

# S6 Research report

Jordy Walraven



## Version history:

Version	Author(s)	Description	Date
1	Jordy Walraven	Setup first research question	28-3-2024

## Distribution history:

Version	Distributed to	Comments	Date
1	Fontys Teachers	First portfolio delivery	14-4-2024

## Table of Contents

Introduction .....	4
Problem .....	4
Research questions .....	4
What architectural patterns best suit the requirements of my system? .....	4
Which architecture styles exist and what are some pros and cons? .....	4
Layered Pattern .....	5
Client-Server pattern .....	5
Event-Driven pattern .....	5
Microkernel pattern .....	6
Microservices .....	6
Broker pattern .....	6
Event-Bus pattern .....	6
Pipe-filter pattern .....	8
Blackboard pattern .....	8
Component-based pattern .....	8
Service-oriented architecture .....	9
Monolithic architecture .....	9
Space-based architecture .....	9
Which architecture patterns are most suitable for a web application that fits the requirements? .....	10
Conclusion .....	11
How do different services or components communicate with each other? .....	12
What are the different types of microservice communication and what are the advantages? .....	12
Synchronous communication .....	12
Asynchronous Communications .....	12
Which of these communication types best suit my system and where? .....	13
Conclusion .....	13
How can I ensure that my application meets GDPR requirements? .....	14
1. Check whether I can process personal data .....	14
2. A customer can manage their account .....	15
3. Inform users of their privacy rights .....	15
4. Minimize the amount of data stored .....	16
Conclusion .....	16
How can I ensure that my application meets OWASP requirements? .....	17
Broken Access control .....	17
Cryptographic failures .....	18

Injection.....	19
Insecure Design .....	20
Security misconfiguration .....	21
Vulnerable and outdated components.....	21
Identification and authentication failures .....	22
Software and Data integrity failures .....	23
Security logging and monitoring failures .....	23
Server side request forgery .....	24
How to deploy a scalable application? .....	25
How to dockerize an application? .....	25
How to run an application on Kubernetes? .....	26
How to deploy our Kubernetes environment in Azure? .....	28
Conclusion .....	29
Bibliografie .....	30

## Introduction

### Problem

The sheer number of design and architecture patterns available is overwhelming. Each seems to offer a perfect solution, but selecting the right one for a system designed to handle a million users feels like navigating a maze. The core challenge lies in striking a delicate balance my non-functional requirements..

Through research, I aim to identify the ideal design patterns and architectural approaches that can overcome these challenges. This will help me build a system that is not only scalable but also remains understandable and manageable for future development and maintenance.

### Research questions

**Main Research:** How to design and deploy a web application that enables user-submitted system performance evaluation for game compatibility, ensuring scalability, security, reliability, maintainability, and high performance?

**Sub Questions:**

- What architectural styles best suit the requirements of my system?
- How do different services or components communicate with each other?
- How can I ensure that my application meets GDPR requirements?
- How can I ensure that my application meets OWASP requirements?
- How to deploy a scalable application?
- How do I make sure the application is reliable deployed without breaking issues?
- How do I make sure the application is reliable running in the cloud?

## What architectural patterns best suit the requirements of my system?

There are a lot of different architectural styles you need to consider when creating your own web application. Especially when looking at requirements like scalability, security, reliability, maintainability and performance. First we will have a look at the different architecture styles that exist, and we will compare these with each other to see which best suit the requirements of the system.

### [Which architecture styles exist and what are some pros and cons?](#)

There exists an abundance of architectural styles, making it impractical to delve deeply into each one. Therefore, I will focus on the 13 most widely utilized patterns, as they possess sufficient support and documentation. Other styles, lacking in such resources, will not be explored in this context. The primary source I will be using is an article by (Ritvik Gupta, 2023).

The patterns I will be researching are:

- Layered pattern

- Client-server pattern
- Event-Driven pattern
- Microkernel pattern
- Microservice pattern
- Broker pattern
- Event-bus pattern
- Pipe filter pattern
- Blackboard pattern
- Component-based pattern
- Service-oriented architecture
- Monolithic architecture
- Space-based architecture

#### Layered Pattern

The layered pattern (Kaseb, 2022) is a pattern where each responsibility of the application is separated in layers. For example an API might consist of a presentation layer (controllers), Business layer and data access layer. This modular approach promotes segregation of concerns and facilitates easy maintenance and scalability in a complex system.

Pros	Cons
Promotes separation of concerns	Overhead
Encourages modularity	
Supports parallel development	

#### Client-Server pattern

The Client-Server pattern (Gayantha, 2020) is a fundamental architectural design that divides an application into two distinct components: the client and the server. This separation enables efficient communication and enhances scalability and maintainability. In this pattern, the client is typically a user interface or application that interacts directly with the end user, while the server is responsible for managing data processing, storage, and business logic.

Pros	Cons
Scalability	Data manipulation
Security	
Reliability	

#### Event-Driven pattern

The Event-Driven pattern enables systems to respond dynamically to events, such as user interactions or data updates. Events are emitted by producers and consumed by handlers, allowing for asynchronous, loosely-coupled communication. This pattern fosters flexibility, scalability, and real-time responsiveness in software architectures.

Pros	Cons
Asynchronous	Complexity
Scalable	Debugging
Modular	Overhead
Scalable	

### Microkernel pattern

The microkernel architecture pattern (Heusser, 2020) is a software design approach that sits between monolithic and microservices architectures. It is characterized by a central core system that contains only the essential components necessary to run the application, and additional functionalities are implemented as plugins. This architecture allows for a high degree of modularity and flexibility, as new features or modifications can be added or changed without altering the core system. The core system and plugins communicate through interprocess communication mechanisms provided by the microkernel, ensuring that they remain isolated from each other.

Pros	Cons
Modularity	Performance overhead
Flexibility	Complexity
Stability	Dependent on kernel

### Microservices

Microservices architecture (IBM, n.d.) is a cloud-native approach where a single application is composed of many loosely coupled and independently deployable smaller components or services. These services are organized by business capability, often referred to as a bounded context. This architecture allows for easier code updates, as new features or functionality can be added without affecting the entire application. Teams can use different stacks and programming languages for different components, and components can be scaled independently, reducing waste and cost associated with scaling entire applications.

Pros	Cons
Fault isolation	Complexity
Independent scaling	Resource intensive
Smaller and faster deployment	Data inconsistency
Scalability	

### Broker pattern

The Broker Pattern (Wikipedia, 2023) is an architectural pattern used to structure distributed software systems with decoupled components that interact by remote procedure calls. It involves an intermediary software entity, known as a broker, which is responsible for coordinating communication between clients and servers. The broker acts as a middleman, receiving messages from one component and forwarding them to the appropriate recipient. This pattern allows components to remain decoupled and focused on their own responsibilities, while still being able to communicate and collaborate with other components in the system.

Pros	Cons
Decoupling	Single point of failure
Scalability	Complexity
Changeability	Performance overhead

### Event-Bus pattern

The Event Bus pattern is a design pattern that facilitates communication between components in a distributed system by using an event bus as a central hub. This pattern is particularly useful in large-scale applications where components need to interact without being tightly coupled, adhering to principles of loose coupling and separation of concerns. The event bus acts as a pipeline, where components (referred to as subscribers) can

register to receive specific types of events. When an event occurs, it is dispatched to the event bus, which then forwards it to all registered subscribers that are interested in that type of event. This mechanism allows components to communicate asynchronously, enabling them to operate independently and react to events as they occur.

<b>Pros</b>	<b>Cons</b>
Loose coupling	Complexity
Asynchronous	Debugging
Scalability	Performance overhead

### Pipe-filter pattern

The Pipe and Filter pattern is an architectural pattern that structures a system as a sequence of processing elements, where each element is a filter that performs a specific operation on the data. Data flows through the system in a pipeline, where each filter takes input from the previous filter and passes its output to the next filter in the sequence. This pattern is particularly useful for data processing and transformation tasks, where operations can be performed in a series of steps.

Pros	Cons
Modularity	Complexity
Flexibility	Latency

### Blackboard pattern

The Blackboard pattern is a way to solve complex problems by breaking them down into smaller, manageable parts. It involves a central "blackboard" where different parts of a program, called "agents," can share information and work together to find a solution.

Pros	Cons
Modularity	Complexity
Flexibility	Coordination
Collaboration	Security

### Component-based pattern

Component-based architecture is a software design approach that structures applications into reusable, modular components. Each component encapsulates specific functionality and communicates with others through well-defined interfaces, promoting decoupling and flexibility. This architecture enhances development efficiency, reliability, and scalability by facilitating the reuse of components across different parts of an application or even in different applications. It supports the creation of extensible, encapsulated, and independent components that can be easily replaced or extended without significant disruption to the overall system. Despite its advantages, component-based architecture may not be suitable for every scenario, especially when applications are large or when the need for customization limits the reusability of components.

Pros	Cons
Modularity	Performance issues
Easy maintenance	
Easy testing	



### Service-oriented architecture

Service-Oriented Architecture (SOA) is an approach to designing software systems where different functions or services are organized to communicate and work together effectively. In SOA, services are like building blocks, each performing specific tasks and capable of independent deployment. These services encapsulate multiple functionalities within a specific domain. SOA promotes flexibility, scalability, and interoperability by breaking down complex systems into manageable parts, though the services are typically larger and encompass more functionalities compared to microservices.

Pros	Cons
Reusability	Complexity
Easy maintenance	
Flexibility	

### Monolithic architecture

Monolithic architecture refers to a traditional approach to software design where all components of an application are tightly integrated into a single, indivisible unit. In this model, the entire application, including its user interface, business logic, and data access layers, is developed and deployed as a single unit.

Pros	Cons
Simplicity	Scalability
Easy of development	Maintainability
	Deployment

### Space-based architecture

Space-based architecture is an architectural pattern where data is distributed across a grid of interconnected nodes, often referred to as a "space." This architecture is characterized by its ability to scale horizontally and handle large volumes of data by partitioning and replicating data across multiple nodes. It allows for high availability, fault tolerance, and scalability, making it suitable for applications with demanding performance requirements. Space-based architectures are commonly used in distributed caching, data grids, and real-time analytics systems.

Pros	Cons
Scalability	Complexity
Performance	Data Consistency
	Resource Overhead

### Which architecture patterns are most suitable for a web application that fits the requirements?

Now that we have taken a look at which styles there are, we need to decide which ones fit our requirements best. We will be making a highly scalable, maintainable, reliable, performant and secure application. Keeping this in mind we will choose which patterns to use, so down below there is a table with all above patterns and if they are applicable to my application.

Pattern	Applicable	Description
Microservice pattern	Yes	This pattern is central to my requirements. It decomposes CIRI2 into smaller loosely coupled services, each responsible for a specific business function. This allows for independent scaling,
Layered pattern	Yes	This pattern can be applied within each microservice to organize its internal architecture, separating concerns such as presentation, business logic and data access. It aids maintainability.
Client-Server pattern	Yes	This pattern is implicit in my architecture, as I will be using a SPA that serves as client side, and my microservices that act as a server side. It ensures clear separation of concerns and support scalability
Event-Driven pattern	Yes	Using this pattern for communication between my microservices would be beneficial for ensuring loose coupling and scalability. Each microservice will then communicate asynchronously with each other.
Broker pattern	Yes	For communication between my microservices a broker pattern would be ideal, as this helps decoupling and scalability.
Component-based pattern	Yes	Because I will be using a SPA my frontend needs to stay modular and use reusable component, this can help with maintainability and extensibility
Service-oriented architecture	Yes	While microservices are a form of SOA, explicitly following SOA principles can help in designing services that are loosely coupled, reusable, and interoperable. It emphasizes service abstraction, standard interfaces, and service composition, which align with the goals of scalability, reliability, and maintainability.
Space-based architecture	No	Space-based architecture is an alternative to traditional microservices architecture that focuses on distributing data and processing across multiple nodes in a shared memory space. Unlike microservices, which involve separate services communicating via APIs, space-based architecture uses a centralized space for communication and coordination.
Micro-Kernel pattern	No	Because the application needs to be scalable and reliable, I don't want a central kernel. I want everything to be completely standalone, this pattern might fit better for a operating system.
Pipe-filter pattern	No	Pipe filter pattern doesn't really address the requirements of the system. This is because I won't be doing that much processing on my data.

BlackBoard-Pattern	No	Blackboard pattern has a centralized nature, which will introduce scalability and reliability challenges. It also holds shared domain knowledge, when you want to keep this separated.
Monolithic architecture	No	Monolithic are a centralized way to handle you application, this absolutely does not fit the goals I am trying to meet. Monolithic application have challenges in scalability agility, and flexibility. And also comes with other challenges, like big deployments etc.

## Conclusion

In conclusion, after carefully evaluating various architecture patterns, we have identified those that best meet the specific requirements of our web application. Our aim is to build a highly scalable, maintainable, reliable, performant, and secure application. The selected patterns align with these goals, providing a robust framework for development and future growth.

1. **Microservice Pattern:** Central to our architecture, this pattern breaks down the application into smaller, loosely coupled services. Each service handles a specific business function, allowing for independent scaling, deployment, and development, thus enhancing maintainability and reliability.
2. **Layered Pattern:** Applied within each microservice, this pattern helps organize the internal architecture by separating concerns such as presentation, business logic, and data access. This separation aids in maintainability and supports clean code practices.
3. **Client-Server Pattern:** Implicit in our architecture, this pattern underpins the structure of our single-page application (SPA) and the backend microservices. It ensures a clear separation of concerns and supports scalability by allowing independent scaling of client and server components.
4. **Event-Driven Pattern:** Beneficial for inter-microservice communication, this pattern ensures loose coupling and scalability. By enabling asynchronous communication, it enhances the responsiveness and reliability of the system.
5. **Broker Pattern:** Ideal for facilitating communication between microservices, this pattern helps achieve decoupling and scalability. It acts as an intermediary, managing service interactions efficiently.
6. **Component-Based Pattern:** Essential for the modularity of our SPA, this pattern supports the development of reusable components. It enhances maintainability and extensibility, allowing for easier updates and feature additions.

**Service-Oriented Architecture (SOA):** By explicitly following SOA principles, we ensure that our services are loosely coupled, reusable, and interoperable. This pattern emphasizes service abstraction, standard interfaces, and service composition, aligning perfectly with our goals of scalability, reliability, and maintainability.

These patterns collectively create a solid foundation for our web application, ensuring that it meets the demanding requirements of modern web development. By adhering to these well-established patterns, we can build a system that is not only robust and secure but also flexible and scalable to adapt to future needs.

## How do different services or components communicate with each other?

What are the different types of microservice communication and what are the advantages?

### Synchronous communication

In synchronous communication, the client waits for a response from the service before proceeding. This is typically done using request-response protocols.

#### Types:

##### 1. Http/REST

- **Description:** Uses standard HTTP methods (GET, POST, PUT, DELETE) and usually involves JSON or XML payloads.
- **Advantages:**
  - Simple and widely understood.
  - Leveraging existing web infrastructure.
  - Easy to debug with standard tools like Postman or cURL.
  - Well-suited for CRUD operations.

##### 2. gRPC:

- **Description:** A high-performance, open-source universal RPC framework using Protocol Buffers for serialization.
- **Advantages:**
  - Efficient and faster than REST (binary serialization).
  - Strongly typed interfaces.
  - Supports bi-directional streaming.

### Asynchronous Communications

#### Types:

##### 1. Message Brokers (e.g., RabbitMQ, Kafka):

- **Description:** Messages are sent to a broker, which routes them to appropriate services.
- **Advantages:**
  - Decouples message producers and consumers.
  - Supports various messaging patterns (pub/sub, fan-out).
  - Enhances scalability and fault tolerance.

##### 2. Event Streaming (e.g., Apache Kafka, Amazon Kinesis):

- **Description:** Services publish events to a stream, which can be consumed by multiple services.
- **Advantages:**
  - Real-time data processing.
  - Highly scalable and fault-tolerant.
  - Allows for event sourcing and CQRS (Command Query Responsibility Segregation).

### Which of these communication types best suit my system and where?

For my application I will be creating multiple microservices. You can find more information about my microservices in the System architecture document. Each microservice might take advantage of different communication strategies. We will look at each application and what is important for this application to make a decision of which communication strategy to use:

#### **Frontend**

The frontend will talk with the gateway and external api's like auth0, that is why the frontend will need to communicate in a very standardized way. This is why I will be using REST in the frontend.

#### **Gateway**

The gateway should have an interaction layer that is very open, and a lot of services can communicate with. The frontend will talk with the microservices via the gateway. Because the gateway needs to have a standard way of communicating. The gateway has 2 sides, the client facing side, and the microservice facing side. The client facing side will be using REST as this is the standard way communication happens synchronously, it makes it easy for the frontend to communicate with the gateway. The microservice facing side, can use multiple strategies, in my case it will use a combination of REST and a message queue. The REST endpoints will be used for synchronous communication and the message queue for asynchronous.

#### **Microservices**

The microservices will be using a combination of REST and Message queue, because a microservice needs both asynchronous and synchronous communication. When a microservice needs to return data to a frontend or the gateway immediately it will be using REST. If a microservice needs to get data from another microservice it will be using a message queue, because a message queue gives a standard and external way to handle the communication. Which means that even if the service is done, it will be processed in a later date.

### Conclusion

In conclusion, after evaluating the communication needs of each component in our microservices architecture, we have determined the most suitable communication strategies for our system. Here's a summary of our decisions:

1. Frontend: The frontend will communicate with the gateway and external APIs (e.g., Auth0) using REST. This standardized approach ensures reliable and straightforward synchronous communication, making it easy to interact with various services.
2. Gateway: The gateway serves as an intermediary between the frontend and the microservices. It will employ REST for the client-facing side, ensuring standardized and synchronous communication with the frontend. For the microservice-facing side, a combination of REST and a message queue will be used. REST will handle synchronous communication, while the message queue will facilitate asynchronous interactions, enhancing the system's scalability and reliability.
3. Microservices: Each microservice will utilize both REST and message queues. REST will be used for synchronous communication when immediate data exchange is

required, such as responses to the frontend or the gateway. Message queues will be employed for asynchronous communication between microservices, allowing for efficient and decoupled interactions, ensuring that tasks can be processed independently and reliably.

This hybrid approach, combining REST for synchronous communication and message queues for asynchronous communication, optimally supports the diverse requirements of our microservices architecture, ensuring scalability, reliability, and maintainability.

## How can I ensure that my application meets GDPR requirements?

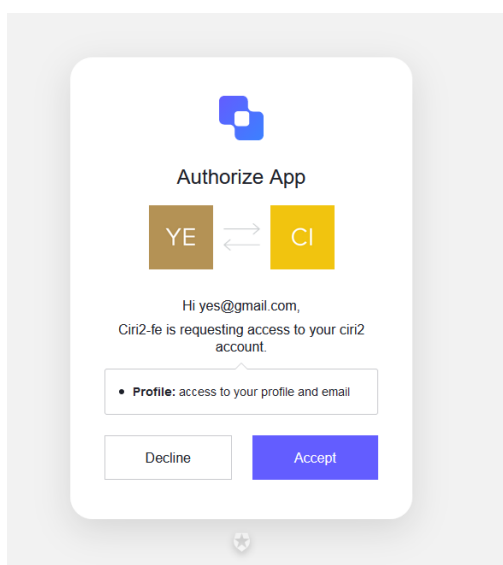
To answer this question I will be using a source supplied by the government (Netherlands Enterprise Agency, n.d.). I will go through the checklist and make sure that my application meets given requirements.

### 1. Check whether I can process personal data

I can process personal data if I fall in one of the following circumstances:

1. I have permission from the person involved
2. I need the data to fulfil an agreement
3. I need the data to meet legal obligations
4. I need the data to protect someone's life or health, and can't ask the person for permission
5. I have a justified cause for processing the data

Of these circumstances, none but the first one match my application. That is why I need to ask the user if they give permission to process their data. I use auth0 for my application, which shows the user what I will use of their data:

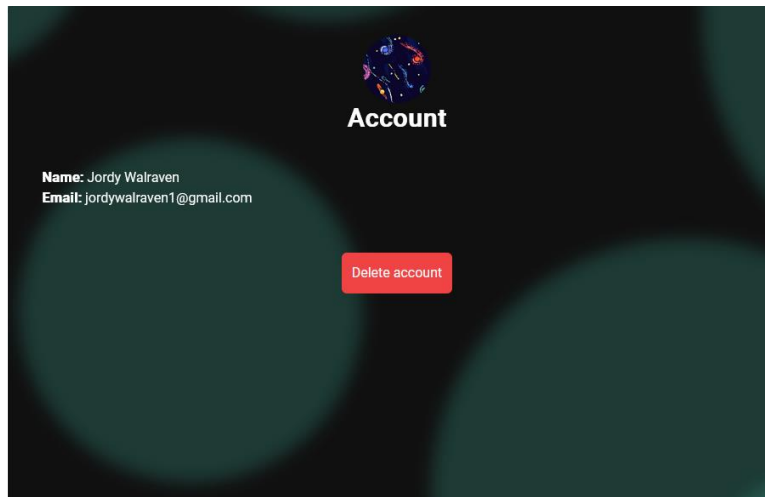


## 2.A customer can manage their account

A customer has multiple rights they can use. A user should be able to do the following things:

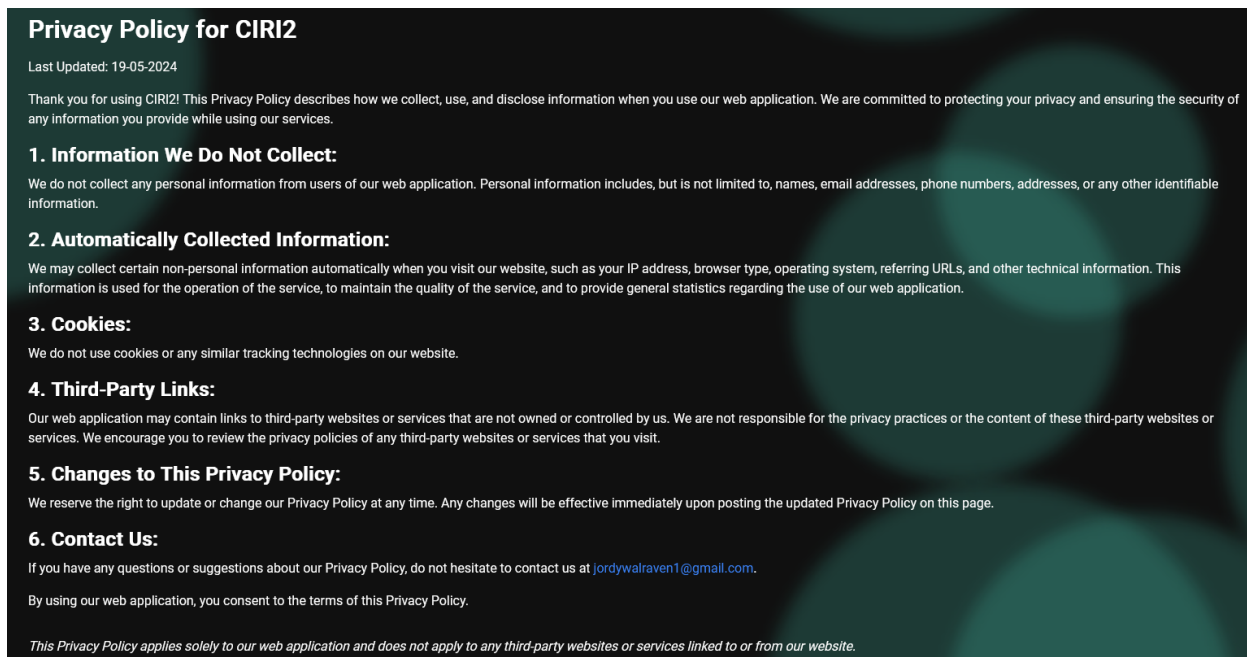
- View and Delete their account
- Request their data

In my application I am storing no user data, but an account does get created on auth0, on my profile page I will show all the user data I have. And Make the user be able to dele their account to adhere to this requirement:



## 3.Inform users of their privacy rights

To inform users of their privacy rights I created a privacy policy. In here the user can find information about their privacy:



#### 4. Minimize the amount of data stored

For my application I am not storing any data myself, and the user data stored by auth0 is very minimal, and only the minimal required information for the application to work.

#### Conclusion

In conclusion, ensuring compliance with GDPR requirements is paramount for any application handling personal data. By diligently following established guidelines, such as those provided by authoritative sources like the Netherlands Enterprise Agency, and systematically assessing our application against key criteria, we can confidently safeguard user privacy and meet legal obligations.

Our approach involves thorough scrutiny across various fronts:

1. **Processing Personal Data:** We recognize the importance of obtaining explicit consent from users before processing their personal data, as outlined by GDPR. Leveraging Auth0, we transparently inform users about the data we collect and seek their permission, ensuring compliance with this fundamental principle.
2. **Customer Account Management:** Empowering users with control over their accounts aligns with GDPR's emphasis on data subject rights. While we leverage Auth0 for account management and store minimal user data, we enable users to view, delete, and request their data, promoting transparency and accountability.
3. **Informing Users of Privacy Rights:** Our commitment to transparency extends to informing users about their privacy rights through a comprehensive privacy policy. This document serves as a guide for users to understand how their data is handled and their rights concerning its usage.
4. **Minimizing Data Storage:** Recognizing the importance of data minimization, we refrain from storing unnecessary user data. Leveraging Auth0's minimal data storage practices, we ensure that only essential information required for application functionality is retained, thereby reducing the risk associated with data storage.

By adhering to these principles and continuously monitoring regulatory updates, we can uphold the highest standards of data protection and foster trust with our users, ensuring that our application remains compliant with GDPR requirements now and in the future.



## How can I ensure that my application meets OWASP requirements?

I will be researching the OWASP top 10 to make sure my application is protected against the most critical security risks. The OWASP top 10 is as follows (Foundation, 2021):

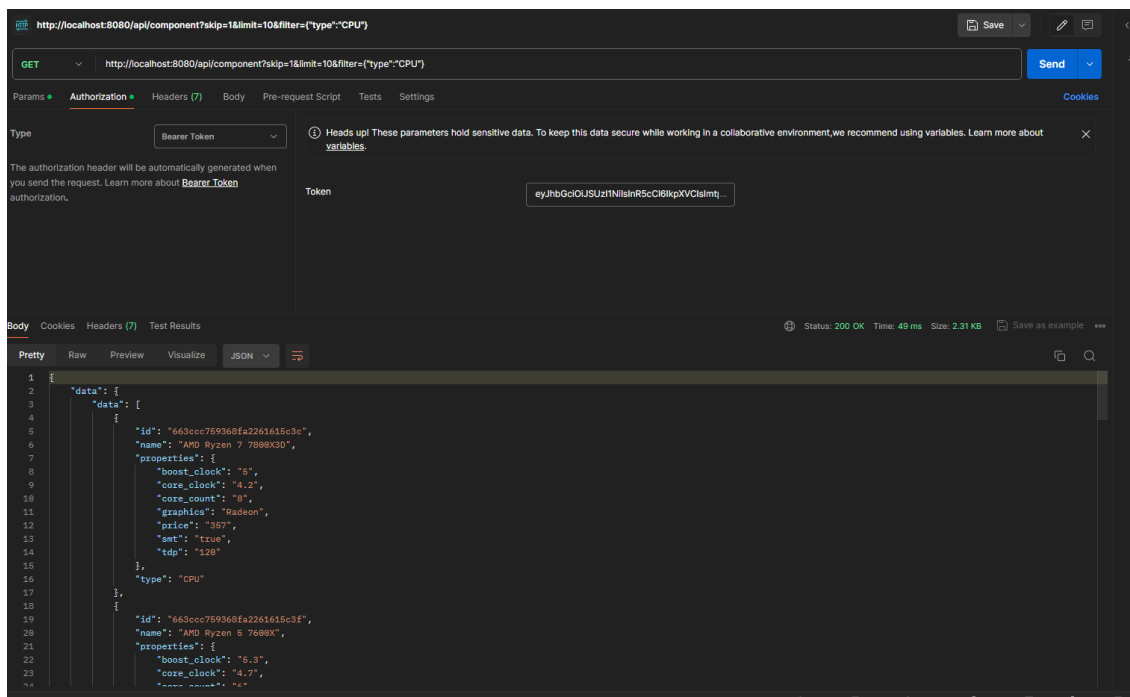
- Broken Access Control
- Cryptographic failures
- Injection
- Insecure Design
- Security Misconfiguration
- Vulnerable and Outdated components
- Identification and authorization failures
- Software and data integrity failures
- Security, logging and monitoring failures
- Server side request forgery

I will be handling each of these topics:

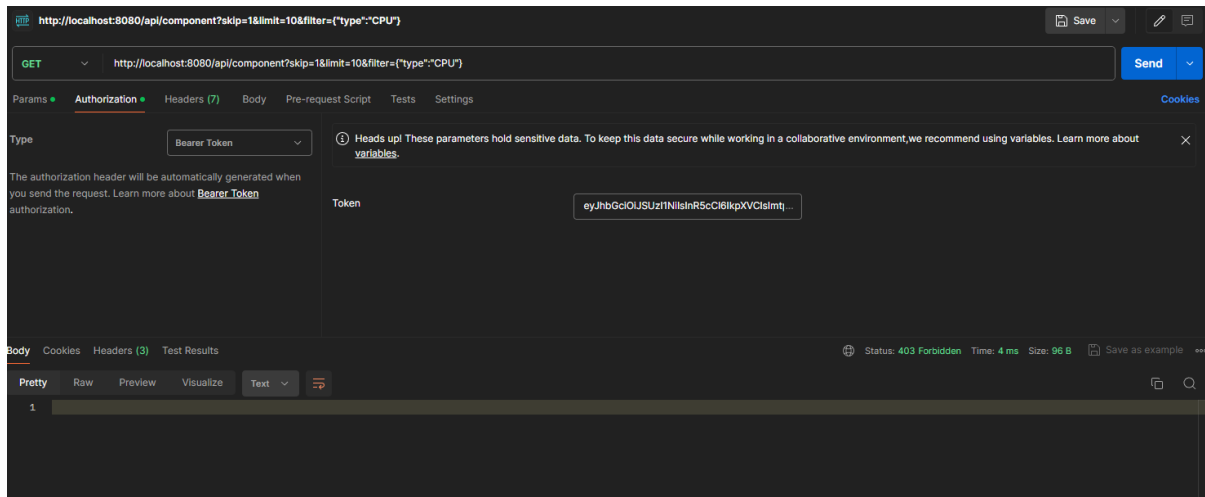
### Broken Access control

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits (Foundation, 2021)

For my application I use auth0 for access control. I do this by getting the jwk public secret from auth0 which I use to verify my JWT. This means that users can't tamper with it as they don't know the signing secret. Meaning that if they change it will fail. I also use the permission system of auth0 and integrated that into my gateway. Down below a screenshot of a user with required permissions and without required permissions:



### Account with permissions



### Account without required permissions

I also rate limited my api gateway, meaning that attempts to crack something will be rate limited decreasing the chance of a breach.

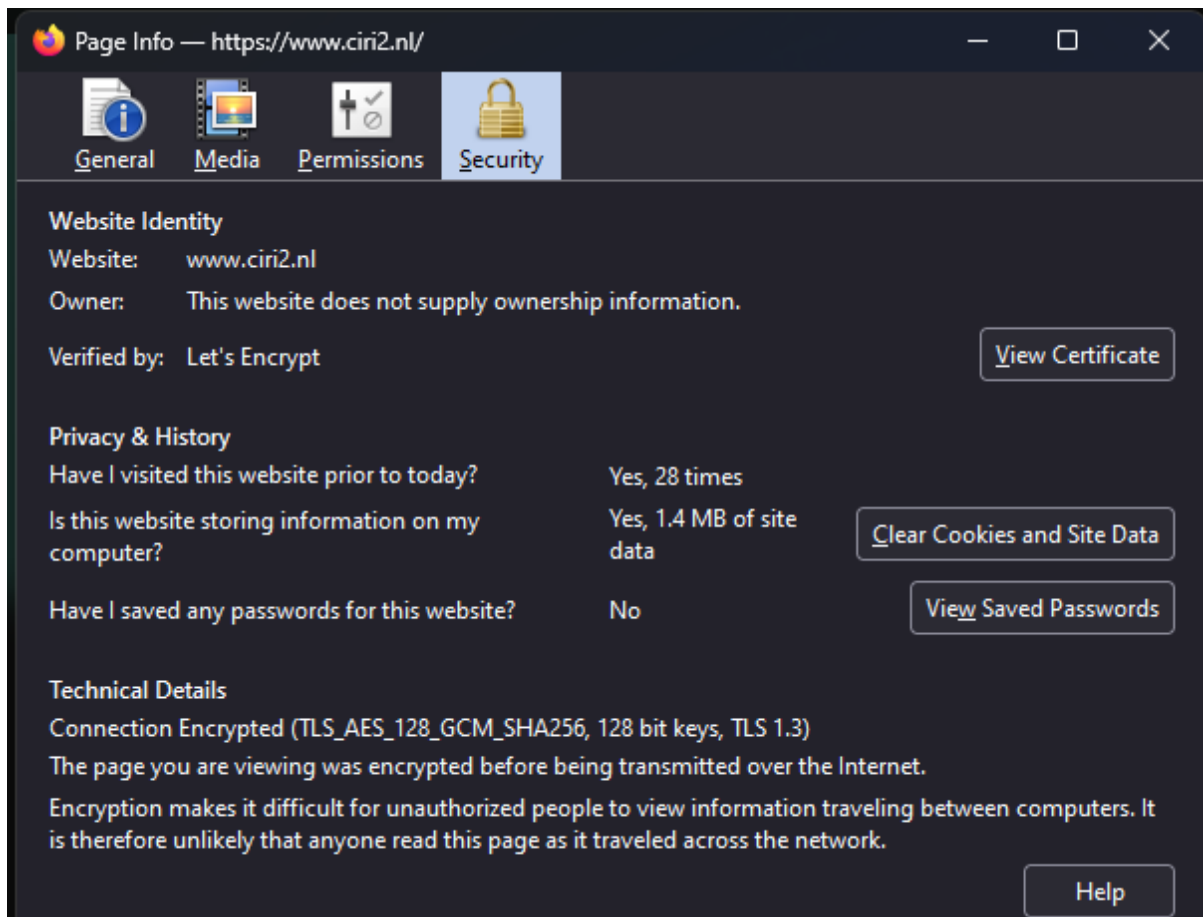
```
"qos/ratelimit/router": {
  "max_rate": 50,
  "capacity": 50,
  "client_max_rate": 5,
  "client_capacity": 5,
  "every": "10m",
  "strategy": "ip"
},
```

### Cryptographic failures

Cryptographic failures encompass a range of vulnerabilities and weaknesses in the implementation and use of cryptographic techniques. These failures can include the use of weak algorithms, improper key management, insecure storage of keys or secrets, and misconfigurations in cryptographic APIs. Such weaknesses can lead to serious security risks, including data breaches, unauthorized access, and compromised system integrity.

In my application I don't send classified information and all information isn't that important. To prevent some cryptographic failures I work with TLS and https, I implemented this to fix man in the middle attacks, because it encrypts the data. I get my TLS secret from let's encrypt:

<a href="#">cert-manager-webhook</a>	cert-manager
<a href="#">cert-manager-cainjector</a>	cert-manager
<a href="#">cert-manager</a>	cert-manager



## Injection

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

For my application I am using MongoDB, with bson. The bson library has an injection vulnerability for the \$where command when using javascript comparisons. For my applications I use go with bson, and never compare a javascript function. I also use LIMIT and validate the input of the users. This is why my application is safe against Injection.

## Insecure Design

Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top 10 risk categories. There is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes and remediation. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

To prevent this issue I have multiple security checks at different tiers of the application. In my application I have unit testing and E2E testing, to test the complete flow of my application. I also have Sonarcloud analysis that looks at the static code I push for vulnerabilities. I have a dependabot that looks at outdated dependencies, or dependencies with vulnerabilities and fixes them. Other than this I also run the ZAP security scan, that checks for vulnerabilities in my frontend and backend.

```
PASS: Vulnerable JS Library (Powered by Retire.js) [10003]
PASS: In Page Banner Information Leak [10009]
PASS: Cookie No HttpOnly Flag [10010]
PASS: Cookie Without Secure Flag [10011]
PASS: Re-examine Cache-control Directives [10015]
PASS: Cross-Domain JavaScript Source File Inclusion [10017]
PASS: Content-Type Header Missing [10019]
PASS: Information Disclosure - Debug Error Messages [10023]
PASS: Information Disclosure - Sensitive Information in URL [10024]
PASS: Information Disclosure - Sensitive Information in HTTP Referrer Header [10025]
PASS: HTTP Parameter Override [10026]
PASS: Information Disclosure - Suspicious Comments [10027]
PASS: Open Redirect [10028]
PASS: Cookie Poisoning [10029]
PASS: User Controllable Charset [10030]
PASS: User Controllable HTML Element Attribute (Potential XSS) [10031]
PASS: Viewstate [10032]
PASS: Directory Browsing [10033]
PASS: Heartbleed OpenSSL Vulnerability (Indicative) [10034]
PASS: Strict-Transport-Security Header [10035]
PASS: Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) [10037]
PASS: X-Backend-Server Header Information Leak [10039]
PASS: Secure Pages Include Mixed Content [10040]
PASS: HTTP to HTTPS Insecure Transition in Form Post [10041]
PASS: HTTPS to HTTP Insecure Transition in Form Post [10042]
PASS: User Controllable JavaScript Event (XSS) [10043]
PASS: Big Redirect Detected (Potential Sensitive Information Leak) [10044]
PASS: Retrieved from Cache [10050]
PASS: X-ChromeLogger-Data (XCOLD) Header Information Leak [10052]
PASS: Cookie without SameSite Attribute [10054]
```

### Security misconfiguration

OWASP Security Misconfiguration refers to inadequate or improperly implemented security settings that can lead to vulnerabilities in web applications. This includes default configurations, incomplete configurations, open cloud storage, misconfigured HTTP headers, or verbose error messages that reveal sensitive information. Such misconfigurations can be exploited by attackers to gain unauthorized access to systems, data, or administrative functions, leading to potential breaches. Ensuring proper security configuration involves regular reviews, updates, and testing of security settings across the application, software, and underlying infrastructure.

To ensure there were no security misconfigurations in my application, I meticulously went through all aspects of my setup and verified the following:

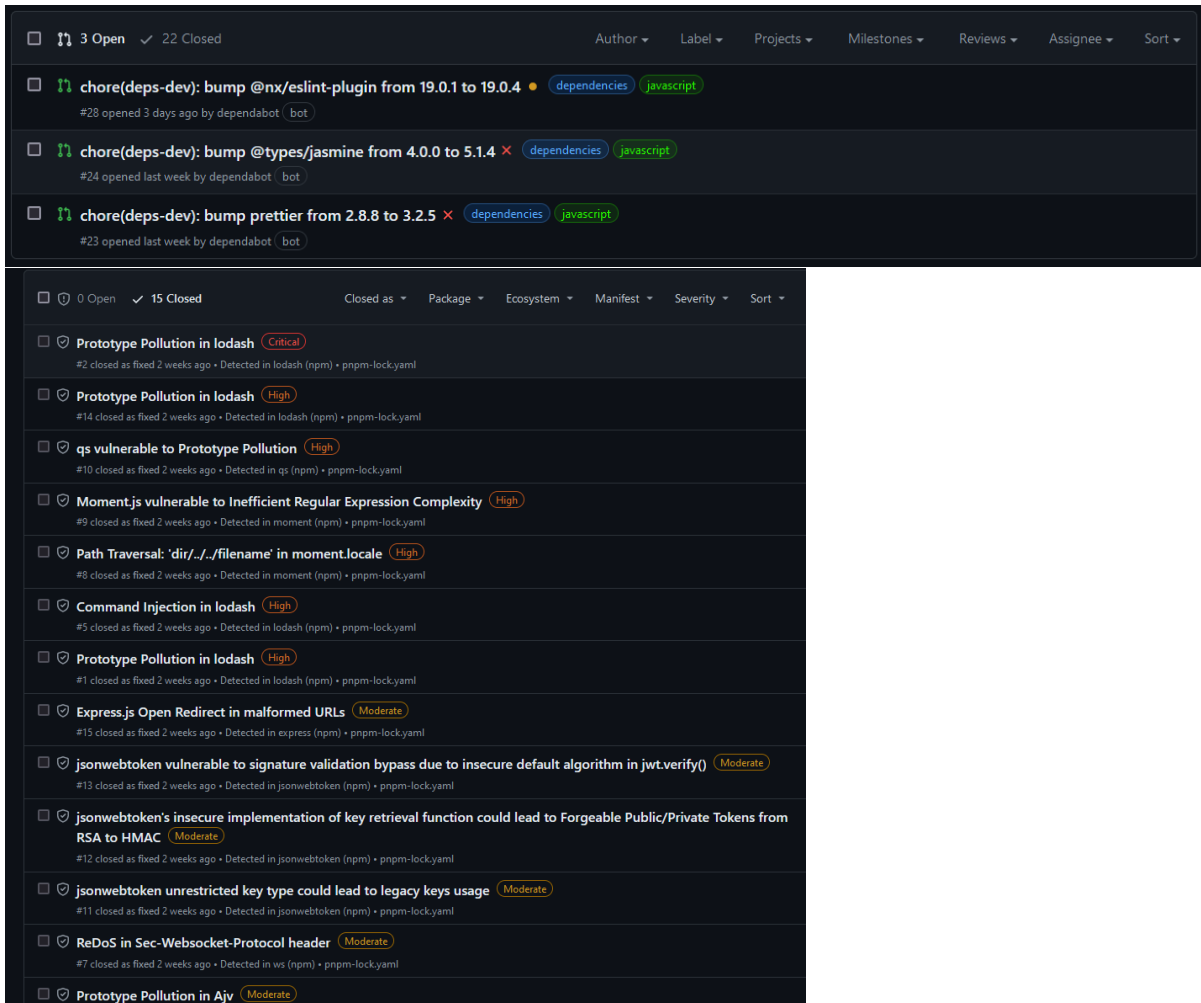
- E2E Test User Security: My end-to-end test user has a robust, secure password to prevent unauthorized access during testing.
- Database Security: My MongoDB cluster is protected with a strong, secure password to safeguard sensitive data against breaches.
- HTTP Header Validation: My HTTP headers are checked and validated by my gateway, ensuring they are configured correctly to protect against common web vulnerabilities like XSS and clickjacking.
- Dependency Management: All components of my application are always up to date, facilitated by Dependabot, which automatically manages and updates dependencies to patch any known security vulnerabilities.
- Minimized Attack Surface: Unused ports, services, pages, and accounts have been identified and disabled to reduce potential entry points for attackers.

By thoroughly reviewing and addressing these areas, I can confidently validate that my application's configurations are secure and adhere to best practices, significantly reducing the risk of security misconfigurations.

### Vulnerable and outdated components

OWASP Vulnerable and Outdated Components refers to the risk associated with using software libraries, frameworks, and other components that have known vulnerabilities or are no longer supported by their maintainers. These outdated or insecure components can be exploited by attackers to gain unauthorized access, execute malicious code, or cause system failures. Ensuring that all components are regularly updated and patched is critical to maintaining the security of an application and protecting it from potential threats. Regular monitoring, vulnerability scanning, and using tools to manage dependencies are essential practices to mitigate this risk.

To prevent vulnerable and outdated components of existing in my application I use dependabot, this dependabot scans my repository for outdated components.



I have already fixed a lot of vulnerable and outdated dependencies.

### Identification and authentication failures

OWASP Identification and Authentication Failures refer to weaknesses in the mechanisms used to identify and authenticate users, leading to unauthorized access to systems and data. These failures include inadequate password policies, weak credential storage, flawed session management, and improper implementation of authentication protocols. Such vulnerabilities can be exploited by attackers to impersonate users, gain unauthorized access, and escalate privileges. Ensuring robust identification and authentication involves enforcing strong password policies, secure storage of credentials, proper session handling, and multi-factor authentication to enhance security and prevent unauthorized access.

To prevent this I use auth0 in my application. In auth0 things like brute forcing logins, login failures, alerts and other things get managed. On my server side I also verify the JWT with a credential provided by auth0 to make sure the JWT is valid. Auth0 also requires the user to make pretty strong passwords, as weak passwords aren't allowed. For my api gateway I also have a rate limit that stops brute force attacks for tempered jwt's. My application also isn't deployed with any default credentials and passwords. The admin accounts all have secure and long cryptographically generated passwords.

#### Software and Data integrity failures

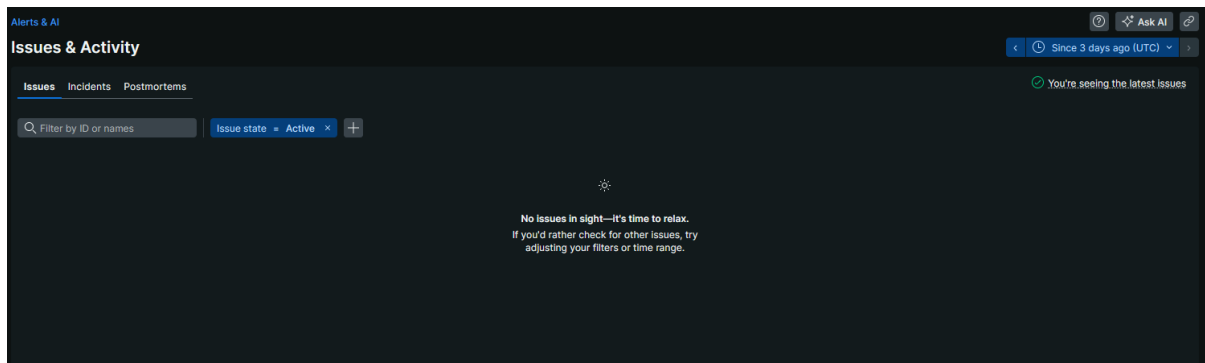
Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.

Dependabot takes care of the dependency checking for NPM packages, meaning that my application only consumes trusted npm packages. Also for code changes a review is required, meaning that not everyone can just push the code to main. For releasing the code on azure a admin needs to release a tag. My CI/CD also has proper access control, with each pipe having it's own environment in which it runs.

#### Security logging and monitoring failures

OWASP's 'Security Logging and Monitoring Failures' addresses the critical importance of effective logging and monitoring practices in identifying and responding to security incidents within software systems. It highlights common failures such as insufficient logging, poor log management, ineffective monitoring, ignoring log data, and misconfiguration. By implementing robust logging policies, configuring monitoring tools effectively, and regularly reviewing log data, organizations can enhance their ability to detect and mitigate cyber threats, improving overall security.

For me logging takes place in different log levels. The log levels which my application produces are Info, Debug, Warning and Error. Each application uses this same layout to easily find and monitor the logs. Logging is centralized to new relic, which collects the logs and gives a nice UI to navigate through the logs. You can also see the traces of each webvisit in this log. New relic also looks for issues in the logs with AI:



### Server side request forgery

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

For my application I always sanitize user input data. In my angular application the user isn't allowed to input any url's meaning that they can't request anything from my server, as the only way into my cluster is through my angular application. The application blocks any url's that it doesn't know, hence is why requests won't go through to the backend, that have been tempered with.



### How to deploy a scalable application?

For my application I will be using Azure as a host. I will be using Azure as they have a lot of benefits for students and free credits. To deploy my application I will make use of a combination of technologies. I will be using docker, Kubernetes and Azure.

### How to dockerize an application?

To dockerize an application I will be using docker. (Inc, n.d.) Docker makes use of Dockerfiles, these dockerfiles tell docker what actions to perform to run the application in a virtual machine. Docker has some terms that we need to know, docker makes use of containers and images. An image are the files and the instructions on how to run, while the container is the virtual machine in which the image runs.

The dockerfile for my PC microservice looks like the following:

```
# Use a smaller base image for the builder stage
FROM golang:1.22.1 AS builder

# Set destination for COPY
WORKDIR /app

ENV MONGOURI=${MONGOURI}

# Download Go modules
COPY apps/pc-microservice/go.mod apps/pc-microservice/go.sum ./

# Install ca-certificates
RUN apt-get update && apt-get install -y ca-certificates

RUN go mod download

# Install ca-certificates
RUN apt-get update && apt-get install -y ca-certificates

COPY apps/pc-microservice .

# Build go application
RUN CGO_ENABLED=0 GOOS=linux go build -o ciri2-pc-microservice ./cmd

# Use a minimal base image for the final image
FROM scratch
# Copy the built binary from the builder stage
COPY --from=builder /app/ciri2-pc-microservice /ciri2-pc-microservice

# Copy the SSL certificates from the builder stage
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/

# Expose the port
EXPOSE 4050

# Run the binary
CMD ["/ciri2-pc-microservice"]
```

This file builds the image for my application, which then can be run in a container.

An image can after it has been build, pushed to a repository. For my application I will be using docker hub, as docker hub is a third party service from things like GitHub or azure. Meaning that I can always change my tech stack. After an image has been pushed, it can be pulled from other places to run the docker image. Later on we can pull the dockerfile from dockerhub in our azure environment.

#### [How to run an application on Kubernetes?](#)

To run an application on Kubernetes we first need to have our applications dockerized and published to a repo. We can then start by creating the config files we need for our Kubernetes deployment.

To run Kubernetes locally we will be using Minikube, we can start minikube by installing it and running “minikube start”. This will couple our local kubectl to minikube as well. For our application to work in Kubernetes we need multiple config files. We need a **deployment** file, here we tell Kubernetes which image to run, how many resources to use, the port we need to expose and setting env variables. We then also have a **Service** file, this service file contains information on where to expose the application and on which port. There are multiple types of services, we have a ClusterIp service, LoadBalancer service, and some more. We will be using the ClusterIp mode on any service that needs to talk to another service inside the cluster, but doesn’t need to be reachable from outside. This is where the LoadBalancer comes in, this LoadBalancer service gets assigned a public Ip which we can use to contact it from the outside.

To see how this looks in Kubernetes files I will show the Kubernetes files for the pc-microservice:

## Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: C:\Users\jordy\AppData\Local\Microsoft\WinGet\Packages\Kubernetes.kompose_Microsoft.Winget.Source_8wekyb3d8bbwe\kompose.exe convert
    kompose.version: 1.31.2 (a92241f79)
  labels:
    io.kompose.service: pc-microservice
  name: pc-microservice
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: pc-microservice
  strategy: {}
  template:
    metadata:
      annotations:
        kompose.cmd: C:\Users\jordy\AppData\Local\Microsoft\WinGet\Packages\Kubernetes.kompose_Microsoft.Winget.Source_8wekyb3d8bbwe\kompose.exe convert
        kompose.version: 1.31.2 (a92241f79)
      labels:
        io.kompose.network/ciri2-ciri2-network: 'true'
        io.kompose.service: pc-microservice
    spec:
      containers:
        - env:
            - name: MONGOURI
              value: mongodb://mongo:27017
            - name: PC_PORT
              value: '4050'
          image: jordywalraven/ciri2-pc-microservice:release
          name: pc-microservice
          ports:
            - containerPort: 4050
              protocol: TCP
          resources:
            limits:
              cpu: 500m
              memory: 512Mi
          restartPolicy: Always
status: {}
```

## Service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: C:\Users\jordy\AppData\Local\Microsoft\WinGet\Packages\Kubernetes.kompose_Microsoft.Winget.Source_8wekyb3d8bbwe\kompose.exe convert
    kompose.version: 1.31.2 (a92241f79)
  labels:
    io.kompose.service: pc-microservice
  name: pc-microservice
spec:
  ports:
    - name: '4050'
      port: 4050
      targetPort: 4050
  selector:
    io.kompose.service: pc-microservice
status:
  loadBalancer: {}
```

If we now run `kubectl apply -f .`, you can see that a deployment and service get created in minikube. We can also see that it starts pulling the image that we specified in the deployment, after the image has been pulled. It will start to run the application. We can also set up auto scaling for this application with Horizontal pod autoscalers. The pod autoscaler for the PC-microservice looks like the following:

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: pc-microservice-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: pc-microservice
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50

```

This HPA (horizontal pod autoscaler) tells Kubernetes when a new replica of the application should be created, we also specify the min and max replicas we want to have in our cluster.

### How to deploy our Kubernetes environment in Azure?

To get started with Azure we first need to create a **Resource group**, this resource group will be a collection of all the resources we need for our deployment. In this resource group we will be creating an Kubernetes cluster, we can create this cluster with the Azure UI:

Home > Kubernetes services >

## Create Kubernetes cluster

Basics Node pools Networking Integrations Monitoring Advanced Tags Review + c

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more](#)

**Project details**  
Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \*

Resource group \*  [Create new](#)

**Cluster details**  
Cluster preset configuration \*   
To quickly customize your Kubernetes cluster, choose one of the preset configurations above. You can modify these configurations at any time. [Compare presets](#)

Kubernetes cluster name \*

Region \*

Availability zones

AKS pricing tier

Kubernetes version \*

[Previous](#) [Next](#) [Review + create](#)

We need to give this cluster a name and select the resource group. After we created the Kubernetes cluster we can click the connect button in azure. This will give is instructions on how to connect my local environment to Azure. After connecting the Azure environment to my local environment I can run Kubectl on the Azure environment. This means that I can just run kubectl apply -f . , and the application will be deployed to Azure.

## Conclusion

Deploying a scalable application involves integrating several advanced technologies to ensure efficient, reliable, and flexible application management. In this guide, we focused on using Azure as the hosting platform, leveraging its benefits for students and free credits. The deployment process is structured around three core components: Docker, Kubernetes, and Azure.

### **Dockerizing the Application:**

Docker simplifies the process of creating, deploying, and running applications in containers. By writing a Dockerfile, we define the necessary actions to set up our application within a virtual machine-like environment. The Docker image, built from the Dockerfile, encapsulates the application and its dependencies, enabling consistent and isolated environments. Pushing these images to Docker Hub allows for seamless integration and flexibility, making it possible to change the tech stack as needed.

### **Running the Application on Kubernetes:**

Kubernetes orchestrates the deployment, scaling, and management of containerized applications. With Docker images published to a repository, we create Kubernetes configuration files to define deployments and services. Using Minikube for local testing, we simulate a Kubernetes environment to ensure everything functions correctly. Deployments manage the desired state of application replicas, while services handle internal and external accessibility. Implementing a Horizontal Pod Autoscaler (HPA) further enhances scalability by dynamically adjusting the number of replicas based on demand.

### **Deploying to Azure Kubernetes Service (AKS):**

Azure simplifies the process of deploying Kubernetes clusters through its user-friendly interface. By creating a resource group and a Kubernetes cluster in Azure, we establish a robust environment for our application. Connecting the local development setup to Azure allows for straightforward deployment using familiar Kubernetes commands. The integration with Azure ensures that our application benefits from Azure's scalability, reliability, and global reach.

By combining these technologies, we create a scalable, manageable, and efficient deployment pipeline for our application. Docker ensures consistent environments, Kubernetes provides orchestration and scaling capabilities, and Azure offers a reliable hosting platform with additional resources and support. This holistic approach ensures that our application can handle varying loads, maintain high availability, and adapt to changing requirements, all while leveraging the best practices and tools available in modern cloud and container orchestration technologies.

## Bibliografie

(2023, September 7). Retrieved from Wikipedia:

[https://en.wikipedia.org/wiki/Broker\\_pattern#:~:text=The%20broker%20pattern%20is%20an,for%20transmitting%20results%20and%20exceptions.](https://en.wikipedia.org/wiki/Broker_pattern#:~:text=The%20broker%20pattern%20is%20an,for%20transmitting%20results%20and%20exceptions.)

Foundation, O. (2021). *OWASP Top Ten*. Retrieved from OWASP: <https://owasp.org/www-project-top-ten/>

Gayantha, K. (2020, Januari 2020). *Client-Server architectural pattern*. Retrieved from Medium: <https://medium.com/@96kavindugayantha/software-architectural-patterns-17b05bfa3aef>

Heusser, M. (2020, March 10). *What is a microkernel architecture, and is it right for you?* Retrieved from TechTarget: <https://www.techtarget.com/searchapparchitecture/tip/What-is-a-microkernel-architecture-and-is-it-right-for-you>

IBM. (n.d.). *What are microservices?* Retrieved from IBM: <https://www.ibm.com/topics/microservices>

Kaseb, K. (2022, March 27). *The Layered Architecture Pattern in Software Architecture*. Retrieved from Medium: <https://medium.com/kayvan-kaseb/the-layered-architecture-pattern-in-software-architecture-324922d381ad>

Netherlands Enterprise Agency, R. (n.d.). *10 steps to comply with the GDPR in the Netherlands*. Retrieved from Business.gov.nl: <https://business.gov.nl/running-your-business/business-management/administration/how-to-make-your-business-gdpr-compliant/>

Ritvik Gupta, A. S. (2023, October 15). *software-architecture-patterns-types*. Retrieved from Turing: <https://www.turing.com/blog/software-architecture-patterns-types/>