KEA CARS™

Group:
Oliver Roat Jørgensen
Marcus Klinke Thorsen

Github:
https://github.com/Marcus-K-Thorsen/kea_car_microservices.git
https://github.com/OliverRoat/kea_car_frontend.git

Security in Web Development report by Oliver Jørgensen and Marcus Thorsen
2025

1. Introduction

KEA CARS™ is a mid-sized automotive business with a global focus, employing approximately 60 staff members, including salespeople, managers, and administrative personnel. The company specializes in assembling customized cars for customers, allowing them to select models, accessories, and insurance options tailored to their needs. The core of KEA CARS™'s operations revolves around registering cars, managing car purchases, and ensuring that each sale is completed within a specified deadline.

To support these business processes, KEA CARS™ utilizes a modern microservices-based web platform. Instead of relying on a single monolithic backend, the system is divided into multiple microservices, each responsible for a distinct aspect of the business. This architecture allows responsibilities such as car sales, customer registrations, employee management, and insurance offerings to be handled independently and securely. The majority of the system is designed for internal use by employees, providing secure access with role-based controls, while robust authentication and authorization mechanisms protect critical business data.

In addition to its internal functions, the platform also exposes selected non-critical information to potential customers and unauthenticated users, such as available car models, brands, colors, accessories, and insurance options. This is achieved without exposing sensitive data or compromising the security of the system's infrastructure. The following rapport explores the security considerations, challenges, and solutions implemented in the KEA CARS™ web platform, highlighting how security best practices are integrated into the design and operation of a real-world business application.

1.1. Project Description

The project includes five microservices, that are developed in Python:

- **Admin Microservice**: admin_microservice
- **Authentication Microservice**: auth_microservice
- **Employee Microservice**: employee_microservice
- **Synchronizer Microservice**: synch_microservice
- **Customer Microservice**: customer_microservice

The project also includes one client in a different repository, that is developed with React:

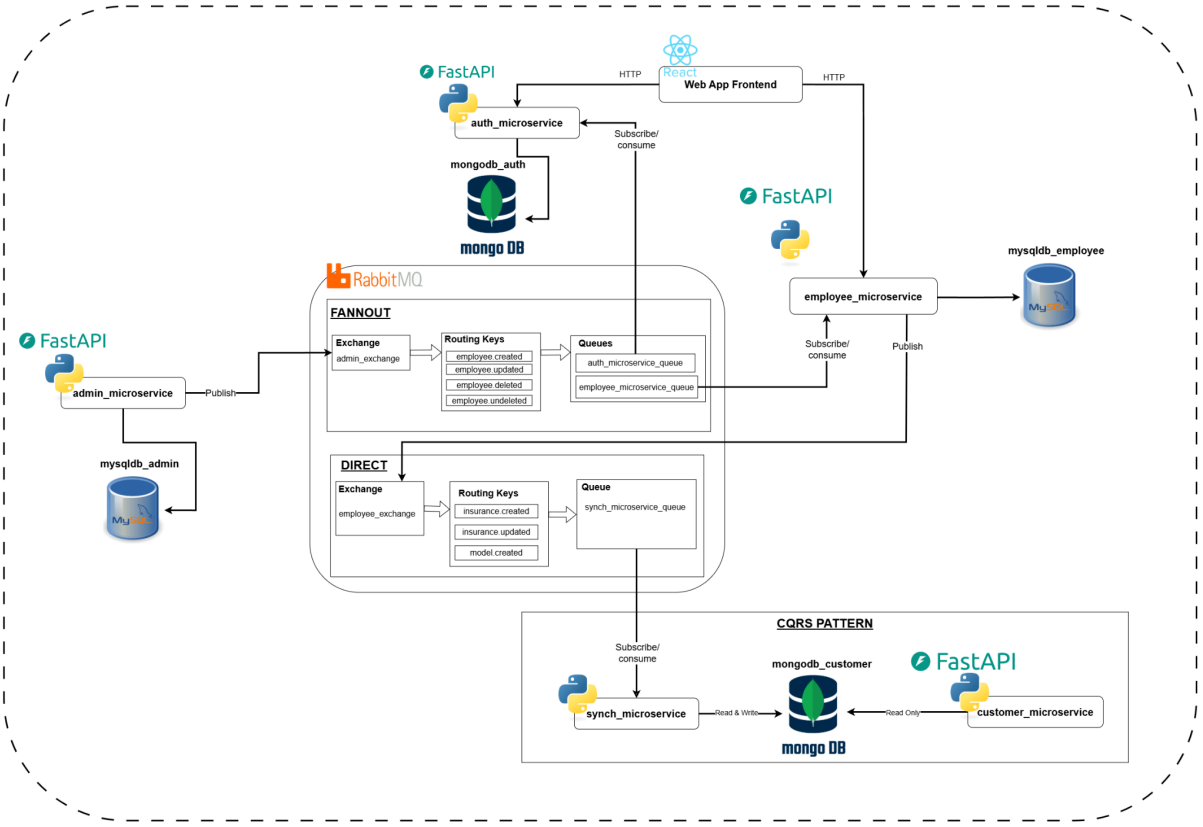
- **Frontend:** employee_client

The KEA Cars Microservices system is a secure, role-based web platform designed to support the daily operations of KEA CARS™, a company specializing in selling vehicles. The system is built using a microservices architecture, where each service is responsible for a distinct business domain and communicates with others through RabbitMQ as a message broker. This approach ensures scalability, maintainability, and clear separation of responsibilities.

The platform supports multilevel login with backend authentication, allowing employees with different roles—admin, manager, and sales person—to access features according to their

privileges. New user (employee) registration is managed by the **Admin Microservice**, which securely handles employee records, including password hashing and role assignment. Authentication is handled by the **Authentication Microservice**, which issues JWT tokens for secure access to protected endpoints across the system.

The **Employee Microservice** is the core business service, enabling employees to manage brands, models, colors, accessories, insurances, customers, cars, and purchases. It enforces role-based access control, so only authorized users can perform sensitive operations. The **Customer Microservice** provides public, read-only access to non-critical data, allowing potential customers to browse available options without authentication, while the **Synchronizer Microservice** keeps this public data up to date. Here is a diagram of the system architecture of the KEA Cars Microservices system.



For file uploads, such as vehicle model images, the system includes robust security checks to ensure only valid image files of acceptable size are accepted. Uploaded images are stored externally (on DigitalOcean), and only the image URL is saved in the database, minimizing the risk of storing large or unsafe files directly.

The frontend interacts with the backend using secure session management. Authentication tokens and user data are stored in the browser's sessionStorage, ensuring that sensitive information is only available for the duration of the session and is not shared across tabs or persisted after the session ends.

2. User Registration, Authentication, and Role-Based Access Control

The KEA Cars Microservices system enforces secure user registration, authentication, and strict role-based access control to ensure that employees only have access to the data and actions appropriate for their role.

2.1. User Registration

New employees can only be registered by an admin through the admin microservice. The admin is the only role with access to endpoints for creating, updating, deleting, and undeleting employee accounts in the admin database. During registration, the admin assigns each employee a role—admin, manager, or sales person—which determines their privileges throughout the system. Passwords are securely hashed and validated for strength before storage.

2.2. Authentication

All employees authenticate via the authentication microservice, which issues a secure JWT token upon successful login. This token, containing the employee's id, must be included in the Authorization header for all protected API requests.

2.3. Role-Based Access Control

The KEA Cars Microservices system enforces strict separation of data visibility and access privileges, both at the application (API) level and the database level. This ensures that users only see and interact with data appropriate to their role, and that backend services operate with the minimum privileges required.

Application-Level Roles and Data Visibility
At the API level, employees are assigned roles—admin, manager, sales person—or are unauthenticated users (potential customers). Each role determines what data and actions are available:

- **Admin:** Has exclusive access to all employee management endpoints in the admin microservice and can view all employee records there. In the employee microservice, admins have the same access as managers, meaning they can view, create, and delete cars and purchases for any employee.
- **Manager:** In the employee microservice, managers can view all employee records, cars, and purchases, and can create or delete cars and purchases for any employee. However, managers cannot create, update, or delete employee records in any database. As well as they can create and update insurances, and insert new car models available in the system.
- **Sales Person:** In the employee microservice, sales people can only view their own employee record, and can only view, create, and delete cars and purchases that are

attached to themselves. They cannot access or modify records belonging to other employees. Nor can they modify insurance and car model records.
- **Unauthenticated Users (Customers):** Can only access public, non-critical data (such as available car models, brands, colors, accessories, and insurances) via the customer microservice, with no access to sensitive or private data.

This structure ensures that private data is only visible to authorized roles, while public data is accessible to all.

Database-Level Roles and Access
Each database in the system is configured with two types of users: a root (admin) user and an application user. The application user is used exclusively by the API endpoints and is granted only the minimum privileges necessary for the microservice's operations:

- **Admin Database (MySQL):**
  - *Root User*: Full privileges, used for development and internal operations.
  - *Application User*: Can only SELECT, INSERT, and UPDATE employee records; cannot DELETE.

- **Auth Database (MongoDB):**
  - *Root User*: Full privileges, used by the message consumer and for seeding data.
  - *Application User*: Read-only access, used by the API endpoints to authenticate users and issue tokens.

- **Customer Database (MongoDB):**
  - *Root User*: Full privileges, used by the message consumer and for seeding data.
  - *Application User*: Read-only access, used by the API endpoints to serve public data.

- **Employee Database (MySQL):**
  - *Root User*: Full privileges, used by the message consumer for employee data synchronization.
  - *Application User*: Can SELECT, INSERT, UPDATE, and DELETE on all tables except the employees table, where it can only SELECT.

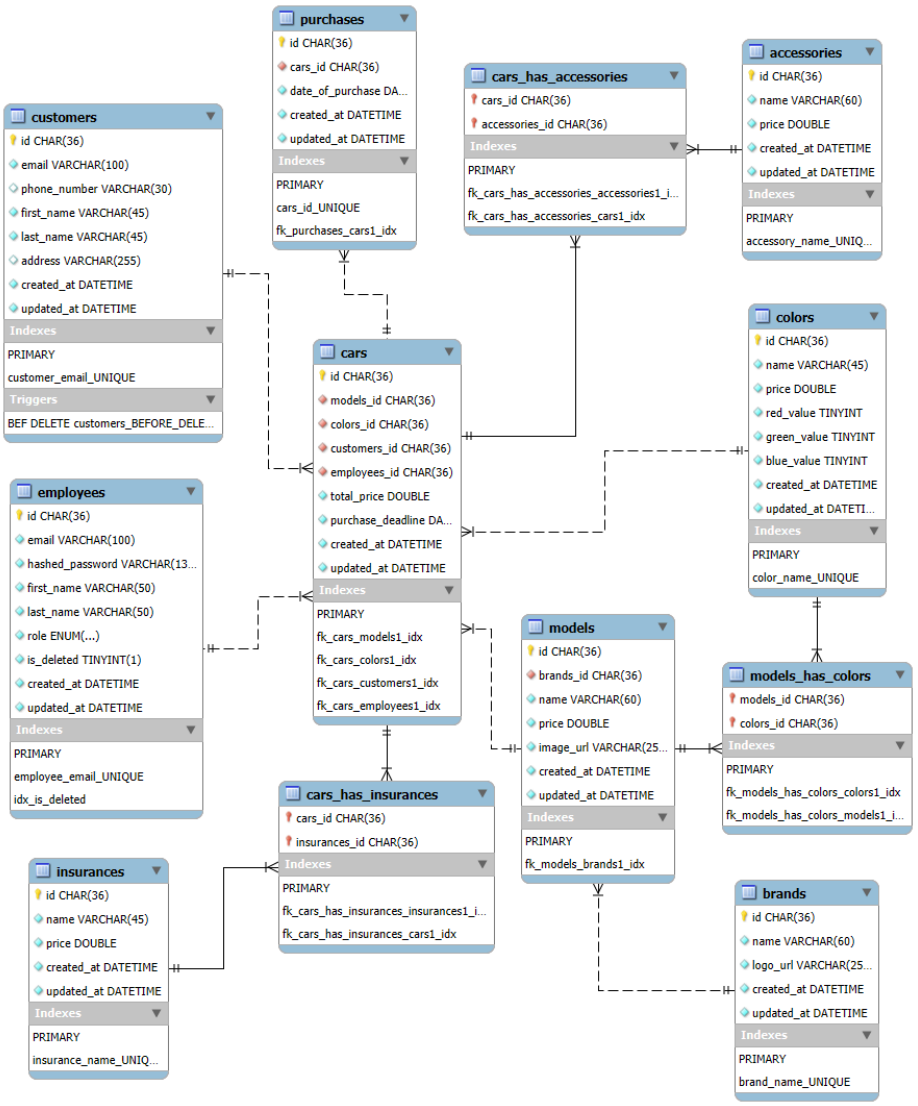
By restricting the application user's privileges, the system ensures that even if an API endpoint is compromised, the potential impact is limited. The root user, with full privileges, is never exposed through the API and is only used internally for trusted operations such as data seeding or message consumption.

This layered approach to access control—combining application-level roles with tightly scoped database users—ensures robust protection of both private and public data, and enforces the principle of least privilege throughout the system.

This architecture ensures that sensitive operations are strictly limited to authorized roles, and that each user only has access to the features and data necessary for their responsibilities.

2.4. Employee Data Structure & Database Synchronization

The following EER diagram shows the structure of the employee MySQL database used by the Employee Microservice. This structure is shared across multiple parts of the system with slight variations:



- **The Admin MySQL database** contains the same employees table structure for managing user creation and updates.

- **The Authentication MongoDB database** also stores employee data, but without the is_deleted field, as it is not needed for authentication logic.

- **The Customer MongoDB database** does not store employee data. Instead, it contains public, read-only collections that mirror selected tables from the Employee database, including models, brands, accessories, insurances, and colors.

This duplication of data across services ensures fault tolerance and decoupling of responsibilities in line with microservice principles.

3. Data Validation and Secure Handling

Ensuring the integrity, security, and reliability of data is a core principle in the KEA Cars Microservices system. This chapter presents the key strategies and mechanisms used to validate, securely handle, and monitor data throughout the application. You will learn how session data is managed in the web application, how user input (such as email addresses and file uploads) is validated, and why it is essential to enforce business rules on both the client and server sides. The chapter also explains how errors are handled and logged in a secure manner, ensuring that sensitive system details remain protected while still supporting effective troubleshooting. Together, these practices help safeguard the system against invalid or malicious input, maintain high data quality, and provide a secure and user-friendly experience for all users.

3.1. Secure Session Data Storage in the Web Application

A critical part of secure data handling is how authentication and session information are managed on the client side. In web applications, the way session data is stored and transmitted directly impacts both security and user experience. This subchapter explains the authentication approach used in the KEA Cars system and how session data is securely managed in the browser.

An authentication scheme defines how a client proves its identity to a server when accessing protected resources. Common schemes include:

- **Basic:** Sending username and password in the header.
- **Digest:** Using hashed credentials.
- **API Key:** Sending a unique key with each request.

Each has its own security properties and use cases.

The KEA Cars microservices project uses the **Bearer** authentication scheme, which is particularly well-suited for stateless, distributed systems. In this scheme, a bearer token (such as a JWT) is issued to the client after successful login. The token itself is proof of authentication—anyone who possesses it (the "bearer") can access protected resources. The client includes this token in the Authorization header of each HTTP request, formatted as Authorization: Bearer <token>. Unlike Basic or Digest schemes, the Bearer scheme does not require the server to maintain session state, making it ideal for microservices architectures where each service can independently validate tokens.

In our system, when an employee logs in through the frontend, the authentication microservice returns a JWT access token and employee information (excluding sensitive data like the hashed password). This token and employee data are stored in the browser’s sessionStorage, ensuring that authentication data is only available for the duration of the browser tab session. Additionally, each token is configured to expire after one day, after which it becomes invalid and cannot be used for authentication. This expiration mechanism helps limit the risk if a token is ever leaked or stolen.

For every authenticated request, the frontend retrieves the access token from sessionStorage and attaches it to the Authorization header using the Bearer scheme. This is automated using an Axios request interceptor, ensuring all protected API endpoints receive the necessary credentials.

By using sessionStorage and the Bearer authentication scheme with time-limited tokens, the KEA Cars application achieves secure, stateless authentication across all microservices. This approach not only protects sensitive session data but also supports the system's goals of data integrity, secure handling, and a seamless user experience.

3.2. User Input Validation and File Upload

User Input Validation (Email)
A critical aspect of our KEA Cars Microservices system is ensuring that user input is validated before being processed or stored. For email addresses, the system uses the EmailStr type from the Pydantic library, which relies on the popular email-validator package. This checks that any email provided has the correct format, contains a single @, a valid domain, and only allowed characters, while also enforcing length limits.

The main shortcoming of this approach is that it does not support every rare or unusual email format, and it does not check if the email address actually exists—only that it looks valid. However, this is more than sufficient for our project, since only admins (trusted users) enter employee emails. The goal is to prevent obvious mistakes and ensure each employee's email is well-formed for use as a username and for communication. This makes the chosen validation method both practical and reliable for our needs.

File Upload (Images)
File upload functionality is provided in the employee microservice, specifically for adding images to new car models. Only users with the role of admin or manager can access the endpoint for creating a model with an image, which performs several security checks on uploaded files:

- **Content Type Validation:** Only image files of type PNG or JPEG are accepted.
- **Extension Validation:** Only files with .png, .jpg, or .jpeg extensions are allowed.
- **File Size Check:** Uploaded images must not exceed 3MB in size.
- **Image Verification:** The system verifies that the uploaded file is a valid image..
- **External Storage:** After validation, the image is stored externally (e.g., on DigitalOcean), and only the URL is saved in the database, reducing the risk of storing large or potentially unsafe files directly in the system.

These checks help prevent malicious file uploads and ensure that only appropriate images are associated with car models.

3.3. Server-Side Validation vs. Client-Side Manipulation

Server-side validation and client-side manipulation are two fundamental concepts in web application security and data integrity. Client-side validation refers to checks performed in the user's browser before data is sent to the server. This validation provides immediate feedback to the user, helping them understand what is allowed and preventing many common mistakes before a request is even made. By alerting users to errors as they occur, client-side validation makes the application easier to use and helps users avoid confusing or technical error messages from the server.

However, it is a core security principle to never trust data received from the client, regardless of any validation performed on the frontend. Client-side validation can always be bypassed by malicious users who manipulate requests or use custom tools to send data directly to the server. Therefore, robust server-side validation is essential to prevent invalid, inconsistent, or potentially harmful data from entering the system. For example, while the frontend may check that an email address appears to be valid, the backend must independently verify the email format before accepting it.

A practical example in this project is the process of selecting accessories when creating a car. The frontend restricts users to selecting a maximum of 10 accessories for a car, providing a user-friendly experience and preventing most mistakes. However, to ensure this rule cannot be bypassed, the backend also enforces the same limit. When a request to create a car is received, the server validates that no more than 10 accessories have been selected before proceeding. If this condition is not met, then the request is rejected, regardless of what the client attempted to submit.

By implementing validation on both the client and server, provides the system a better user experience while maintaining strong security and data integrity. This layered approach ensures that even if client-side checks are circumvented, the server will always enforce the business rules and protect the application from unintended or malicious input.

3.4. Secure Error Handling and Logging

A critical aspect of our system is the secure handling and logging of errors. Every microservice with API endpoints uses its own handle_http_exception function to manage exceptions in a consistent and secure manner.

This function ensures that only generic or controlled error messages are returned to the client, preventing the exposure of sensitive internal details about the system’s structure or logic. We have also defined custom exceptions for specific error scenarios, allowing us to return appropriate status codes and clear, minimal messages tailored to the type of error encountered.

All detailed error information, including stack traces and exception details, is logged internally within the container running the microservice. These logs are accessible only to developers and system administrators, allowing for effective debugging and monitoring while keeping sensitive information protected from end users and potential attackers.

This approach minimizes the risk of information disclosure through error messages, supports secure operations, and ensures that only authorized personnel have access to the full details needed for troubleshooting and system maintenance.

4. Mitigation of Common Web Vulnerabilities

Web vulnerabilities are weaknesses or flaws in web applications that can be exploited by attackers to compromise the security, integrity, or availability of a system. Common web vulnerabilities—such as SQL injection, command injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and XML external entity (XXE) attacks—are well-known threats that can lead to data breaches, unauthorized access, or service disruption if not properly addressed.

Mitigating these vulnerabilities means identifying potential risks in the application's design and implementation, and applying best practices, secure coding techniques, and architectural safeguards to make exploitation as difficult as possible. While it is impossible to guarantee complete security, the goal is to reduce the likelihood and impact of attacks by making the system a less attractive or accessible target. This chapter outlines the most prevalent web vulnerabilities relevant to modern web applications and explains how the KEA Cars Microservices system is designed to defend against them, ensuring robust protection for both data and users.

4.1. SQL Injection and Command Injection

SQL injection and command injection are serious security vulnerabilities where attackers attempt to execute unauthorized commands or manipulate database queries by injecting malicious input through application interfaces. SQL injection targets the database layer, aiming to access, modify, or delete data, while command injection seeks to execute arbitrary system commands on the server.

4.1.1. SQL Injection

SQL injection is a common and dangerous vulnerability that occurs when user-supplied input is improperly handled in database queries, allowing attackers to manipulate SQL statements and potentially access, modify, or delete sensitive data. This can lead to unauthorized data exposure, data corruption, or even complete loss of database integrity.

To prevent SQL injection our KEA Cars microservices, several best practices and architectural safeguards are implemented:

- **Use of ORM (SQLAlchemy):**
  All microservices that interact with SQL databases use SQLAlchemy as their Object-Relational Mapper (ORM). SQLAlchemy abstracts away direct SQL query construction and enforces the use of parameterized queries. This means that user input is never directly concatenated into SQL statements; instead, it is safely passed as parameters, ensuring that input is treated strictly as data.

- **Repository Pattern:**
  Database operations are encapsulated in repository classes, which use SQLAlchemy's ORM methods for all CRUD operations. This design prevents developers from accidentally writing raw SQL queries that could be vulnerable to injection.

- **Session and Transaction Management:**
  Database sessions are managed using context managers, which automatically handle committing or rolling back transactions. This reduces the risk of partial or unintended query execution, further protecting against injection attacks.

- **Input Validation:**
  All user input is thoroughly validated at the API layer before it is processed or sent to the database. For example, the system checks that each field—such as an employee's name—does not exceed the maximum length allowed by the corresponding database column (e.g., names are limited to 50 characters). This validation also includes type checks, format checks, and enforcement of business rules. By ensuring that all input meets these requirements, the system prevents malicious or malformed data from ever reaching the database layer, protecting both data integrity and application security.

- **No Raw SQL with User Input:**
  The codebase avoids using raw SQL queries with user-supplied data. All queries are constructed using SQLAlchemy's ORM interface, and there are no instances where user input is directly embedded into raw SQL statements.

- **Restricted Database Privileges:**
  The database users that the API endpoints utilize are granted only the minimum privileges necessary for application functionality. For example, the application user for the admin and employee microservices cannot create or alter tables, nor can it delete rows from the employees table. This restriction ensures that even if an attacker were to exploit an injection vulnerability, their ability to cause harm would be severely limited.

By combining these practices, our project architecture provides strong protection against SQL injection attacks, ensuring that all user input is handled securely and that the integrity of the database is maintained.

4.1.2. Command Injection

Command injection is a critical security vulnerability that occurs when an application passes unsafe user-supplied data to a system shell or operating system command. If user input is not properly validated or sanitized, an attacker may be able to execute arbitrary commands on the server. For example, if an endpoint allowed a user to submit a filename that was then directly inserted into a function that interacts with the operating system to delete files, a malicious user could craft their input in such a way that, instead of just deleting a specific file, the command would delete important system files or even the entire server.

In our system, command injection risks are mitigated by design and secure coding practices:

- **No Shell Execution from User Input:** The backend services do not expose any functionality that executes shell commands or system-level operations based on user input. All business logic is handled within the application code, and there are no endpoints or features that would allow user data to be passed to the operating system.

- **Strict Input Validation:** All user input is validated and sanitized at the API layer before being processed. This ensures that even if a feature were to be added in the future that interacts with the system shell, input would be rigorously checked to prevent malicious payloads.

- **Use of High-Level Libraries:** The system relies exclusively on high-level libraries and frameworks for all critical operations. For example, SQLAlchemy is used for all SQL database interactions, and PyMongo is used for MongoDB operations in the authentication microservice and customer microservice. These libraries provide safe abstractions that do not expose direct access to the system shell or allow arbitrary command execution, further reducing the risk of command injection.

- **MongoDB Security:** Both the authentication and customer microservices use MongoDB for data storage. All interactions with these databases are performed using PyMongo, which does not provide any mechanism for executing system commands based on user input. Additionally, database users are configured with the minimum privileges necessary, and the application never constructs or executes shell commands from user data.

- **Principle of Least Privilege:** The application runs with restricted permissions, and database users are limited to only the operations required for normal functionality, further reducing the potential impact of any attempted command injection.

By avoiding the use of system shell commands, relying on secure high-level libraries, and enforcing strict input validation, the KEA Cars Microservices architecture effectively eliminates the risk of command injection attacks within its current design.

4.2. Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a vulnerability where attackers inject malicious scripts into web pages viewed by other users, potentially stealing session tokens, impersonating users, or manipulating page content. The three main types of XSS are Stored XSS, Reflected XSS, and DOM-based XSS.

In our KEA Cars project, the combination of React's automatic escaping, the absence of raw HTML injection, structured API responses, and careful use of sessionStorage for sensitive data all work together to make XSS attacks highly unlikely. This ensures that authentication tokens and user data remain protected throughout the user's session.

4.2.1. Stored XSS

Stored XSS occurs when malicious scripts are permanently stored on the server, such as in a database, and later served to users as part of normal content.

In this project, the risk of stored XSS is very low because only authenticated users can submit or store data, and the customer microservice, which is accessible to unauthenticated users, is strictly read-only. All user-generated content is validated and sanitized before being stored or displayed. On the frontend, React escapes all dynamic values rendered in the UI by default, so even if potentially malicious content were present in the data, it would be displayed as plain text rather than executed as code.

4.2.2. Reflected XSS

Reflected XSS happens when user input is immediately returned by the server in a response, such as in error messages or search results, without proper escaping, allowing scripts to execute in the victim's browser.

In our project, API endpoints are designed to return only structured JSON data, not HTML, which eliminates direct injection points for reflected scripts. The frontend renders all API data as plain text using JSX expressions, so even if user input is reflected in the UI, it is not executed as code.

4.2.3. DOM-based XSS

DOM-based XSS occurs when client-side JavaScript modifies the page based on untrusted data, leading to script execution without any server involvement.

In this project, the frontend is built with React and Material UI, which do not manipulate the DOM with raw user input. All data from APIs is rendered as structured data, not as HTML, and user input is handled through controlled components and validated before being sent to the backend, further reducing the risk of DOM-based XSS.

4.3. Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a web security vulnerability where an attacker tricks a user's browser into making unwanted requests to a web application in which the user is authenticated. For example, in the context of this project, imagine an attacker crafts a special link or webpage that, when visited by an employee who is already logged into the KEA Cars system, automatically sends a request to the backend to delete a car or perform another sensitive action. The attacker might send this link via email or embed it on a malicious website. If the employee clicks the link while authenticated, their browser would unknowingly send the request, and the backend could process it as if it came from the legitimate user. However, since we do not use cookie-based authentication, this type of attack is not possible in our system.

How CSRF is Handled in This Project

- **Token-Based Authentication:**
  All authentication is handled using JWT tokens, which are stored in the browser's sessionStorage and must be explicitly included in the Authorization header for every request. Browsers do not automatically attach these tokens to outgoing requests, unlike cookies, so an attacker cannot force a victim's browser to send authenticated requests

to our backend.

- **No Cookie-Based Sessions:**
  Because we do not use cookies for authentication, there is no automatic credential sharing between browser requests and the backend. This eliminates the primary mechanism that CSRF attacks rely on.

- **Stateless APIs:**
  All endpoints require a valid JWT token in the request header. If the token is missing or invalid, the request is rejected, regardless of the source.

- **CORS Configuration:**
  During development, our CORS settings allow all origins to make requests to the backend to facilitate frontend development. However, we are aware that in a real-world production scenario, the allow_origins setting should be restricted to only the trusted frontend domain(s). This would further reduce the risk of unauthorized cross-origin requests.

In summary, the use of explicit token-based authentication, stateless API design, and careful CORS configuration ensures that CSRF attacks are not possible in our KEA Cars Microservices system.

4.4. XML External Entity (XXE) and Serialization/Injection

XML External Entity (XXE) is a security vulnerability that can occur when an application parses XML input and allows the use of external entities. This can enable attackers to craft XML documents that reference sensitive files or external resources, potentially exposing confidential data or enabling denial-of-service attacks.

Serialization/injection vulnerabilities, on the other hand, arise when an application deserializes untrusted data—such as XML, JSON, or binary objects—without proper validation. If not handled securely, this can allow attackers to manipulate application logic or even execute arbitrary code.

Both XXE and insecure serialization/injection are related in that they exploit the way applications process complex, structured data from untrusted sources. XXE is specific to XML parsing, while insecure serialization/injection can affect any data format if the application trusts and processes user-supplied data without sufficient safeguards.

In the context of the KEA Cars system, these risks are minimized by design. For example, when an employee uploads a file as part of creating a new car model, the backend reads the file and performs a series of validations to ensure it is a legitimate image before any further processing or storage. The file is never parsed as XML or deserialized as an object, and after validation, it is simply sent to external storage (such as DigitalOcean). This approach ensures that even if an attacker attempted to upload a malicious file, it would not be interpreted or executed by the system.

Currently, the KEA Cars microservices do not accept or process XML files or serialized objects from users. All data is submitted as JSON through API endpoints, and file uploads are strictly validated as images. However, it is important to recognize that if the system were to support XML or custom serialization in the future, secure parsing practices would be essential—such as disabling external entity resolution and validating all input before processing. But at the moment the system is currently not vulnerable to XXE or insecure serialization/injection attacks because it does not process XML or deserialize untrusted objects. All user-uploaded files are strictly validated as images, and no file content is executed or deserialized.

5. Network and Transport Security

Network and transport security are essential components of any modern software system, ensuring that data remains protected as it moves between users, services, and infrastructure. This chapter explores the strategies and best practices used to safeguard network communications and control access to system resources. It covers the role of firewalls in restricting unauthorized connections and the importance of encrypting data in transit using protocols like TLS. While the KEA Cars Microservices project is currently run locally for development and testing, the principles and recommendations discussed here are critical for any real-world deployment. The following subchapters explain how these security measures work, why they matter, and how they would be applied to protect the system in a production environment.

5.1. Firewall Configuration

A firewall is a security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules. Its primary purpose is to create a barrier between trusted internal networks and untrusted external networks, such as the internet, to prevent unauthorized access and protect sensitive data and services.

In the context of the KEA Cars Microservices project, all services are currently run locally for development and testing purposes. As a result, there is no need for a dedicated firewall configuration, since the services are not exposed to the public internet and are only accessible from the local machine or trusted development environment.

However, in a real-world deployment scenario, enabling and properly configuring a firewall would be essential. A firewall would be set up to restrict access to only the necessary ports required by each microservice (for example, the API ports and database ports) and to allow connections only from trusted IP addresses or networks. This would significantly reduce the attack surface and help protect the system from unauthorized access and network-based threats.

By planning for proper firewall configuration in production, the KEA Cars Microservices system can ensure a higher level of network security and resilience against external attacks.

5.2. Use of Transport Layer Security (TLS)

Transport Layer Security (TLS) is a cryptographic protocol that provides secure communication over a network by encrypting data transmitted between clients and servers. When enabled, TLS ensures that sensitive information—such as authentication tokens, user credentials, and personal data—cannot be intercepted or tampered with by unauthorized parties. TLS is most commonly recognized as the technology behind HTTPS, which secures web traffic in browsers.

In the context of the KEA Cars Microservices project, TLS is not currently implemented because all services are run locally for development and testing purposes, and the system is not exposed to the public internet. As a result, all network traffic remains within the local environment, and there is no immediate risk of interception by external attackers.

If the project were to be deployed in a real-world scenario—especially using Kubernetes for orchestration—TLS would be essential for securing all external traffic between users, services, and any public endpoints. In such a deployment, TLS certificates would be configured at the ingress controller or load balancer level (for example, using Nginx or a managed cloud solution) to ensure that all data in transit is encrypted and protected from eavesdropping or man-in-the-middle attacks.

While financial and practical constraints prevent public hosting and TLS configuration in the current setup, the project is designed with the understanding that enabling TLS is a critical step for any production deployment. This ensures that, when the system is eventually exposed to real users or external networks, all communications will be properly secured.

6. Cryptography and Secure Storage

Cryptography and secure storage are essential components of modern software security, ensuring that sensitive information remains protected from unauthorized access and tampering. Cryptography involves techniques for transforming data so that only authorized parties can read or verify it, while secure storage focuses on keeping this protected data safe at rest. Two fundamental concepts in this area are hashing and encryption.

Hashing is a one-way process that converts data, such as a password, into a fixed-length string of characters. This process is designed to be irreversible—meaning it is computationally infeasible to recover the original data from its hash. Hashing is commonly used for securely storing passwords, as it allows the system to verify a password without ever needing to store or transmit the original value.

Encryption, on the other hand, is a reversible process that transforms data into an unreadable format using a secret key. Only those with the correct key can decrypt the data and restore it to its original form. Encryption is used when sensitive information needs to be protected during storage or transmission but must also be recoverable by authorized users.

The main difference between hashing and encryption is that hashing is intended to be one-way and irreversible, while encryption is designed to be two-way and reversible with the appropriate key.

This chapter explores how the KEA Cars Microservices system applies these principles to protect employee passwords and manage authentication. The following sections detail the use of secure password hashing algorithms and the implementation of JWT-based authentication with secret keys, demonstrating how cryptography and secure storage are integrated into the project's security architecture.

6.1. Password Hashing and Storage

A fundamental aspect of securing user accounts in our system is the proper handling and storage of employee passwords. To protect against unauthorized access and data breaches, employee passwords are never stored in plain text. Instead, the system uses strong, industry-standard hashing algorithms to securely transform passwords before they are saved in the database.

When a new employee is registered, the password is first checked to ensure it meets minimum strength requirements, such as a sufficient length of 8 or more characters. The system also checks whether the chosen password has previously appeared in known data breaches by querying a reputable online service such as "Have I Been Pwned". To protect user privacy, only a partial, hashed version of the password is sent to this service, ensuring that the actual password is never exposed during the check.

Once the password passes these checks, it is securely hashed using the bcrypt algorithm. Bcrypt is a widely trusted password hashing function designed to be computationally expensive, making brute-force and rainbow table attacks significantly more difficult. It also incorporates a unique salt for each password, ensuring that identical passwords result in different hashes.

In this project, bcrypt is configured using the Passlib library, as specified in the project's configuration. The password hashing context is set up to use bcrypt as the default scheme, ensuring that all password hashes are generated and verified using this secure algorithm.

The hashed password is then stored in the database, ensuring that even if the database were compromised, the original passwords would remain protected. During authentication, the system verifies login attempts by hashing the provided password and comparing it to the stored hash, never exposing or transmitting the plain text password.

By following these best practices for password validation, breach checking, and secure storage with bcrypt, our system significantly reduces the risk of credential theft and helps maintain the security and privacy of employee accounts.

6.2. Use of JWT and Secret Keys

Our KEA Cars Microservices system uses JSON Web Tokens (JWT) as the foundation for authentication and secure communication between the frontend and backend services. When an employee successfully logs in, the authentication microservice generates a JWT that encodes essential information about the user, such as their unique identifier.
This token is then signed using a secret key, ensuring its authenticity and integrity.

All microservices that require authentication share a copy of the same secret key, which allows them to verify the validity of incoming JWTs. When a protected endpoint receives a request, it checks the token's signature and decodes its contents to confirm the user's identity and permissions. This stateless approach to authentication is well-suited for a microservices architecture, as it eliminates the need for centralized session storage and enables each service to independently validate requests.

The security of this system relies heavily on the secrecy and strength of the secret key used to sign and verify JWTs. If the secret key were to be leaked or compromised, an attacker could forge valid tokens and gain unauthorized access to protected resources. For this reason, the secret key is stored securely in environment variables and is never exposed in the codebase or logs.

By using JWTs signed with a strong, shared secret key, our KEA Cars Microservices system achieves secure, scalable, and efficient authentication across all services, while maintaining a clear separation of responsibilities and minimizing the risk of unauthorized access.

7. Session Management and CSRF

Session management and protection against Cross-Site Request Forgery (CSRF) are fundamental aspects of web application security.

Session management refers to the methods used to track and authenticate users as they interact with a system, ensuring that each request is properly associated with the correct user. Traditionally, this has been achieved using session IDs stored in cookies, allowing the server to maintain state and recognize returning users.

CSRF is a type of attack where a malicious website tricks a user's browser into making unwanted requests to another site where the user is authenticated, potentially leading to unauthorized actions. To defend against this, web applications often use CSRF tokens—unique values that must be included with sensitive requests to verify their legitimacy.

This chapter explores how session management and CSRF protection are typically implemented, and explains the approach taken in our system, and discusses the differences between traditional session and CSRF token mechanisms and the stateless JWT-based authentication used in this project, highlighting the reasons for this choice and its implications for security and scalability.

7.1. Session IDs and CSRF Tokens: When and Why

Session IDs and CSRF (Cross-Site Request Forgery) tokens are traditional mechanisms used to manage user authentication and protect against certain web vulnerabilities in stateful, cookie-based web applications.

Session IDs are unique identifiers generated by the server and stored in a secure cookie on the client's browser. Each time the client makes a request, the session ID is sent automatically by the browser, allowing the server to recognize the user and maintain session state.

CSRF tokens are random values generated by the server and embedded in forms or requests. They are used to ensure that requests made to the server are intentional and not forged by malicious third-party sites.

In our KEA Cars Microservices system, session IDs and CSRF tokens are not used. Instead, the system relies on stateless JWT (JSON Web Token) authentication. When a user logs in, they receive a JWT, which is stored in the browser's sessionStorage and explicitly included in the Authorization header of each request. This approach eliminates the need for server-side session storage and makes it easier for multiple microservices to independently verify user identity and permissions using the same token and shared secret key.

Advantages of using JWT-based, stateless authentication:

- **Scalability:** No need to synchronize session state across multiple services or servers.
- **Microservice compatibility:** Each microservice can independently validate tokens without relying on a central session store.
- **Simplicity:** The backend does not need to manage session expiration or cleanup.

Disadvantages of using JWT-based, stateless authentication:

- **No built-in CSRF protection:** Since JWTs are not sent automatically by the browser (unlike cookies), CSRF is less of a concern, but developers must ensure tokens are not exposed to XSS attacks.
- **Token security:** JWTs must be securely stored and transmitted, as anyone with a valid token can access protected resources until the token expires.

If the system were to use session IDs and CSRF tokens, the session ID would be stored in a secure, HTTP-only cookie, and the CSRF token would be included in each form submission or AJAX request (for example, as a hidden form field or custom HTTP header). The server would then validate both tokens on each request to ensure authenticity and protect against CSRF attacks.

In summary, the use of JWTs in our system is well-suited for a distributed, stateless microservices architecture, enabling secure and scalable authentication without the complexity of managing session IDs and CSRF tokens.

8. Configuration Management

Configuration management is the process of systematically handling and organizing the settings that control how software systems operate. In the context of security, it ensures that sensitive information—such as credentials and secret keys—is managed safely and that services are correctly connected and protected.

8.1. Project Configuration Settings

In the project, configuration is managed using environment variables, .env files, and Docker Compose. Each microservice has its own .env file containing sensitive values such as database credentials, JWT secret keys, and other service-specific settings. While certain values—like the secret key for JWT authentication—must be identical across the microservices that use them, each service keeps its own independent copy in its own .env file. To help developers configure the system correctly without exposing sensitive information, .env.example files are provided as templates, indicating where and what environment variables are required.

For ease of development, database users and their privileges are defined in SQL files within the **init_db** folder. While this makes setup straightforward, it exposes credentials and schema details to anyone with repository access. This approach is not secure for production; a real deployment would use more secure provisioning and avoid storing sensitive credentials in version control.

Docker Compose is used to orchestrate all services, passing the correct environment variables to each container and centralizing configuration management. This simplifies setup but requires careful coordination of shared variables, especially when multiple services interact with the same resources.

To maintain security, sensitive configuration files should never be committed to public repositories, and access should be limited to trusted team members. In production, secrets management tools and stricter provisioning practices are recommended. Because a leak of sensitive data from one microservice, can in certain cases lead to a security breach within other microservices, that shares the same values, such as the secret key used for token validation.

By using environment variables, .env files, and Docker Compose, the project achieves flexible and manageable configuration while remaining aware of the security risks and improvements needed for a production environment.

9. Conclusion

Working on our KEA Cars Microservices project has given us valuable practical experience with securing a modern web application. Throughout the project, we implemented key security principles from the course, including user authentication with JWT, role-based access control, input validation, and protection against common vulnerabilities like SQL injection, XSS, and CSRF.

The use of sessionStorage, hashed passwords, and strict access controls helped us build a foundation for a more secure system. That said, much of our focus was on local development, and our project lacks several production-level protections—such as proper firewall setup, secure secret management, and TLS encryption. These were acknowledged in theory but not implemented in practice, partly due to scope and resource limitations.

A significant aspect of our configuration management was the use of environment variables and .env files for sensitive settings such as database credentials and JWT secret keys. Each microservice maintains its own .env file, and while certain values—like the JWT secret key—must be identical across services, each service keeps its own independent copy. This approach supports the microservices architecture by avoiding direct sharing of variables, but it also means that a compromise of one .env file could potentially impact other services that use the same secret value. To help developers configure the system, .env.example files are provided as templates, but the real .env files contain sensitive information in plain text and are not securely managed. In a real deployment, this would be a critical issue, and more secure solutions—such as dedicated secrets management tools and restricted access to configuration files—would be necessary.

Additionally, for development convenience, database users and privileges are defined in SQL files within the repository, exposing credentials and schema details to anyone with access. While this simplifies setup, it is not suitable for production and highlights the importance of secure provisioning practices.

Despite these limitations, the project helped us connect course theory with real-world development, and showed us the importance—and difficulty—of maintaining security throughout all layers of a system. We leave the project with a stronger appreciation for secure design, a clearer sense of the work involved in building robust web applications, and an understanding of the improvements needed for a production-ready deployment.