Developing a Full-Stack Web Application

Daniel Isaksson

Nackademin

JAVA23_EXAM: Examensarbete

May 23, 2025

Daniel Isaksson
Nackademin

Introduction 3
System overview 5
Technologies used 8
Backend 8
Frontend 9
Testing 10
Results 11
Backend 11
Database 12
Frontend 13
Image Upload and Cloud Storage 14
Testing 15
Deployment 16
Conclusions 20
Attachments 22

Daniel Isaksson
Nackademin

Introduction

This project explores the process of building and deploying a full-stack web application using Java. The purpose is to develop a website for a client — a professional tattoo artist — that includes both a customer-facing site and an admin dashboard. The application aims to streamline the artist's interactions with customers while allowing easy management of bookings and website content.

The project can be found here:
Github repo
Customer facing website

The customer-facing side of the application will provide general information, available flash images, a booking request form, a portfolio, and general information about the artist. The admin dashboard will enable the artist to manage content on the customer facing site and the booking of customers.

Below is the submitted requirements specification:

Pages to serve customers:

- Home page
  - Display latest news and general information.
- About me
  - Display information about the artist.
- Frequently asked questions (FAQ)
  - Dynamically display FAQ that the artist can update.
- Calendar with available hours for booking
  - Dynamically display available days and hours.
  - Show dates one month at a time.
- Booking form
  - Form that sends email to artist with information provided by the form.
- Flash images
  - Dynamically display available flash images that the artist has uploaded.
- Portfolio from Instagram
  - Dynamically display Instagram content from URL that the artist can change.
- Shop page
  - Display products that customers can buy by contacting the artist.

Daniel Isaksson
Nackademin

Admin functionality:

- Manage bookable dates.
    - Set a date as standard, drop in, or touch up date.
- Update the latest new section.
- Change the Instagram portfolio URL.
- Add or update FAQ content.
    - Create or delete FAQ.
    - Change order that FAQs are displayed on the customer page.
- Book customers manually on any date.
    - Manage details (price, notes, image) of existing booking.
- Create and manage image categories.
- Upload and manage flash images.
    - Choose category for uploaded flash image.
- Create products with titles, information, price, and an image.

Daniel Isaksson
Nackademin

System overview

The developed web application is a full-stack system designed to meet the needs of a tattoo artist for both internal scheduling management and self-promotion through images and a calendar with available dates. The system is divided into two primary components: a customer-facing website and an administrative dashboard. These components share a common backend built using Java and Spring Boot, which handles business logic, data management, and communication with external services.

The frontend is constructed with Thymeleaf. JavaScript is used to enhance interactivity on the client side, and CSS is applied for layout and visual styling.

Key elements of the system include:

User Interface (UI)

Rendered using Thymeleaf templates and styled with CSS, the UI provides distinct views for both customers and the administrator.

REST Controllers

Spring Boot REST controllers handle routing, form submissions, and serve dynamic content to the frontend.

Database

A relational Database (PostgreSQL) stores data related to bookings, FAQs, flash images, news updates, and other content managed through the admin dashboard.

Email Service

Integrated through a controller to send booking request emails directly to the artist based on customer form submissions.

Image storage

Flash images and tattoo images are uploaded by the administrator and stored in an Amazon S3 bucket. These images are distributed through Amazon CloudFront to ensure fast, scalable delivery.

Security

Authentication and authorization mechanisms are implemented to restrict access to the admin section, ensuring that only the artist can make changes to content, bookable dates, and bookings.

Daniel Isaksson
Nackademin

The database schema was developed using a code-first strategy with Spring Data JPA. In this approach, the domain model was defined directly in Java code through annotated entity classes, and the corresponding database schema was automatically generated by the persistence framework.

This method ensures that the data structures and entity relationships were consistently aligned with the business logic layer. Code-first development also allowed for rapid iteration of the data model during the early phases of the project, as changes could be easily introduced and deployed without manually altering database scripts.

To maintain stability and prevent unintended schema drift in production environments, the project can later benefit from integrating a schema migration tool such as Flyway or Liquibase, ensuring version-controlled evolution of the database. Although migrations were not a primary concern during initial development, awareness of this aspect is crucial for scaling and maintaining the application long-term.

Key components:

- Customer Management:
  - Customer entities hold personal information such as phone, email, and Instagram handle.
  - Each customer can have multiple associated Booking records.
- Booking System:
  - A Booking includes information about the appointment date, pricing, deposit status and whether it is a touch-up session.
  - Bookings can reference a previous booking, allowing for linked appointments (e.g., touch-ups following initial sessions).
  - Bookings are optionally associated with a TattooImage uploaded for that session.
- Scheduling Availability:
  - BookableDate and BookableHour represent available booking slots.
  - Each date holds multiple hours, each of which can be individually booked.
  - Dates can be marked as dropIn or touchUp.
- Image management:
  - FlashImage and TattooImage represent general and customer-specific tattoo images, respectively.
  - Images are categorized through ImageCategory.
- Product Catalog:

    - Product entities are linked to a ProductCategory, enabling product browsing and purchases.
- Content Management:
    - CustomerPageText manages dynamic content displayed on the customer-facing website, divided into sections and pages.
    - InstagramEmbed entities store a link to an Instagram post that allows the system to display embedded Instagram media content.

Daniel Isaksson
Nackademin

Technologies used

Backend

A backend's primary purpose is to manage the business logic of an application, handle data processing, and serve dynamic content to the frontend. It is responsible for processing user input, interacting with the database, securing access to restricted resources, and communicating with external services like email and cloud storage.

Java

Java is a high-level, general-purpose, memory-safe, object-oriented programming language. It is widely used in enterprise and web development due to its portability, scalability, and strong ecosystem. In this project, Java forms the core of the backend logic and powers the Spring Boot framework.
It has been the primary language used during my studies at Nackademin.

Spring Boot

Spring boot is a framework that simplifies the development of production-ready Java applications. It provides built in configurations, dependency injection, and support for RESTful APIs. In this project, Spring Boot is used to create REST controllers, manage application configuration, and integrate with the database and security layers.

Spring Data JPA

Spring Data JPA simplifies database interactions by providing an abstraction layer over standard Java Persistence API (JPA). It allows the use of repository interfaces for create, read, update, delete (CRUD) operations without writing boilerplate SQL code.

Spring Security

Spring Security was used to secure the admin dashboard, Spring Security handles authentication and authorization within the application. It ensures that only authorized users can access and modify protected resources.

JavaMailSender (Spring Email)

JavaMailSender was Integrated into the backend to send emails when customers submit booking requests through the booking request form. This component makes it easy to send HTML or plain-text messages directly from the server.

Daniel Isaksson
Nackademin

Frontend

The frontend is responsible for presenting data to the user, handling user interactions, and sending requests to the backend. It ensures that the application is visually appealing, functional, and user friendly. In this project, the frontend is designed to provide an interactive experience for both customers and the tattoo artist.

Thymeleaf

Thymeleaf is a modern server-side Java template engine that integrates seamlessly with Spring Boot. It allows the backend to dynamically generate HTML pages with content from the server, making it ideal for rendering data-driven web pages. One key advantage of Thymeleaf is its natural templating ability: the HTML remains valid and viewable in browsers even before server-side processing, which improves maintainability and collaboration between developers and designers.

In this project, Thymeleaf is used extensively for rendering both customer-facing pages and the admin dashboard. It supports dynamic data binding and modular template construction through reusable fragments.

Thymeleaf's expression language and fragment system are leveraged to modularize layouts, dynamically populate data elements, and maintain a consistent structure across the application's pages. These capabilities help create a scalable and maintainable frontend architecture.

While Thymeleaf provided a server-rendered UI suited to the project's needs, client-side frameworks like React or Angular are more commonly used today and could have been a consideration, if time was not a factor.

HTML5

HTML5 is the standard markup language for creating web pages. It is used to structure the content of the application, including headings, paragraphs, links, forms, and multimedia. HTML5 ensures that the content is presented in a semantically correct way and is compatible with modern web standards.

CSS3

Cascading Style Sheets (CSS) is used to style the HTML content, defining the layout, colors, fonts, and overall design. CSS3 allows for advanced styling features like animations, grid layouts, and responsive design. It ensures that the application looks good across a range of devices and screen sizes.

Daniel Isaksson
Nackademin

JavaScript

JavaScript is a client-side scripting language used to add interactivity and dynamic behavior to web pages. It allows for features like client-side validation, asynchronous data loading, and interactive UI elements. In this project, JavaScript enhances the user experience by handling calendar interactions, image modals and dynamic content loading.

Bootstrap

Bootstrap is a popular frontend framework that provides pre-designed CSS and JavaScript components for building responsive web pages. It simplified the process of creating a visually appealing and mobile-friendly UI. In this project, Bootstrap is used for layout, navigation, and form design, ensuring the application is responsive and works well on both desktop and mobile devices.

Testing

Testing ensures the correctness, reliability, and maintainability of the application's backend logic. In this project, testing was performed using JUnit 5 together with Spring Boot's integrated testing framework.

An in-memory database was used to isolate persistence tests, ensuring that database operations could be verified without affecting production data. Service layer mocked external dependencies, allowing business logic to be validated independently. This approach enabled both unit and integration testing across the system, contributing to overall application robustness.

Daniel Isaksson
Nackademin

Results

Backend

The backend of the platform is built with a clear separation of concerns. It uses a layered architecture composed of entities, services, and controllers. Each plays a distinct role in handling application logic, data management, and request processing.

Entity Layer

The entity layer defines the core data models representing the database structure. Each entity is mapped to a table using JPA annotations. Relationships between entities, such as associations between products and categories, are explicitly declared and managed through proper cascading rules. This setup ensures data consistency and enables complex object relationships to be handled automatically by the Object-Relational Mapping (ORM).

Entities are designed with immutability and builder patterns (via Lombok) to simplify object creation and management.

Service layer

The service layer encapsulates the business logic of the application. It acts as an intermediary between the controllers and the data repositories, ensuring that operations are performed in a transactional and controlled manner.

Key responsibilities of the service layer include:

- Data management: Creating, updating, and deleting entities based on business rules.
- Validation: Checking the integrity and completeness of inputs before processing.
- Integration: Handling interactions with external systems, such as the file storage service that integrates with Amazon S3.
- Error handling: Catching and managing exceptions to maintain application stability.
- DTO conversion: Transforming entities into Data Transfer Objects (DTOs) when lightweight representations or obscuring of object data is needed for view models.

Transactional boundaries are defined to maintain data integrity across operations that involve multiple steps.

Daniel Isaksson
Nackademin

Controller Layer

The controller layer is responsible for handling HTTP requests and returning appropriate responses for views. It routes requests to the appropriate service methods and prepares data for the frontend.

Controllers are organized logically (e.g., "`/admin/products`" for administrative product management) and enforce security restrictions through method-level authorization (`@Preauthorize`) in cases where it is needed. Only users with appropriate roles are permitted to perform sensitive operations.

Controller actions include:

- Managing input data from users and sending that data to relevant service methods.
    - CRUD operations for the administrator.
    - Form data or category selections from visitors.
- Handling multipart file uploads.
- Providing feedback to the administrator (e.g., success or error messages) through the view model.
- Binding HTTP request parameters to method inputs in a clean and readable way.

Views are populated with model attributes to display dynamic data, ensuring a smooth interaction between the backend and frontend components.

Additional Considerations

- Security: Administrative routes are protected by role-based access control.
- File management: Media files are stored externally on S3 and managed through create and delete operations.
- Code quality: The use of Lombok for boilerplate reduction, builder patterns for object creation, and stream operations contributes to a clean, maintainable codebase.
- Scalability: The modular structure enables easy expansion of features, such as adding new entity types or new services.

Database

The application uses a PostgreSQL relational database to manage persistent data. During development, a local PostgreSQL database was run inside a Docker container using

Daniel Isaksson
Nackademin

Docker Desktop, enabling easy setup, isolation, and teardown of the database environment without installing PostgreSQL directly on my computer.

For production deployment, the database is hosted on AWS using Amazon Relational Database Service (RDS) for PostgreSQL. AWS RDS provides managed database hosting, offering benefits such as automated backups, scaling, high availability, and improved security through built-in encryption and VPC integration.

Connection to the database was handled through standard Spring Boot JPA configurations, with Hibernate as the ORM tool.

Frontend

Thymeleaf was integral in building the dynamic user interfaces for both the public website and the administrative dashboard:

Customer pages

Dynamic web pages such as the monthly booking calendar and product catalog were constructed using Thymeleaf. Fragments like the calendar (th:fragment=”calendar”) were inserted into layout templates (th:replace), promoting code reuse and design consistency. Customer-facing elements – including available booking dates, times, and products – were dynamically populated using Thymeleaf expressions like “th:text”, “th:each”, and “th:attr”.

Admin dashboard

The admin panel used Thymeleaf to render real-time data views and manage user interaction. For example, management tables for bookings were built dynamically (<table th:each=”booking: ${bookingsAtDate}”>), and forms for customer lookups were bound to backend models. Conditional displays using th:if adapted the interface based on model content and admin actions.

Dynamic Calendar and Booking Management

Thymeleaf’s expression capabilities (#numbers.sequence, #temporals.format) were employed to generate dynamic booking calendars. Days were styled differently depending on booking availability through conditional class attributes (th:classappend), visually distinguishing between available, fully booked, drop in, and touch-up session slots.

Reusable Layouts with Fragments

Common components such as navigation bars, headers, and modals were modularized using Thymeleaf fragments. Examples include:

- ~{layouts/customer-layout :: nav-bar} for customer pages.
- ~{layouts/admin-layout :: header} for the admin panel.
- ~{layouts/image-modal :: image-modal} for modal displays.

Client-side Enhancements

Although Thymeleaf handled server-side rendering. Additional interactivity was provided with lightweight JavaScript enhancements, such as tooltip displays for available hours (show-hours-tooltip-on-date-mouseover.js) and modal previews for product images (display-image-modal.js).

Styling and Responsiveness

Thymeleaf-generated pages were styled with Bootstrap and custom CSS to ensure responsive design across desktop and mobile devices. Data-driven elements like calendars and tables adapted dynamically to different screen sizes, preserving usability and layout consistency.

Image Upload and Cloud Storage

To enable dynamic image management within the application – such as flash images, tattoo images, and product photos – an image upload and storage service was integrated using Amazon S3 and Amazon CloudFront.

The S3ImageService class handles all interactions with AWS cloud storage, providing functionality to upload, resize, and delete images. Uploaded images are served to end-users via a CloudFront distribution, ensuring low latency and high availability.

Key features

1. Uploading Images to Amazon S3:
    - The artist (via the admin dashboard) can upload images to specific folders in the S3 bucket.
    - A unique filename is generated for each upload by combining a UUID with the original filename to avoid naming collisions.
    - Large images (over 1MB) are automatically resized before upload to optimize performance and reduce storage costs. The resizing is performed using the Thumbnailator library, reducing the image dimensions by 50% while maintaining quality.

- Metadata such as cache control headers (public, max-age=31536000) is set to leverage browser caching, improving page load times for returning users.
- After uploading, a public URL to the image is generated using the CloudFront domain.

2. Image resizing Logic:
    - If an uploaded image exceeds 1MB, the system reads it as a BufferedImage, resizes it to half its original dimensions and converts it to a JPEG format before sending it to S3.
    - This resizing is done in-memory using ByteArrayInputStream and ByteArrayOutputStream, avoiding the need for temporary files on the server.
3. Deleting single and Multiple Images:
    - The system provides methods to delete either a single image or a batch of images from S3.
    - When deleting, the CloudFront URL is stripped to retrieve the actual S3 key.
    - Batch deletions are optimized by grouping multiple objects into a single DeleteObjectsRequest, reducing the number of network calls.
4. AWS Integration and Configuration:
    - The S3Client is configured in a dedicated S3Config class using credentials and region settings injected from application properties.
    - Credentials are securely loaded and managed through Spring's configuration system.
    - By decoupling the S3 configuration into its own bean, the application maintains clean separation of concerns and can easily switch to different environments with minimal changes.

This design ensures that the application is highly scalable, secure, and efficient in handling media assets, which are critical for a visually driven business owner like a tattoo artist.

Testing

Testing was a critical part of ensuring the reliability and stability of the application. While I admittedly did not practice strict "test-first development", testing still became a major part of the project. The primary focus was on backend service testing, validating both individual functionalities and the integration of components across the system.

Unit and integration tests were implemented using JUnit 5 combined with Spring Boot's testing framework. Services were tested in an environment configured with an in-memory

database, ensuring that database interactions could be verified without affecting real production data. A separate test configuration profile was used to isolate test behavior from the main application settings.

Key areas tested

- Data Persistence
  - Save, update, and delete operations were thoroughly tested to ensure that entity management through repositories functioned correctly. This included cases where an entity had connections to other entities.
- Business Logic Validation
  - Operations, such as filtering bookings by date, counting associated tattoo images, and handling linked entities like touch-up appointments, were tested under a variety of input conditions.
- Pagination and Sorting
  - Tests confirmed that pagination services returned items in the correct order and quantity, especially where domain-specific logic was applied (e.g., ordering tattoo images by latest booking date).
- Edge Cases and Robustness
  - Special scenarios, such as handling nonexistent data or excluding irrelevant results based on category associations, were tested to prevent unexpected application behavior.
- Mocking External Dependencies
  - External services, such as AWS S3 cloud storage used for image uploading, were mocked in tests. This allowed testing of internal business logic without introducing side effects from third-party interactions.

Negative test cases were included to verify that invalid or unexpected input did not cause errors or instability. After each test execution, database cleanup was automatically handled to ensure isolation between tests and to avoid state contamination.

The testing contributed significantly to the overall robustness of the application.

Deployment

Docker

The backend application was containerized using Docker, following a multi-stage build approach.

The Dockerfile is structured as follows:

Build Stage:

- Uses gradle:8.5-jdk21 as the base image for compiling the project.
- Copies the full project source code into the container, setting proper ownership with –-chown=gradle:gradle.
- Executes gradle bootJar –-no-daemon to build the executable Spring Boot .jar file inside the container.

Runtime Stage:

- Uses a much slimmer eclipse-temurin:21-jre image to run the application.
- Copies only the compiled .jar file from the previous build stage into the final image.
- Sets the working directory to /app and exposes port 8080.
- Defines the entrypoint to start the application with java –jar app.jar.

This multi-stage build approach ensures that the final image is lightweight, containing only the compiled application and necessary runtime libraries – without bundling any Gradle files or source code.

Benefits:

- Smaller image size
- Faster container startup.
- Reduced attack surface for production deployment.

AWS ECR

After building the Docker image locally, the next step was to push it to Amazon Elastic Container Registry (ECR), a managed Docker container registry provided by AWS.

The process included:

- Tagging the built image appropriately.
- Authentication against AWS ECR.
- Pushing the image.

Storing the image in ECR allowed later deployment onto EC2 without needing local Docker artifacts anymore.

AWS EC2

The application container is deployed onto an Amazon EC2 instance.

Deployment steps included:

- Provisioning the EC2 instance.
  - Selecting the base OS.
  - Using an appropriate instance type to balance cost and performance.
  - Creating a key pair, for SSH access.
- Security Group configuration.
  - Open necessary port to my IP for remote management.
  - Open necessary port for public view of the website.
- Installing Docker on the EC2 instance.
  - Docker Engine was installed manually via the official Docker installation scripts.
- Running the container.
  - After logging into ECR from the EC2 instance and pulling the image, the container was started via docker run.

With this, I was able to access and configure the EC2 instance from my own computer via the terminal.

Domain setup

The application is accessible via a custom domain purchased from Loopia.se.

Initially, the “@” (root domain) and “www” DNS records were configured in Loopia’s DNS management portal to point directly to the EC2 instance’s public IP address. However, after integrating Cloudflare, the domain's DNS settings were updated to route traffic through Cloudflare’s network.

Now, both the “@” and “www” DNS records point to Cloudflare, which acts as an intermediary between users and the EC2 instance. Cloudflare filters traffic, enhances security, and caches static content. It then forwards the valid requests to the EC2 instance, while keeping the server’s IP address hidden from the public.

This configuration ensures that visitors can access the application whether they include “www” in the URL or not, while benefiting from Cloudflare’s performance and protection features.

Reverse proxy and HTTPS

Nginx was installed on the EC2 instance to act as a reverse proxy between the internet and the Docker container.

Daniel Isaksson
Nackademin

Nginx was configured to:

- Forward HTTP (port 80) and HTTPS (port 443) traffic to the Spring Boot app inside the container
- Terminate SSL, handling HTTPS encryption and decryption

Initially, HTTPS was enabled using Let's Encrypt and Certbot, which automatically issued and renewed SSL certificates via a scheduled cron job.

After integrating Cloudflare, Nginx continued to serve as the internal reverse proxy, but Cloudflare took over as the public-facing HTTPS endpoint. Cloudflare now:

- Terminates HTTPS from the client's browser to its edge servers
- Forwards secure traffic to Nginx over HTTPS
- Enables Full (Strict) SSL mode, which validates the origin certificate (Let's Encrypt on EC2)
- Block all requests towards unknown endpoints.
- Rate limits certain endpoints.

This ensures end-to-end encryption, improves resilience, and adds an extra layer of protection and performance.

Daniel Isaksson
Nackademin

Conclusions

What I learned

The biggest lesson I learned from all of this was that when apps are in production, things can go south in unexpected ways. I assume that most developers, like me, do not test as well as they think they do against edge cases.

The entire process of creating a runnable container was not new to me, but still quite foreign. I developed new skills and learned a lot from having to do that on my own. From creating a Dockerfile according to my needs and running a container via SSH inside a remote server.

Challenges I faced

There were a couple of problems that arose due to differences between running the app locally and running it with the AWS infrastructure. A few of these issues stumped me for a while and required extensive research to fix.
For example, the admin dashboard did not work at all when the site was live because the admin role was not capitalized in the database. Since I create the role upon startup if it does not already exist, I had to change that part of the code and make some updates to the data in the database itself.

After launching the application in production, I began noticing suspicious traffic patterns, particularly during nighttime hours. Thousands of requests were sent in bursts to random, non-existent endpoints—mostly originating from countries with no clear reason to access a Swedish-language site. This outbound traffic raised concerns about server resource usage and security.
After investigating further, I decided to integrate Cloudflare as a protective layer. Cloudflare forbids calling unknown endpoints, filters out malicious bots, rate-limits suspicious activity, and generally reduces unnecessary load on the EC2 instance.

What I am proud of

I have never created something that is “alive” on the internet. Creating something and having it work well enough has made me feel an incredible amount of joy and a deep sense of accomplishment like never before in my programming journey.

Building a working calendar from scratch was something I had never done before. While my solution might not be perfect, it does what it is supposed to do. The journey to get there took a couple of days.

Daniel Isaksson
Nackademin

Setting up everything to run within AWS and having it work felt amazing, even though I took a couple of nervous stumbles along the way.

Setting up HTTPS was but a footnote in one of courses during my education at Nackademin. Setting that up and having it work as intended was a journey and something to keep with me moving forward.

Improvements or further exploration

While I dipped my toes into testing Controllers, I decided not to go too far with it. I felt that I would just be testing the service methods all over again, even though I know that is not the whole truth.
The fact that I chose to use thymeleaf also complicated things a bit. If my endpoints returned pure objects, writing tests would have been easier.

Which leads to the choice of frontend. Thymeleaf was chosen because it is something that I am comfortable with, and I felt that the allotted time did not allow me to get into React or Angular. I have dipped my toes into Angular and will continue doing so in the future.

I did not build a CI/CD pipeline for this project, but I might do so in the future. GitHub action together with AWS was something I started building but decided against in this early stage. Using a pure AWS solution for my pipeline is something I will have to investigate later when I am not pushing changes quite as frequently, for cost reasons.

Concluding thoughts

This project has been “my baby” for the past six months. Getting to spend a full month focusing on developing it has been a joy, a struggle, and a great learning journey.

While my future projects may look different, I am genuinely proud of what I accomplished here.

I hope that what I have built will help me in my journey to find a job in the programming field, and that I will continue developing my skills as I move forward.

Daniel Isaksson
Nackademin

Attachments

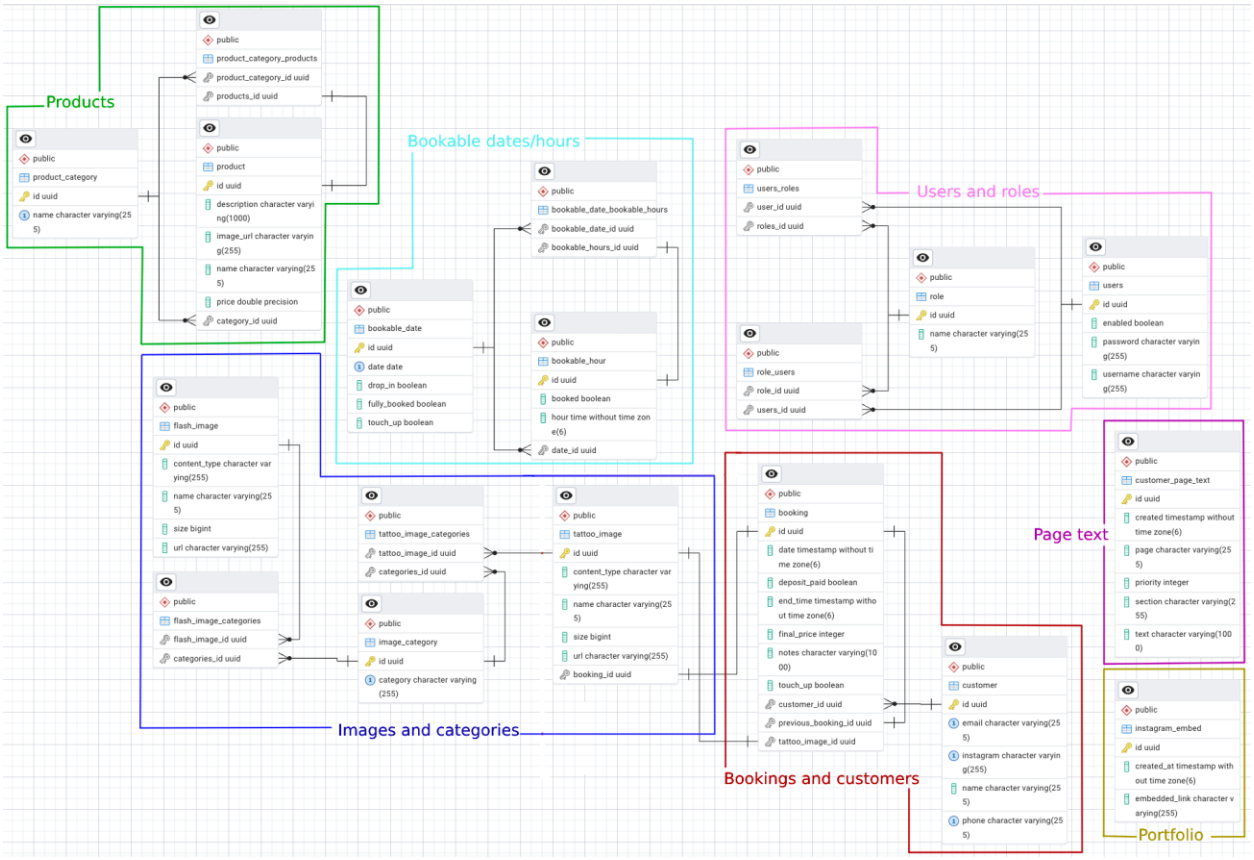

Figure 1: Database ERD

Daniel Isaksson
Nackademin

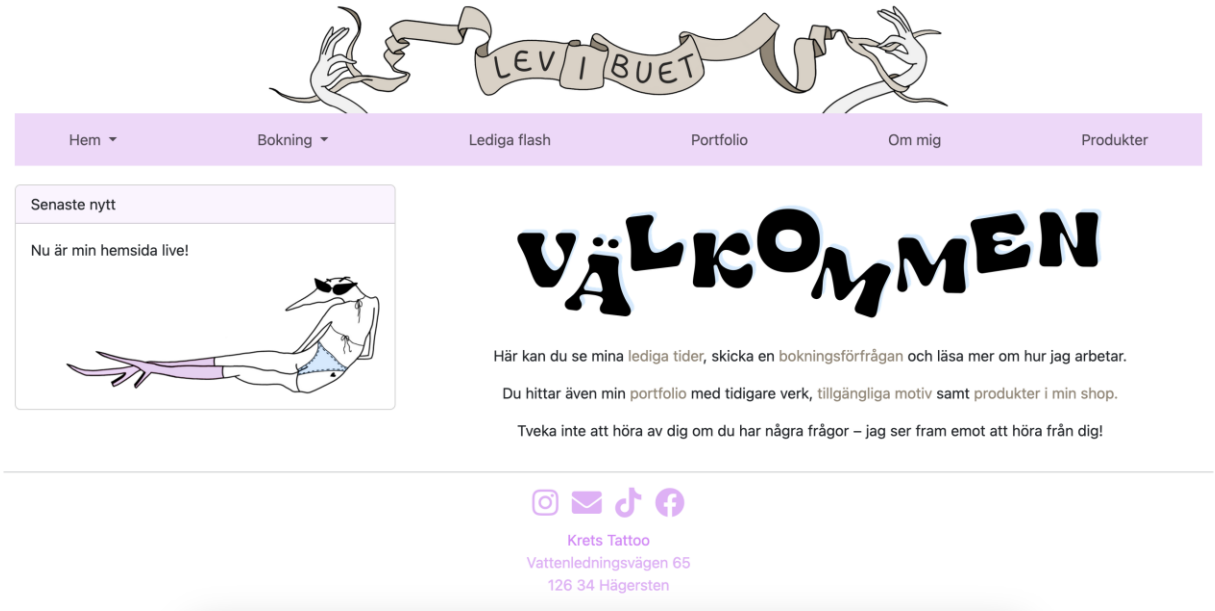

Figure 2: Home page on desktop

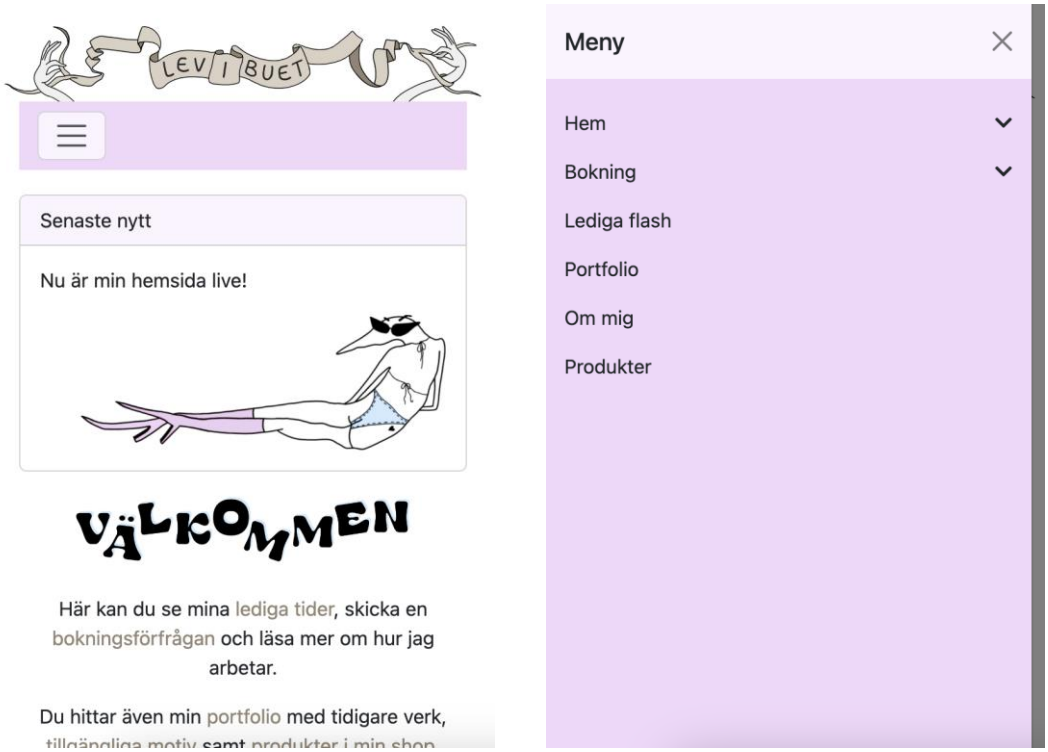

Figure 3: Home page and menu on phone

Daniel Isaksson
Nackademin

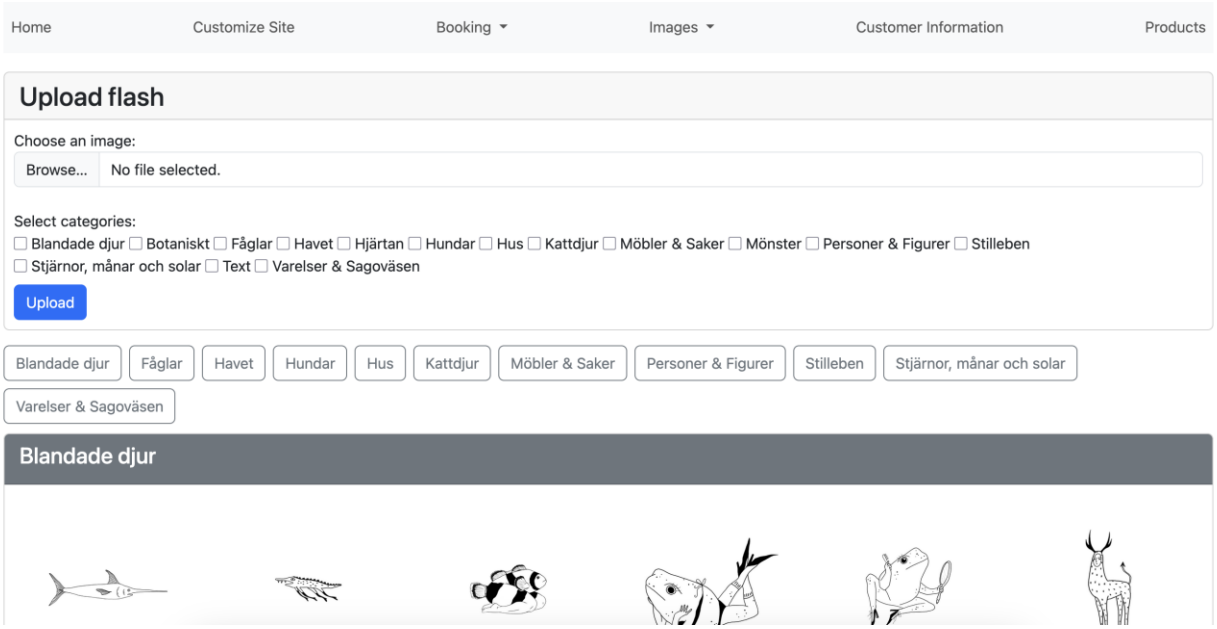

Figure 4: Admin dashboard - Flash images

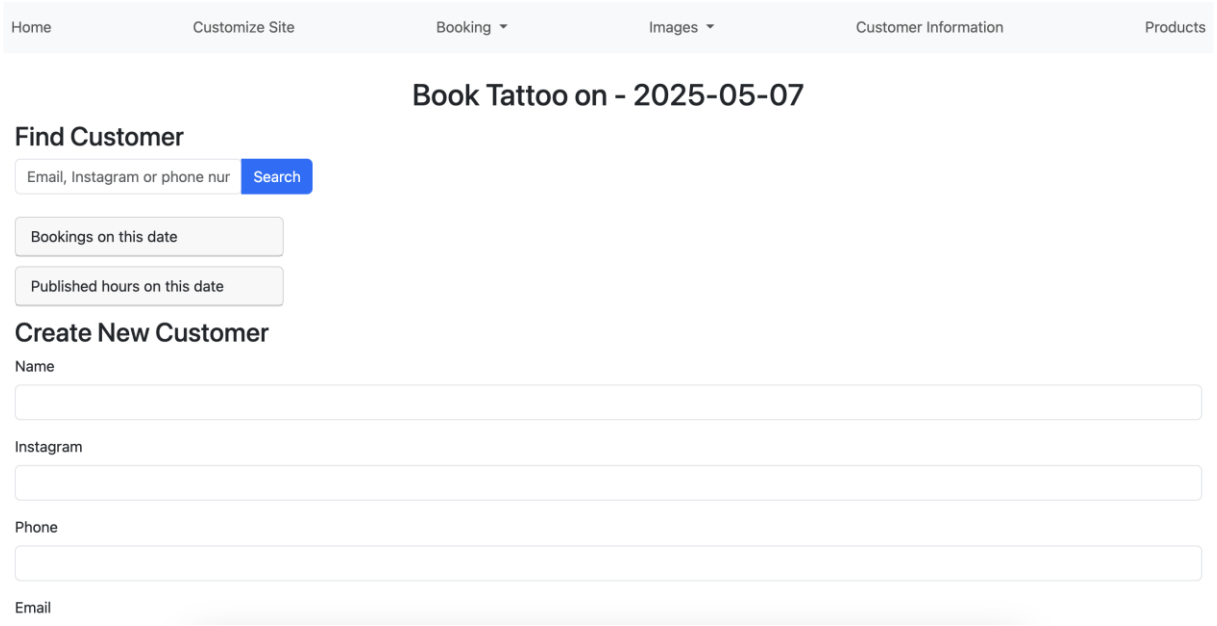

Figure 5: Admin dashboard – Book tattoo

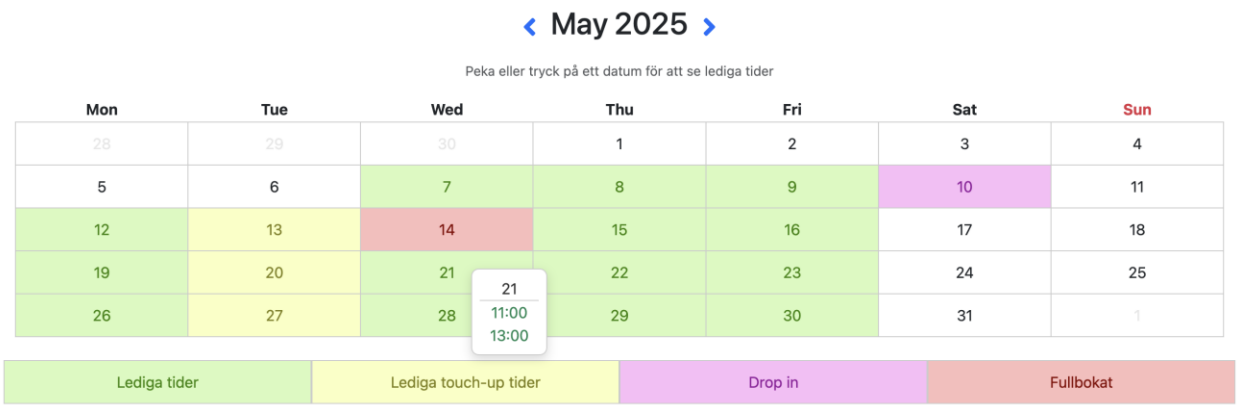

Figure 6: Calendar with hover or tap effect.

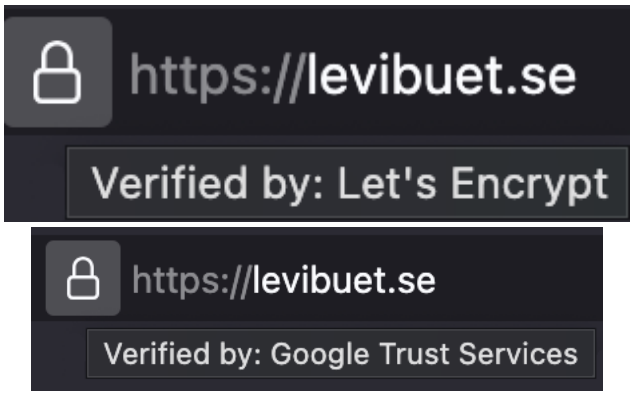

Figure 7: Encrypted web page.

May 14, 2025 9:32:22 AM	Block	Russian Federation	[redacted]	Custom rules
May 14, 2025 9:32:12 AM	Block	Russian Federation	[redacted]	Custom rules
May 14, 2025 9:31:14 AM	Block	Russian Federation	[redacted]	Custom rules
May 14, 2025 9:28:52 AM	Block	Russian Federation	[redacted]	Custom rules
May 14, 2025 9:12:44 AM	Block	Singapore	[redacted]	Custom rules
May 14, 2025 9:12:44 AM	Block	Singapore	[redacted]	Custom rules
May 14, 2025 9:12:44 AM	Block	Singapore	[redacted]	Custom rules
May 14, 2025 9:12:44 AM	Block	Singapore	[redacted]	Custom rules
May 14, 2025 9:12:44 AM	Block	Singapore	[redacted]	Custom rules

Figure 8: Blocked requests.

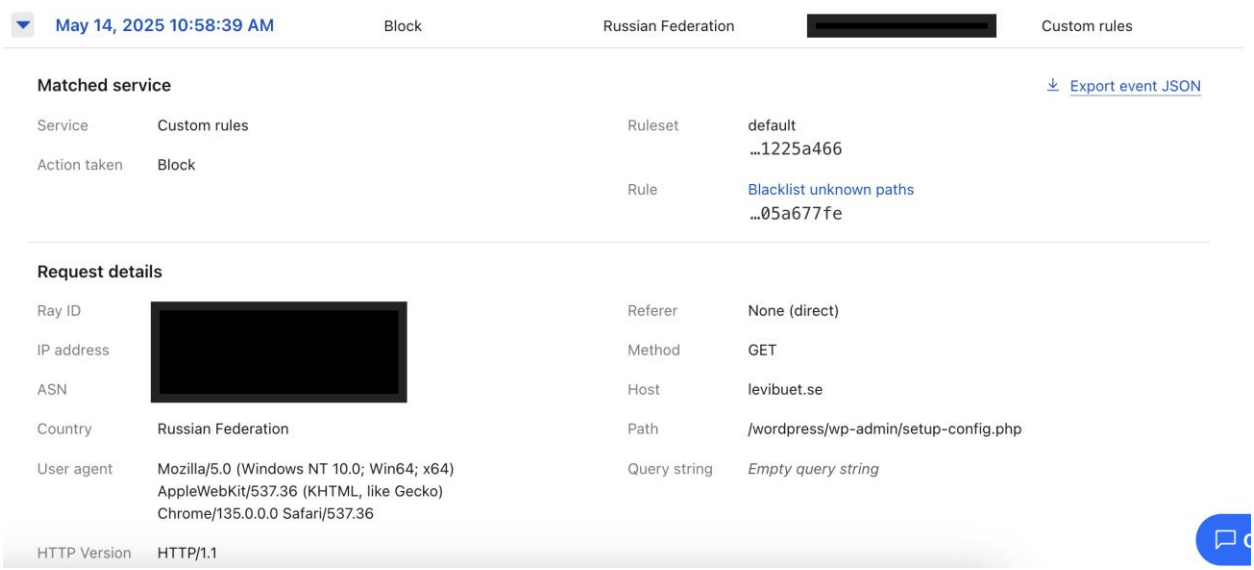

Figure 9: Specifics about blocked request.