires native dependencies. [2]
ImageMagick	ImageMagick is a powerful command-line tool for image manipulation, offering extensive support for different operations and formats [3].	Supports 100+ image formats, advanced image processing, highly customizable. [3]	Slower than Sharp, requires more system resources. [3]
MozJPEG	MozJPEG is an advanced JPEG encoder developed by Mozilla to enhance compression while maintaining visual quality [4].	High-quality JPEG compression, reduces artifacts, ideal for web optimization. [4]	Only supports JPEG, slower than Sharp. [4]

TinyPNG API	TinyPNG is a cloud-based image compression service that optimizes PNG and JPEG images effectively [5].	High compression with minimal quality loss, simple API integration, supports transparent PNGs. [5]	Paid API for higher usage, requires an internet connection. [5]

Selecting the appropriate image compression library for the Image Resizer API necessitates a careful evaluation of several key criteria: compression efficiency, performance, format support, ease of integration, and customizability. Below is a discussion of four prominent libraries—Sharp, ImageMagick, MozJPEG, and TinyPNG API—based on these criteria.

Compression Efficiency

Compression efficiency pertains to a library's capability to minimize file size while preserving image quality. MozJPEG is specifically engineered to enhance JPEG compression, effectively reducing file sizes with minimal quality degradation, making it advantageous for web optimization [4]. TinyPNG API excels in compressing PNG images, maintaining transparency and achieving significant size reductions. Sharp offers customizable compression settings across various formats, enabling developers to balance quality and file size according to specific requirements. ImageMagick also provides extensive compression options; however, its real-time processing efficiency is comparatively lower than that of Sharp.

Performance

Performance is critical for applications requiring real-time image processing. Sharp distinguishes itself through its integration with the libvips library, facilitating rapid and memory-efficient image processing. This design choice results in resizing operations that are approximately 4 to 5 times faster than those performed using ImageMagick [6]. In contrast, ImageMagick, while feature-rich, tends to consume more system resources, potentially leading to slower performance in high-demand scenarios. MozJPEG, although proficient in JPEG compression, operates at a slower pace compared to Sharp, particularly when processing large batches of images. The TinyPNG API, being a cloud-based service, introduces latency due to network dependencies, which may affect performance relative to local processing solutions.

Format Support

Comprehensive format support enhances a library's versatility. ImageMagick is notable for its extensive compatibility, supporting over 100 image formats, including widely used ones such as JPEG, PNG, GIF, TIFF, and more [7]. Sharp supports commonly utilized web formats, including JPEG, PNG, WebP, AVIF, and GIF, covering the majority of standard use cases [8]. MozJPEG is specialized for JPEG compression exclusively, limiting its applicability to this

format. The TinyPNG API supports PNG and JPEG formats, which may suffice for basic needs but lacks the breadth offered by ImageMagick and Sharp.

Ease of Integration

The simplicity of integrating a library into existing systems is a vital consideration. Sharp is tailored for Node.js environments, providing a straightforward and well-documented API that facilitates seamless incorporation into modern web applications [8]. ImageMagick, despite its powerful capabilities, relies on command-line operations, which can complicate integration and necessitate additional effort to interface with application code. MozJPEG requires specific configurations to achieve optimal performance, potentially adding complexity to the setup process. The TinyPNG API offers a user-friendly interface; however, its dependence on external cloud services introduces potential reliability concerns and necessitates internet connectivity.

Customizability

Customizability reflects the extent to which a library allows developers to adjust compression parameters to meet specific needs. ImageMagick offers a high degree of customization, enabling fine-tuning across a wide array of image processing tasks. Sharp also provides substantial configurability, allowing precise control over compression levels and processing options [8]. MozJPEG focuses on optimizing JPEG compression but offers limited customization for other formats. The TinyPNG API, being a cloud-based solution, provides minimal customization options, as it employs predefined compression algorithms.

Based on this evaluation, Sharp emerges as the most suitable choice for the Image Resizer API. It offers the best combination of speed, efficiency, and format support while being easy to integrate into a Next.js application. Its libvips-based architecture ensures fast processing with low memory usage, making it highly efficient for real-time image compression. While ImageMagick provides extensive format support and customization, its higher resource consumption and complexity make it less ideal for a lightweight microservice. MozJPEG and TinyPNG API are excellent in specific use cases but lack the versatility required for a general-purpose image resizer.

Overall, this section analyzed multiple image compression libraries to determine the best option for integration into the Resizable API. **Sharp** emerged as the most suitable choice due to its performance, flexibility, and strong community support. Other libraries like ImageMagick and MozJPEG remain viable alternatives depending on specific project needs.

3.2.5. References

[2] Fitzgibbon, L. (2023). *Sharp: High-Performance Image Processing for Node.js*. Available at: https://sharp.pixelplumbing.com/

[3] ImageMagick (2023). *ImageMagick: Convert, Edit, and Compose Images*. Available at: https://imagemagick.org/

[4] Mozilla (2023). *MozJPEG: Improved JPEG Encoding for the Web*. Available at: https://github.com/mozilla/mozjpeg

[5] TinyPNG (2023). *TinyPNG – Compress PNG and JPEG Images*. Available at: https://tinypng.com/

[6] Lovell Fuller, "Sharp - High-performance Node.js image processing," Sharp Documentation, Available: https://sharp.pixelplumbing.com/performance.

[7] ImageMagick Studio LLC, "ImageMagick - Supported Image Formats," Available: https://imagemagick.org/script/formats.php.

[8] Lovell Fuller, "Sharp - High-performance Node.js image processing," GitHub, Available: https://github.com/lovell/sharp.

3.1.3 Defining the Technology Stack

The technology stack for the Resizable API must be optimized for performance, scalability, and ease of integration. Among various web frameworks, Next.js has been selected as the primary framework for API development due to its server-side capabilities, efficient routing system, and seamless integration with modern web technologies.

Next.js, a React-based framework, provides several advantages for building APIs and web applications efficiently. It extends React's capabilities by offering server-side rendering (SSR), static site generation (SSG), and API route handling, all of which enhance performance, security, and developer productivity [1].

Key Benefits of Next.js for the Resizable API

Feature	**Benefit to Resizable API**
API Routes	Enables server-side functions without needing an additional backend server [2].
Server-Side Rendering (SSR)	Ensures fast, dynamic content delivery while reducing client-side processing [3].
Automatic Code Splitting	Improves performance by loading only necessary code per request.
Edge & Serverless Compatibility	Enables deployment to platforms like Vercel and AWS Lambda for scalability.
Static Site Generation (SSG)	Helps in rendering static API documentation efficiently.

Built-in Image Optimization	Reduces image sizes using Next.js's next/image component, ensuring efficient loading.

Next.js provides API routes (/api) that work as backend endpoints. This eliminates the need for a separate Express.js server, making deployment simpler and cost-effective. The Resizable API will leverage Next.js API routes to handle image processing requests.

Next.js is compatible with serverless platforms like Vercel, AWS Lambda, and Cloudflare Workers. The Resizable API will be deployed on Vercel, ensuring automatic scaling based on demand; Edge deployment for faster response times; and Integrated CI/CD for seamless updates.

While Next.js serves as the core framework, it will be integrated with other technologies to ensure a complete and robust image resizer service:

Component	**Technology Used**
Backend Framework	Next.js API Routes
Image Processing	Sharp Library
Deployment	Vercel

In conclusion, Next.js provides a powerful and flexible framework for developing the Resizable API. Its built-in API routing, performance optimizations, and scalability make it an ideal choice. By leveraging its features, the Resizable API will deliver a fast, efficient, and developer-friendly solution for image resizing.

3.3.3. References

[1] Vercel. (2023). Next.js: The React Framework for Production. Available at: https://nextjs.org/
[2] Vercel. (2023). API Routes in Next.js. Available at: https://nextjs.org/docs/api-routes/introduction
[3] Google Developers. (2023). Server-Side Rendering for Performance Optimization. Available at: https://web.dev/rendering-on-the-web/
[4] Fitzgibbon, L. (2023). Sharp: High-Performance Image Processing for Node.js. Available at: https://sharp.pixelplumbing.com/

3.2 System Design of the Image Resizer Microservice

This section presents the design of the Image Resizer Microservice, covering its core processes, API specifications, and developer resources. It includes flow diagrams for image upload, compression, and retrieval, ensuring efficiency and security. The API specifications detail the `/resize` endpoint, request structure, response format, and error handling. Lastly, a frontend prototype provides interactive documentation with API references, request examples, and real-time testing to streamline integration and enhance the developer experience.

3.2.1 Flow Diagrams

The flow diagrams in this section illustrate the structured process of handling image uploads, compression, and retrieval within the Resizable API. These diagrams provide a visual representation of how data moves through the system, ensuring efficiency, reliability, and scalability.

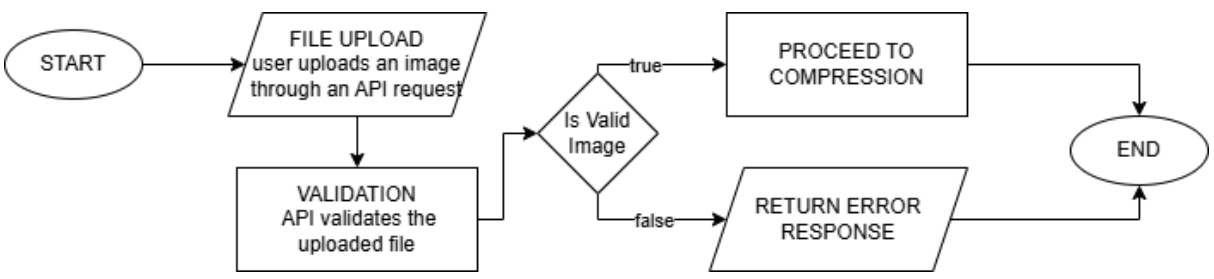
TODO: Insert captions

The Image Resizer Microservice follows a structured workflow consisting of three main processes: file upload, image compression, and file retrieval. The process begins with file upload, where the system validates the image to ensure it meets format and size requirements, preventing the processing of unsupported or excessively large files. Once validated, the image proceeds to the compression stage, where the Sharp library efficiently reduces file size while maintaining visual quality, optimizing the image for better performance and storage efficiency. Finally, the compressed image is retrieved and returned to the user in its original format, ensuring quick access to an optimized version. This structured approach enhances efficiency, security, and overall system performance.

File Upload Flow

The file upload process is designed to validate and manage image submissions before processing. This step ensures that the API only handles supported file types and sizes, preventing unnecessary load on the server.

TODO: Insert captions



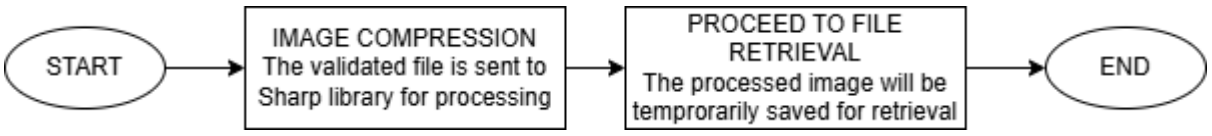
In the file upload process, when a user submits an image through an API request, the system first validates the file based on predefined criteria, such as supported formats like JPEG and PNG. This validation helps prevent errors and ensures compatibility with the compression process. If the uploaded file meets the required specifications, it is temporarily stored for further processing. However, if the file is unsupported or invalid, the API immediately returns an error response, preventing unnecessary processing and maintaining system efficiency.

By implementing validation early in the process, the API prevents the processing of unsupported files, enhancing performance and security.

Image Compression Flow

Once the image passes validation, it proceeds to the compression stage using the Sharp library. This process is essential for reducing file size while maintaining image quality, making it more suitable for web use.

TODO: Insert captions

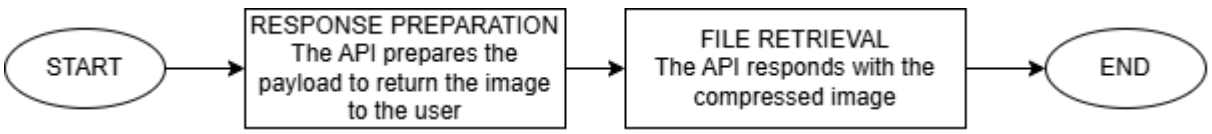


In this stage, the image is sent to the Sharp library for processing. Sharp applies predefined settings to resize and compress the image while preserving its original aspect ratio to prevent distortion. This ensures that the image remains visually accurate while significantly reducing its file size for optimized performance. By efficiently balancing compression and quality, Sharp enhances the image for web use without noticeable degradation. After processing, the optimized image is temporarily stored, making it readily available for retrieval and further use.

File Retrieval Flow

After successful compression, the API returns the optimized image to the user. This step ensures seamless access to processed images without unnecessary delays.

TODO: Insert captions



In this stage, the API retrieves the compressed image from temporary storage and delivers it to the user in its original file format. This ensures a seamless experience, allowing users to access the optimized image quickly without any additional processing on their end. By maintaining the original

format, the API ensures compatibility while providing a more efficient, lightweight version of the image. This ensures users receive a high-quality, compressed image quickly, improving their overall experience.

In this section, the flow diagrams provide a structured breakdown of the Resizable API's core processes. By following these flows, the system can efficiently handle file uploads, ensuring only valid images are processed; Optimize images using Sharp, reducing file sizes while maintaining quality; and Provide users with compressed images, enhancing performance and user experience.

3.2.2 API Specification for Resizable API

The Resizable API is designed to provide a seamless and efficient image resizing service. This section details the API specifications, including the available endpoints, request payload structure, response format, and error handling mechanisms to ensure reliability and security.

Endpoint	Method	Content-Type	Description
`/resize`	`POST`	`multipart/form-data`	Accepts an image file, resizes it based on optional parameters, and returns the optimized version.

The API offers a single endpoint, `/resize`, which processes image uploads by resizing and compressing them based on predefined settings. This endpoint supports multipart form-data requests, allowing users to send image files in formats such as JPG, JPEG, and PNG. The processed image is returned as a downloadable binary stream, ensuring minimal client-side processing.

Parameter	Type	Required	Description
`file`	`File`	**Yes**	The image file to be resized. Supported formats: **JPG, JPEG, PNG**.

The request payload must include the image file, with optional parameters for customizing the output.

Response Headers	
Content-Type:	image/jpeg
Content-Disposition:	attachment; filename="resized-image.jpg"

The API returns the resized image file as a downloadable binary stream, ensuring minimal processing overhead for the client.

Error Code	Description
`400 Bad Request`	Missing or invalid parameters (e.g., unsupported file type).
`415 Unsupported Media Type`	File format not supported.
`500 Internal Server Error`	Unexpected server error during processing.

To maintain robustness, the API incorporates structured error handling. Invalid requests, such as missing parameters or unsupported file formats, return

clear error codes like `400 Bad Request` or `415 Unsupported Media Type`. Additionally, a `500 Internal Server Error` response is triggered if an unexpected issue occurs during processing.

Security and reliability are key considerations in the API design. Features such as CORS handling enable seamless cross-origin integration, while strict file validation prevents malicious uploads. These measures ensure that the API remains efficient, scalable, and secure for developers integrating image compression into their applications.

Overall, the Resizable API provides a secure, efficient, and flexible image resizing service with clear error handling, structured request validation, and robust security measures. This specification ensures a smooth developer experience while maintaining high performance and security.

3.2.3 Frontend Prototype for Developer Reference Documentation

To facilitate seamless integration of the Resizable API, a frontend prototype has been developed to serve as an accessible and structured reference for developers. This prototype acts as a developer documentation platform, ensuring that developers can quickly understand how to use the API, explore its endpoints, and test its functionality in real time. By providing a clear and interactive experience, the documentation helps minimize friction during implementation and accelerates adoption.

The developer documentation is designed with essential features that enhance usability and efficiency.

3.2.3.1 Interactive API Reference

A well-structured interface presents available API endpoints, such as /resize, along with detailed explanations of their functionality. Each endpoint includes information about required parameters, supported image formats, and expected responses. To make integration more seamless, the documentation provides code snippets in multiple programming languages, including JavaScript, Python, and cURL, enabling developers to quickly reference and adapt examples to their own projects.

3.2.3.2 Request and Response Examples

To ensure developers correctly format their API requests, the documentation includes practical examples showcasing different scenarios. These examples highlight successful requests, expected JSON responses, and error-handling cases. By illustrating possible interactions with the API, developers gain a better understanding of what to expect when implementing it within their applications.

3.2.3.3 Real-Time API Testing

An interactive API playground allows developers to test the Image Resizer directly from the documentation interface. This feature enables users to upload raw images and receive the processed output in real time.

Overall, the Resizable API documentation serves as a comprehensive and interactive guide for developers, providing a structured reference, real-time testing, and clear instructional content. By integrating modern design principles and developer-friendly features, the documentation streamlines the learning process and ensures that the API is accessible, efficient, and easy to implement.

3.3 Development and Testing of the Image Resizer Microservice

The development and testing phase of the Image Resizer Microservice focuses on building a functional Minimum Viable Product (MVP) with core image compression capabilities. This phase also involves rigorous testing and the creation of well-documented API usage guidelines to ensure smooth integration and usability.

3.3.1 Setting Up the Development Environment

A well-structured development environment is essential for seamless integration and deployment. The setup process includes selecting the required technologies and tools, installing dependencies, and configuring environment variables.

To develop the Image Resizer Microservice, several key technologies and tools are utilized. Next.js serves as the framework for building the server-side rendered application. Sharp, a high-performance image processing library, is used for compression and resizing tasks. Git and GitHub are implemented for version control, enabling tracking of changes and collaboration. Postman facilitates API testing, while Jest and Supertest provide frameworks for unit and integration testing.

To begin, Node.js and npm must be installed. Once these are set up, a new Next.js project is created. Next, the necessary dependencies are installed including the Sharp library.

Finally, a Version control is established with Git, and the repository is linked to GitHub which is located at https://github.com/codexcancerion/resizable.

3.3.2 Building the MVP

The MVP of the Image Resizer Microservice focuses on implementing fundamental image compression functionalities. These include accepting image uploads via API requests, validating file type and size, compressing and resizing images using Sharp, and returning the processed image to the client.

The primary API endpoint for image compression is the /api/resize endpoint, which allows users to upload images, and receive the optimized image as a response.

The /api/resize endpoint is responsible for processing image uploads, validating them, compressing the images using Sharp, and returning the optimized output. This endpoint is optimized for efficiency and scalability.

3.3.4 Developing Test Cases and Conducting Integration Testing

A comprehensive testing strategy ensures that the Image Resizer API functions correctly across different scenarios. This involves developing structured

test cases to validate the API's behavior, followed by integration testing using Jest and Supertest to confirm smooth interaction between endpoints and the image compression pipeline.

The following test cases validate API functionality:

Test Case ID	**Description**	**Input**	**Expected Output**
TC-001	Upload a valid PNG image	PNG file (1-5 MB)	Resized PNG image returned
TC-002	Upload a valid JPG image	JPG file (1-5 MB)	Resized JPG image returned
TC-003	Upload a large PNG image	PNG file (5-10 MB)	Resized PNG image returned
TC-004	Upload a large JPG image	JPG file (5-10 MB)	Resized JPG image returned
TC-005	Upload an unsupported file format	PDF file	Error message returned
TC-006	Upload without an image	No file uploaded	Error message returned

Integration tests are implemented using Jest and Supertest to verify that API responses align with expected outcomes. These tests help ensure that the compression pipeline remains stable and functional under different conditions.

TODO: Include test result screenshots here

After thorough testing, the Image Resizer API demonstrates strong performance and reliability. Additionally, error handling mechanisms are robust, ensuring that unsupported file formats and missing files return appropriate error messages. The API efficiently processes PNG and JPG images. These findings confirm that the Image Resizer API is ready for deployment, with its functionality, performance, and error handling thoroughly validated.

3.3.5 Developing the Frontend for Developer Documentation

The frontend for the Image Resizer API documentation is designed to provide a structured reference while enabling real-time testing. Built with Next.js, Hero UI, and Tailwind CSS, it ensures clear navigation, interactive examples, and an intuitive layout.

TODO: Include the frontend screenshots here

The documentation presents the /api/resize endpoint with its method, parameters, and expected responses. A card-based layout dynamically renders this information, making it easy to update and expand. Each section clearly explains accepted file types, and response formats.

To simplify integration, sample requests in JavaScript, Python, and cURL are displayed in a structured format. JSON responses are shown alongside, helping developers understand what to expect. The layout uses <pre> and <code> blocks with Tailwind styling for readability without external libraries.

A built-in testing tool lets users upload an image and see the processed result instantly. The frontend sends a fetch request to /api/resize, and the response is displayed dynamically. Basic input validation ensures only images are accepted.

This documentation goes beyond static references by combining structured API details, real-time testing, and clear examples. Using Next.js for scalability, Hero UI for clean design, and Tailwind CSS for responsiveness, it ensures a smooth developer experience.

3.4. Deploying the Image Resizer Microservice

Deployment is the final step in ensuring the API is production-ready, making it accessible for real-world use. This phase involves setting up hosting, running live tests, and implementing monitoring tools to track performance and optimize efficiency.

3.4.1 Deploying to a Live Environment

Choosing the right hosting platform is essential for maintaining performance and stability. Since the API is built with Next.js, Vercel is the most suitable option due to its seamless integration, automatic optimizations, and built-in scaling. Once deployed, the API is hosted on a public URL, making it accessible for developers to integrate into their projects.

Before deployment, optimizations are applied to the codebase to ensure efficiency. Unnecessary dependencies are removed, and the image processing logic is refined to handle requests more effectively. With these improvements, the API is ready for production.

TODO: Insert the successful deployment here and the live production preview

3.4.2 End-to-End Testing in Production

After deployment, thorough testing is conducted to validate the API's functionality in a live setting. Using Postman, test requests are sent to the

/api/resize endpoint, uploading images. The responses are checked to confirm that the resized images are returned correctly.

TODO: Insert the successful postman test here

Error handling is also tested by submitting invalid requests, such as missing files or unsupported formats, to ensure the API responds with appropriate error messages. Performance tests are carried out by processing images of different sizes and formats.

3.4.3 Continuous Monitoring and Optimization

To maintain reliability, Vercel Analytics is used to track API performance, monitoring request counts and response times. This helps identify any inefficiencies or potential bottlenecks.

Optimization is an ongoing process, with adjustments made to the image compression settings to balance file size reduction and quality. Vercel's automatic scaling ensures the API remains responsive, even during traffic spikes.

Deploying the Image Resizer API marks the transition from development to real-world use. With continuous monitoring and optimizations, it remains efficient and scalable, providing developers with a reliable tool for image compression. This final phase ensures that the API is not only functional but also well-equipped to handle diverse use cases seamlessly.