University of Bergen
Department of informatics

Exploring Capability-based security in software design with Rust

Author: Kenneth Fossen
Supervisor: Håkon Robbestad Gylterud

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2022

Abstract

Access control is one of the most critical aspects of software engineering when designing secure software. In 2021, the Open Web Application Security Project (OWASP) [15] released a new Top10 several years after its last release in 2017. “Broken Access Control” [11] made a significant jump to the top of the list, marking it as the most prone and vital security aspect of software development.

Previous research shows that security challenges, such as Confused Deputy [19], can be solved with a capability-based approach. To achieve a capability-based system for REepresentational State Transfer (RESTful) Application Programming Interfaces(APIs), we use the Rust programming language to explore how we can create a capability design pattern. We want to create a library for the developer to harness the power of capabilities when writing the code, adhering to the capability properties and Principles of Least Privilege (PoLP), and creating a RESTful API.

We created a capability library 5.8 we used to implement a RESTful API, simple-api 5.9, connecting it with Grant Negotiation and Authorization Protocol (GNAP) into a proof-of-concept capability-based system published on GitHub [10]. Resulting in successfully creating capability-based access control for RESTful APIs. We show a use-case [27] where the core access control model is Capabilities and potentially mitigates confused deputies in a RESTful API software architecture.

Acknowledgements

I wish to thank my supervisor, Håkon, for believing in this project and for letting me choose to explore it in my way. Your guidance and discussions have been precious and vital to me for this master's thesis.

I want to thank my family for allowing me to explore and support my ideas to find my path in life. Your support and always welcoming home have been a solid foundation for me to reach as far into new worlds as possible.

Last but not least, a special thanks to Boisy. The one that has always been a happy face next to me. He has been connecting me to new people for love and friendship, opened the world for me in a completely new and different way, and made this journey balanced and a true adventure in computer science and life.

Kenneth Fossen
01 June, 2022

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Problem statement and motivation

Access control is one of the most critical aspects of software engineering when designing secure software. It determines how a user can interact with and manipulate data described by the software. The most common access control models are Access Control Lists (ACL), Role-Based Access Control (RBAC), and Attribute Based Access Control (ABAC). Choosing a good access control model and implementation for the software is not easy. When Open Web Application Security Project (OWASP), in 2021, released their newest Top 10 [15] describing the latest web security trends, “Broken Access Control” [11] had become the topmost important security issue in software development.

In OWASP’s description of “Broken Access Control” they describe access control as the following; “Access control enforces policy such that users cannot act outside of their intended permissions” [11]. As software designs are now more complex and distributed, they are more prone to confused deputy problems. Norm Hardy wrote a paper named “Confused Deputy”[19] describing how a user interacting with another privileged process is tricked with valid input to overwrite sensitive information outside the user’s intended permissions, naming it the confused deputy problem. This problem occurs because two principals act together: (1) the user with limited access and (2) the process with access to the sensitive data. Hardy argues how access control models such as ACL are not capable of solving confused deputy problems and that we need Capability-based Access Control (CBAC) to solve this. In CBAC, capabilities are unforgeable tokens describing the authorization to access a resource for a principal. Capabilities may be transferred between principals. In Hardy's example, a *transferred* capability would grant access to the user's resources without compromising access to sensitive resources. Hence, it will allow access to the requested user resources without confusing the privileged process.

Modern software architecture has embraced the REpresentational State Transfer (RESTful) [9] Application Programming Interface (API) style and has since 2005 been increasing in popularity[26]. RESTful API style has many benefits. It is easy to understand due to its design around the well-known HTTP protocol. A RESTful API is based on HTTP protocol, processing requests and responds with the content the user requested in a structured way. One of the constraints in a RESTful API design is statelessness. Statelessness ensures that RESTful APIs are not designed to handle sessions state, e.g. authentication [21], making RESTful APIs prone to Cross-site request forgery (CSRF).

CSRF is the modern version of the confused deputy, where the user's web browser is targeted and tricked into sending forged requests to a trusted RESTful API. OWASP has a good description of this in a banking example [36] of how CSRF is executed. A CSRF attack can have severe consequences for the targeted user, e.g., deleting resources without the user knowing. One way of reducing the attack surface of CSRF is to reduce the user's permissions to the strictly-needed permissions for the intended user interaction. Restricting the user's permissions is also known as Principle of Least Privilege (PoLP). This principle applies not only to the user's access permissions but also to browsers and RESTful API running access permissions. RESTful APIs are recommended to run as a limited-service account in the hosting operating system, giving only access to open ports and access to necessary files needed to serve the user request.

Since the idea of capabilities arrived in 1975 [37], there has been a debate about their usability. Miller's paper "Capabilities myths demolished"[27], tries to answer the usability claims of capabilities, arguing against the common misconceptions about capabilities. PoLP and capabilities work in tandem. Capabilities provide fine-grained access control through unique and shareable tokens, while PoLP limit the user and services to only have access permissions to the intended resources in the software design.

When designing security for a single Service-Oriented Architecture (SOA) or Microservice Architecture (MSA), there are many different concerns. Access control is one of the important choices that software architects are faced with in any software design, and it dictates how a user interacts and accesses resources. Today's recommendations for access control in RESTful APIs are RBAC and ABAC, depending on which fits the needs best suited for the organization size and structure. RBAC will give coarse-grained access control to the system that is suitable for many systems. However, depending on the system design and its needs, this might lead to "Role explosion". Role explosion happens as the organization evolves and the intended role changes or a person hired for a position is acquiring more roles to perform their intended job. In ABAC, role-explosion is not a concern, as the attributes defined on the resource, user, the environment, or a combination of the listed is granting access to resources defined by a policy. Because of this, National Institute of Standards and Technology (NIST) has listed ABAC as the preferred model for access control. Either choice of these approaches, the choice is making many decisions for the developers and software architecture.

Authorization, "To grant a principal access to certain information"[37], is another important aspect to consider. Today the industry-standard is OAuth and is a authorization framework for RESTful API supporting both RBAC and ABAC. It is built on a series of Request For Comments (RFC), building on top of each other, making OAuth complex and hard to implement correctly. In this thesis, we will use Grant Negotiation Authorization Protocol (GNAP) instead of OAuth. GNAP is a transactional authorization protocol that tries to remove the complexity that OAuth has introduced; it also removes the need for browser that OAuth is dependant on and that the new version of OAuth v2.1 is also trying to fix. GNAP also supports RBAC, ABAC, and CBAC.

When it concerns security with RESTful APIs in MSA, Tetiana Yarygina at UiB wrote her Ph.D. thesis on the topic in 2018, "Exploring Microservice Security"[40] concluding that the field of security and microservices was very sparsely researched, even with the rise in popularity in the industry for this new architecture. Continuing in her paper "Overcoming Security Challenges in Microservice Architectures"[40], Yarygina looks at the importance of trust between microservices, introducing the MissFire framework to establish Mutual Transport Layer Security (mTLS) trust between microservices to limit the attack surface. Yarygina's MissFire is successfully limiting some of the scopes of confused deputy problems in MSA design but not eliminating it. MissFire mainly limits services to communicating together at the network level and keeps communication confidential. MissFire changes nothing regarding what privileges the service is running with or what it has access to once approved to talk to another host. Still, the microservice with a certificate has access to all RESTful endpoints on the targeted microservice, making it able to post data with the user context to this service.

Due to this, we are motivated in this thesis to explore how we can implement CBAC to mitigate confused deputies in RESTful APIs. We will use Rust as our programming language to explore how we can structure the design of the code to achieve capabilities for executing the corresponding code blocks. After exploring a few approaches, we will choose one approach and implement it as a library, such as in listing 1.1. We will use this library to build a RESTful API as part of a capability system. This library will aid the developer in creating capabilities (line 1) for data structures and connect them to a function block (line 17). Rust will, together with the library, give early feedback to the developer resulting in a more secure code and design in production. The capability system will be built on top of this library, helping us harness the transactional powers of GNAP and mitigate potentially confused deputies in the design for future RESTful APIs. We have published code for the full proof-of-concept on GitHub under the following repo: `https://github.com/spydx/capability-poc/`.

Listing 1.1: Intro: Library example: **intro.rs**

```
1  #[capabilities(Delete, id = "id")]
2  pub struct Orders {
3      id: i32,
4      name: String,
5  }
6
7  #[service(SqliteDb, name = "db")]
8  #[tokio::main]
9  async fn main() -> Result<(), std::io::Error> {
10     let connection_string = "sqlite::memory:".to_string();
11     let _pool = CapService::build(connection_string)
12         .await
13         .expect("Failed␣to␣create␣database");
14     Ok(())
15 }
16
17 #[capability(Delete, Orders)]
18 fn delete_order(order: Orders) -> Result<(), CapServiceError> {
19     let res = sqlx::query!(r#"DELETE␣FROM␣orders␣WHERE␣id␣=␣$1"#, order.id)
20         .execute(&self.db)
21         .await
22         .map_err(CapServiceError);
23
24     Ok(())
25 }
```

1.2 Related works

In 1975's paper "The protection of information in computer systems"[37], the capabilities idea arrived, and there have been several papers that are important for our work here.

Closely related work to our work is Mark Samuel Miller's dissertation "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control"[28]. Miller is exploring the E language with its capability design to build robust and unified access control in a desktop application and distributed system. Miller is also exploring Concurrency Control; this is out of the scope of our work. Besides that, Miller's work is very similar to our approach, but their design is in the desktop application.

The paper "Capability Myths Demolished"[27], arguing for many of the misconceptions that have arisen over time around the CBAC system. It explores these misconceptions through 3 Myths, Equivalence, Confinement, and Irrevocability, that we cover to understand what a capability-based system is for our work.

Ekaterina Shmeleva's thesis is a state-of-the-art master thesis "How Microservices are Changing the Security Landscape"[38] and the paper "Architecting with Microservices: A systematic mapping study"[7]. Both show the lack of research and usage of CBAC in modern software architectures like SOA and MSA.

Also, the freshest security books out of the press; "API Security"[23] and "Microservice Security In Action"[39], barely mentions CBAC and capability based security for their targeted readers, the modern and security-oriented software developer.

Tetiana Yargina's dissertation "Exploring Microservices Security"[40], where she implements the MissFire framework to mitigate trust issues in a microservice architecture. MissFire is targeting security at the network layer between services. Our work is naturally an extension of hers by adding capabilities to enable services to communicate together, mitigating the confused deputy even further.

1.3 Goals and research questions

The overall goal of this thesis is to explore and evaluate how we can implement CBAC with Rust in a RESTful API. We will look at the built-in language features, e.g., traits and macros, and how they can assist us in creating a capability design for our desired API. After exploring these approaches (experiment 1a–c), we will choose one approach (experiment 2a) and implement a library, letting the developer develop a capability-based RESTful API. We will then integrate the API with an authorization authority (experiment 2b), e.g., GNAP, and build a client to connect it for a proof-of-concept capability-based system.

We will attempt to answer the following questions to evaluate capability-based access control as a viable option for access control in this system. We will evaluate if it further enhances security in the architecture design and limits "confused deputy" problems in the architecture in question.

Research Questions:

- RQ1: Can we structure our code or utilize Rust's ecosystem to achieve CBAC within a RESTful API?
- RQ2: Can CBAC give fine-grained access control and avoid confused deputy problems in a RESTful API?
- RQ3: Could GNAP help us realize capability-based fine-grained access control in a RESTful API?

Evaluation criteria We then will evaluate experiments 1a–c against the following evaluation criteria, but not limited to:

– It should be clear to the developer when a capability operation, such as Read, is needed.
– The code design should not increase code complexity.
– The code design should be generalizable into a library.

For experiment 2, we will evaluate the solution against the following criteria, but not limited to:

- Adhering to the code design should be easy for the developer.
- The code design should clarify what capabilities are needed.
- Should reduce the likelihood of any confused deputy in the RESTful API.

1.4 Chapter outline

A short introduction to the content in each chapter of this thesis:

Chapter 2 - Rust & OWASP This chapter will introduce Rust's essential language features and why we do not use some. We will explain the importance of OWASP and its findings relevant to our thesis.

Chapter 3 - Access Control Models & Challenges Provides an overview of Access Control models, covering the most common access control models and capabilities. The next part of this chapter covers the confused deputy, principal of least privilege, and capability myths that we need to understand before evaluating our experiments.

Chapter 4 - Tokens & Authorization Authorities It introduces an overview of different types of tokens, where we will cover the essential properties of the different token types before we choose one to use in this thesis. Lastly, we will cover two authorization authorities, the well-known OAuth and the new GNAP, currently in draft.

Chapter 5 - Experiments and Results This chapter contains the design approaches we evaluated and their evaluation results against the evaluation criteria. We will go through the design choices and how we developed the library, and how our capability system, together with the library and GNAP, pan out.

Chapter 6 - Discussion, Conclusion and Future work Finally, we will discuss our results and review them in the context of previous research and on research goals. We will reflect and conclude before we mention future work.

Chapter 2

Rust & OWASP

In this chapter, we will start by getting an introduction to the essential features of Rust, continuing with an overview of the software security landscape from OWASP through the last decade, mentioning the challenges in securing web applications and OWASPs findings important for this thesis.

2.1 Rust

Rust is a young programming language that is also gaining popularity. It has been voted the most beloved programming language on the yearly Stack Overflow Developer Survey [32] polls.

Rust is known for its approach to memory safety with its unique ownership feature and avoids many issues that its counterparts C/C++ are challenged with when writing low-level code managing memory. Ownership is a set of three rules the Rust compiler checks at compile time and helps manage memory without a garbage collector. The first rule is (1) that each variable in Rust has an owner, (2) there can only be one owner, and (3) the variable is dropped when the owner goes out of scope. The borrow-checker validated these ownership rules when compiling, preventing memory issues and eliminating the need for a garbage collector. Rust is also a competitor to C/C++ regarding runtime speed with its high-performance multi-threading as Rust can interface C/C++ libraries with its Foreign

Function Interface (FFI). Other languages such as Java are also memory-safe and popular Object Oriented Programming language (OOP) choices for modern software development. Java uses a garbage collector to manage memory and requires a Java Virtual Machine (JVM) that acts as a runtime engine to execute. On the other hand, Rust does not require a similar runtime environment and compiles directly to the target platform. Also, Rust is not a OOP but has traits and trait bounds that support OOP features.

Rust's borrow-checker, strongly typed language, macros, and tooling makes it excellent for secure software development of APIs and as a systems language. Companies such as Amazon and Microsoft are actively advocating the use of Rust, and more companies are searching for Rust developers. Linux Kernel recently adopted Rust as an official language.

Rust has an active community that has provided tools such as Clippy `cargo clippy` and `cargo check`. These tools are excellent at providing early warning about potential problems and other errors in the code, allowing the developer to correct these before it becomes a more significant issue later in the development process. Rust's compiler, by default configured to fail a release build, `cargo build --release` if the compiler produces any warnings. These warnings are configurable, but Rust ships with a strict default policy. Warnings can come from various sources, e.g., unused imports, unused variables, and dead code. This nagging from the compiler about warnings and errors will either force the developer to better code hygiene or warning message fatigue. Nevertheless, overall it is a feature that helps the Rust community keep conforming to Rust's idiom and write better code. Popular IDE tools such as VSCode, CLion, and IntelliJ all have plugins actively maintained by the Rust community, striving to enhance the developer experience. Rust Weekly is a newsletter that informs the community on the latest developments regarding the language and its tooling.

2.1.1 Benefits from Rust

In this thesis project, we would like to utilize Rust's strongly typed language, traits, and macros to give the compiler the ability to let the developer know as early as possible that the developer is adhering to our capability pattern.

With traits, we will define the shared behavior for our design. Traits are similar to Java's interfaces, where we define the expected implementation signatures and then let the user implement them. Interfaces in Java or any other OOP support inheritance in the form of a hierarchy. Rust's traits do not support inheritance. Instead, Rust has implemented trait bounds to allow the developer to expect a trait to be implemented for a type. This short example 2.1 shows how traits are similar to interfaces. To pass any object of type `T` to he function `check_birthday_for` (line 13), we are required to implement the trait `BirthDay` for the type `T` (line 9). The main method and implementation details for `has_birthday` method are omitted.

Listing 2.1: Rust: Traits and Bounds Example: **bounds.rs**

```
1  trait BirthDay {
2      fn has_birthday(&self) -> bool;
3  }
4
5  struct Person {
6      birthday: DateTime<Utc>,
7  }
8
9  impl BirthDay for Person {
10     fn has_birthday(&self) -> bool { ... }
11 }
12
13 fn check_birthday_for<T: BirthDay>(person: T) -> bool {
14     person.has_birthday()
15 }
```

Furthermore, Rust has macros that we will use to simplify our Metaprogramming. Rust has two different forms of macros, `macro_rules!` are declarative macros `procedural macro` are function-like macros. `macro_rules!` is commonly used to express code better. An example of this `vec![]` macro takes several inputs but always creates a `Vec::new()` of the desired types. E.g. `vec![0, 1]` 1 will create a vector of length 2, with type `i32`, and numbers 1 and 2, removing our burden to create a new vector, and `push` two numbers onto the vector, as shown in the following example 2.2 (line 2 - 4).

Listing 2.2: Vectors: **vectors.rs**

```
1  let macro_list = vec![1,2];
2  let mut manual_list = Vec::new(); // mutable
3      manual_list.push(1);
4      manual_list.push(2);
5  assert_eq!(macro_list, manual_list); // true
```

Procedural macros we use to decorate `struct's` and `functions` in Rust. The `procedural macro` then converts the e.g `struct` into a `TokenStream` that we can manipulate. A `TokenStream` is the code that the developer has written, parsed into a fully typed and Abstract Syntax Tree (AST) that we can work with programmatically. A small example of how we use our procedural macro `#[capabilities]` 2.3 to specify the behavior for our code and view the generated code adhering to our specified behavior 2.4. We will explain this code in detail in experiment 1c 5.7.

Listing 2.3: Rust: Procedural macro: **macro.rs**

```rust
#[capabilities(Delete)]
pub struct Person {
    personnumber: i64,
    firstname: String,
    lastname: String,
}
```

Listing 2.4: Rust: Partial procedural macro output: **macro_output.rs**

```rust
pub struct Person {
    personnumber: i64,
    firstname: String,
    lastname: String,
}

pub trait CapDeletePerson:
    Capability<Delete<Person>, Data = Person, Error = CapServiceError>
{
}
impl CapDeletePerson for CapService {}
```

Macros benefit us because it removes the boilerplate code that our design needs. We will use a combination of the `procedural macro` and `macro_rules!` in our library to achieve the desired behavior. Both macros work well together to help us remove and make the design ad-hear to our design goals without the developer having to do much.

During the development of the library, we depended on Rust's tooling to expand the code to catch any potential problems. Rust tools `cargo expand` and `cargo watch` made this process manageable. Expanding and continuously updating the output as we were developing, making it easier to catch failed macro expansions or compile errors as we were developing the library as shown in the following code example 2.5 and 2.6.

Listing 2.5: Rust: Example input: **svc.rs**

```rust
use capabilities::SqliteDb;
use capabilities_derive::service;

#[service(SqliteDb, name = "megakult")]
#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let connection_string = "sqlite::memory:".to_string();
    let _pool = CapService::build(connection_string)
        .await
        .expect("Failed␣to␣create␣database");
    Ok(())
}
```

Listing 2.6: Rust: Example output: **cargo watch -q -c -x ”expand –test svc”**

```rust
use capabilities::SqliteDb;
use capabilities_derive::service;
use sqlx::Pool;
use async_trait::async_trait;
pub struct CapService {
    megakult: SqliteDb,
}
pub struct CapServiceError;
impl CapService {
    pub async fn build(conf: String) -> Result<Self, crate::CapServiceError> {
        let con = Pool::connect(&conf)
            .await
            .expect("Failed␣to␣connect␣database");
        Ok(Self { megakult: con })
    }
}
/* shortened for readbility */
pub trait Capability<Operation> {
    type Data;
    type Error;
    #[must_use]
    #[allow(clippy::type_complexity, clippy::type_repetition_in_bounds)]
    fn perform<’life0, ’async_trait>(&’life0 self,__arg1: Operation,) -> Output =
    ↪ Result<Self::Data, Self::Error>
    >
    where
        ’life0: ’async_trait,
        Self: ’async_trait;
}
#[allow(dead_code)]
fn main() -> Result<(), std::io::Error> {
    let body = async {
        let connection_string = "sqlite::memory:".to_string();
        let _pool = CapService::build(connection_string)
            .await
            .expect("Failed␣to␣create␣database");
        Ok(())
    };
/* removed tokio build */
}
```

2.1.2 Traits in Rust

Rust uses traits to implement reusable behavior for our data types and thus supports polymorphism. There are two types of traits one is trait objects and the other is trait bounds.

Trait objects are more general traits that treat the type more as an object; if it has a `foo`, it must be a `bar`. Also known as *duck typing* in dynamic languages, this may introduce some performance impact because we are using dynamic dispatching, and the Rust compiler does not know all our types before run time.

Trait bounds we use with generics, enabling the developer to implement required behavior for our specified type `T`. In our code example 2.1, our developer has to implement the trait `BirthDay` for our type `T`. If our developer does not implement this trait for our type, the compiler will return an error until we have implemented this trait. When the compiler finds the trait, we can pass any `T + BirthDay` to our method `check_birthday_for`. We will be using trait bounds to implement capabilities in our library.

Rust is not a OOP in the same sense as Java. In Rust, objects can hold data through its data types `struct` and `enum`, and this complies with some definitions of OOP. Using the keyword `impl`, a programmer will add an implementation code block for a named trait. Using the keyword `pub` when defining `fn` in our trait code block, the programmer will decide what should be the public methods for the trait we implement for our `structs` or `enum's`.

There is no support for inheritance in Rust. `struct` or `enum` can not inherit any behavior from any parent data type how a `class` in Java can inherit from one or more `interfaces`, `abstract class`, or `classes`.

2.2 OWASP - Software Security Recommendations

OWASP is a non-profit organization that produces recommendations for software security. They have an extensive list of best practices and how to mitigate several problems. It also maintains a list of the most common vulnerabilities in software, named OWASP Top10 [15]. Since 2003 it has been regularly updated. However, in later years there have been some gaps in the releases of their Top 10. There was a 4-year gap between the two latest releases, 2017 and 2021. In the 2021 Top10, three issues stand out from the crowd and are interesting for this thesis.

Our main concern is the 2017 Top10 “A05:2017 - Broken Access Control” that rose from a fifth place to a solid number one issue “A01:2021 - Broken Access Control”[11] in application security. OWASP is in it’s description of this category mentioning “Violating the principle of least privilege”, “Cross-Site Request Forgery”, “CORS misconfiguration allows API access from unauthorized/untrusted origins”, and “Accessing API with missing access controls for POST, PUT and DELETE” as primary problematic causes for these problems to occur in today’s software. Further, from their data analysis, 94% of the applications showed that they had a form of broken access control, and avoiding the “confused deputy” is not easy. Careful planning and an attentive developer are needed to be able to mitigate “confused deputy” problems such as CSRF. Cross-Origin Resource Sharing (CORS) policies will help the browser limit the simplest forms of CSRF, but relying only on CORS will not mitigate all CSRF problems.

Also, today’s RESTful API design is not inherently ad hearing to the PoLP described in 3.2.2. Commonly APIs are running as a system service user or restricted system service user in the Operating System (OS). However, the developer is usually free to request any stored objects from the database or filesystem that the user-context has access to when the API is running in the OS. Not restricting the service user privileges has been the source of many problems, especially SQL-Injection (SQLi) attacks or reading secrets from configuration files stored together with the attacked API. Design patterns such as Command Query Responsibility Segregation (CQRS) split the model into two separate responsibilities, displaying data and updating data. The display model of the data is limited to a read-only database connection, and updating the data has access to update the same data in the database. They are separating the concerns and limits SQLi and moving the application towards PoLP.

In 2017, the second biggest problem was “A02:2017 - Broken Authentication”. This category has now been renamed and dropped on 2021’s Top 10 to a seventh-place, “A07:2021 - Identification and Authentication Failures”[14]. This category contains problems such as exposing session identifiers in the URL or reusing session identifiers. Reusing or storing session identifiers in cookies makes it easy to hijack sessions. The topic is also covering storing passwords with outdated cryptographic libraries or just allowing the use of simple to guess passwords such as `qwerty` or any password listed on the 10.000 Most Common Password list [1].

Insecure Design [12] is a security concern in today’s software development trends. Insecure design is a new category for 2021 and targets the control design of the architecture. This category targets issues concerning the lifecycle of the development process and how threat modeling the application can mitigate weaknesses in the architecture design.

A concern is also “A06:2017 - Security Misconfiguration”, this topic has taken a small step up to fifth-place “2021:A5 - Security Misconfiguration”[13]. Showing that this still is a major challenge for developers and administrators when shipping new software. Issues covered in this topic include improperly configured permissions and unnecessary privileges for the services. The misconfiguration itself is not something that the developer alone can change, but writing software that ad hears to PoLP 3.2.2 would reduce the impact of this problem for both the system administrator and developer.

Reading through the OWASP Top10, we can see that the terrain has changed significantly. The new security landscape puts more responsibility on the individual developer to attain new knowledge and adapt and comply with newer security standards to develop secure code. In 2017, “A01:2017 - Injections” was the biggest problem that has dropped by 2 places to “2021:A03 - Injection”. Some frameworks have adapted to such a challenge, such as Entity Framework Core in .Net to mitigate SQL Injections in .NET Projects. Similar frameworks exist for Java, Rust, Go, and other languages that handle database interactions. Injection vulnerabilities are not just limited to SQLi, but SQLi is a good reference point to show how the community has evolved to face and mitigate the challenges the developers are facing when developing software. As we are attempting in this thesis, the community has actively worked for better frameworks, making the development process safer and resulting in more secure software.

Chapter 3

Access Control Models & Challenges

3.1 Access Control

This chapter will briefly cover the relevant forms of access control models and then cover the topics around the PoLP and CBAC. It starts with the basics of plain ACL, before we move on to RBAC and ABAC. Lastly, we will cover PoLP and an in-depth view of CBAC. It will give us a historical view of how access control has evolved. Access control dictates how we intend the user to be able to interact with data in our software, and it also dictates how a user can interact with data in our software. Proper access control is crucial when it comes to how we design secure software.

3.1.1 Access Control Lists (ACL)

ACL is commonly used in file systems when controlling access to resources, restricting access based on the identity of the access request. ACL are lists containing entries of user identities and allowed actions on a resource. An example of this is the UNIX filesystem 3.1 where determining access based on the identity of the user with the permissions `r` (read), `w` (write), or `x` (execute) are stored in the filesystem and honored by the OS.

Listing 3.1: ACL: Unix filesystem example

```
1 > ls -la
2 total 0
3 drwxr-xr-x    5 kenneth  staff   160 Apr  5 10:48 .
4 drwxr-xr-x  153 kenneth  staff  4896 Apr  5 10:48 ..
5 -r-xr-xr-x    1 kenneth  staff     0 Apr  5 10:48 read_execute_file.txt
6 -r--r--r--    1 kenneth  staff     0 Apr  5 10:48 read_file.txt
7 --w-------    1 kenneth  staff     0 Apr  5 10:48 write_file.txt
8 -rw---xr--    1 kenneth  staff     0 Apr  5 10:48 own_readwrite_everyone_read.txt
9 >
```

The access control listing `-r-xr-xr-x` is divided into four parts; the type, the owner, a group, and everyone; `{type}{owner}{group}{everyone}`. The type is one letter, describing if the object is a director `d` or a file `-`. The first three positions list the permissions for the object owner, the three successive positions are for the group, and the three last positions describe everyone's permissions.

In the above example 3.1, the user "kenneth" has several files stored in a directory. In line 8; `-rw---xr--`, our user "kenneth" has been granted `read` and `write` on the file named `own_readwrite_everyone_read.txt`. The group "staff" is granted `execute` on the same file, and everyone can `read` our file.

It works well when one principal acts on the resources, not when two principals act together, such as in Hardy's example where the service (compiler) is the confused deputy. ACL has no way of expressing the need for a service to access the other principal's files and also store sensitive data in a different location. If the service principal is granted access through a group, it will be granted access to all files in the filesystem where the group is attributed access. It is a violation of PoLP and still makes it prone to overwriting files for different principals than the two principals acting together.

3.1.2 Role-based Access Control (RBAC)

RBAC has been the most popular choice to manage many users over a large organization. It defines relations for users to roles to mimic the organizational structure, such that it can be applied to resources for access control. We define roles through their intended purpose in the organization and grant them privileges to the required resources needed to accomplish the specified role. We assign user identities to one or more roles, enabling the user to perform their tasks in the organization. An example of this is the predefined roles within the Windows

OS such as the `Users`- and `Administrators`-role. `Users` is giving access to login and run pre-installed software within the OS, and `Administrators` access to administer all aspects of the OS.

Its flexible model is one of its advantages for managing and structuring access controls in an organization. RBAC together with a directory service, e.g., Azure Active Directory, gives a central identity manager to administer roles and users. The challenges start when the organization evolves and when roles change, and it may lead to more roles than intended where we get a phenomenon that we call "role explosion"[16]. Role explosion does not make RBAC an insufficient alternative for access control, but in some organizational hierarchies, other access control models may be a more suitable option.

3.1.3 Attribute-based Access Control (ABAC)

ABAC is the next iteration of access control models, trying to solve the role explosion problem that RBAC has. ABAC differs from RBAC in that we do not only rely on roles for granting access. We can evaluate any attribute on the subject, on the resource, type of action, and attribute in the environment. Administrators have to use these attributes to write access control policies stored at the policy decision point (PDP), to grant access to resources.

When an access request arrives at the resource, a policy enforcement point (PEP) inspects and generates a request to a PDP to evaluate and result in access granted or denied. The third part of this model is the policy information point (PIP), which will provide information about external attributes if requested from the PDP.

With its complex architecture, ABAC may not be suitable for smaller organizations, but it is a good alternative for larger enterprises experiencing role explosion.

3.1.4 Capabilities - CBAC and Object Capabilities

Saltzer is the first to mention capability in his paper “The protection of information in computer systems” [37]. The definition Saltzer uses of a capability is “In a computer system, an unforgeable ticket, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket”. In a modern web application, a token is a type of ticket that Saltzer describes. The token can be any unique information stored in, e.g., a Cookie 4.1.1, Opaque Token 4.1.5, or JWT 4.1.2, to mention a few types, representing the right to access the desired resource.

The token held by a principal has several properties in a capability system; it can be revoked, shared with other principals (delegation) 3.1, and avoids confused deputy problems 3.2.1. Revocation in the capability context means that we can request a token invalidated to an authority. The token holder can no longer use this token to access the described resource in the token. Delegation is the property of sharing a capability with different principals, e.g., user or service, to lend them access to the required resource. Capabilities enable us to avoid confused deputy by leveraging PoLP, further discussed in the sections “Confused Deputy” 3.2.1 and “PoLP” 3.2.2.

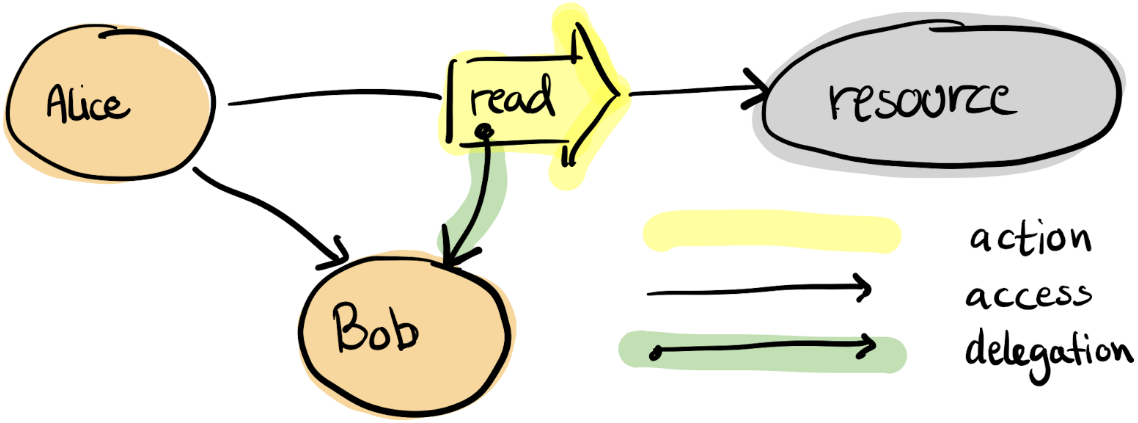


Figure 3.1: CBAC: Principal Alice delegating ”read” to principal Bob for a resource

There are two main types of capability systems one is object-capabilities and the other is CBAC. Some programming languages have object capabilities, and examples of such languages are Mark Miller’s `E` used in his dissertation [28], and the later Pony that has arrived as a newcomer to the field. Jessica Hillert wrote her Bachelor’s thesis on the comparison of Capability Systems in the following languages; Encore, Pony, and Rust [20], where Rust fared rather poorly with object-capabilities. There have been recent discussions in the Rust community [24] [25] to implement new keywords in Rust to support a capability system in the language. Our implemented library is inspired by the works of Zack Mullaly [30], giving us a simple foundation for object-capability in the language through the use of Macros in Rust. The other capability system, CBAC, is a central authority that will contain a file descriptor table that stores tokens listing their granted access and token owners. The principal holding a token is the owner, and this token will, when trying to access a resource, be passed to the resource and evaluated for granting access.

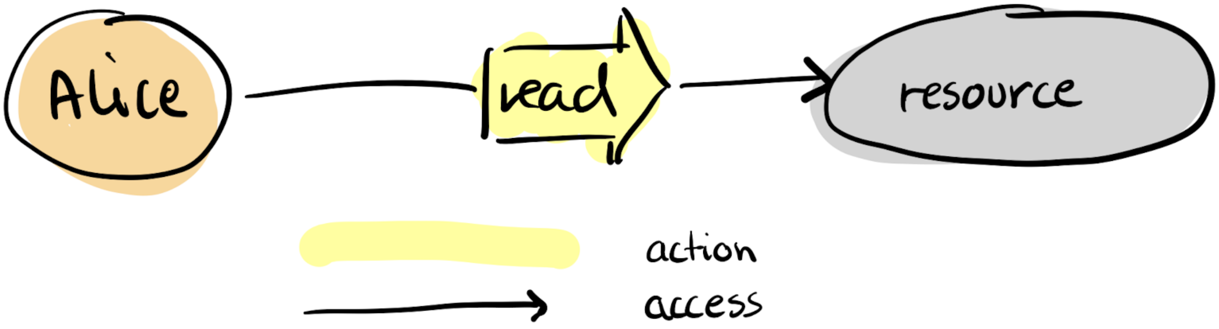


Figure 3.2: CBAC: Principal Alice accessing with ”read” on a resource

There are, unfortunately, some myths that have arisen over time regarding capability-based systems. These myths will be covered in the section “Myths around capabilities” 3.2.3, and they are primarily based on misunderstandings regarding capabilities.

3.2 Access Control Model Challenges

3.2.1 Confused Deputy

When two principals act together, depending on the access control model, the principal acting on behalf of another principal might be subject to being a confused deputy. Hardy exemplifies this in his paper [19] with regards to ACL in a filesystem. Using Hardy's example, there are two principals. We have a principal component we will call the compiler and a principal component we will call the user. When the user submits its request to the compiler, the user specifies the input file for the compiler to compile and the output destination for the compiled file. This compiler also tracks how much time a user consumes for billing purposes and stores this in a sensitive data area. The compiler must have access to read the user's input file, write at the user-selected destination, and write to the sensitive data area for billing. A well-intended user (Alice) 3.3 submitting the parameters `app.rs` and `app.out` will not confuse the compiler. An ill-intended user (Eve) 3.4 submitting the parameters `app.rs` and `/billing/eve.csv` to the compiler will result in the compiler overwriting the sensitive billing information for Eve, resulting in the compiler being a confused deputy.

In our example, the user does not have access to sensitive billing information. However, since the compiler does, it allows the user to specify writing to the sensitive data area. Since the compiler allows this, it has to distinguish between legal and illegal parameters deciding where to read from and where to write, making it prone to being a confused deputy. Having such logic inside the compiler can become a challenging maintenance task. Each system is different, and there might be edge cases that the logic will not catch. Hence it is better to look to other options, such as capabilities 3.1.4 and the principle of least privilege 3.2.2, to solve the confusion for the deputy.

In a capability-based system, when the user is submitting their input and output files. Under the hood, they also delegate a unique token, representing their access to their input and output files, to the compiler service. The compiler does not need access to the user files and runs with fewer privileges. When the user delegates a token, this token will grant the necessary access for the compiler to read and write at the user-specified locations.

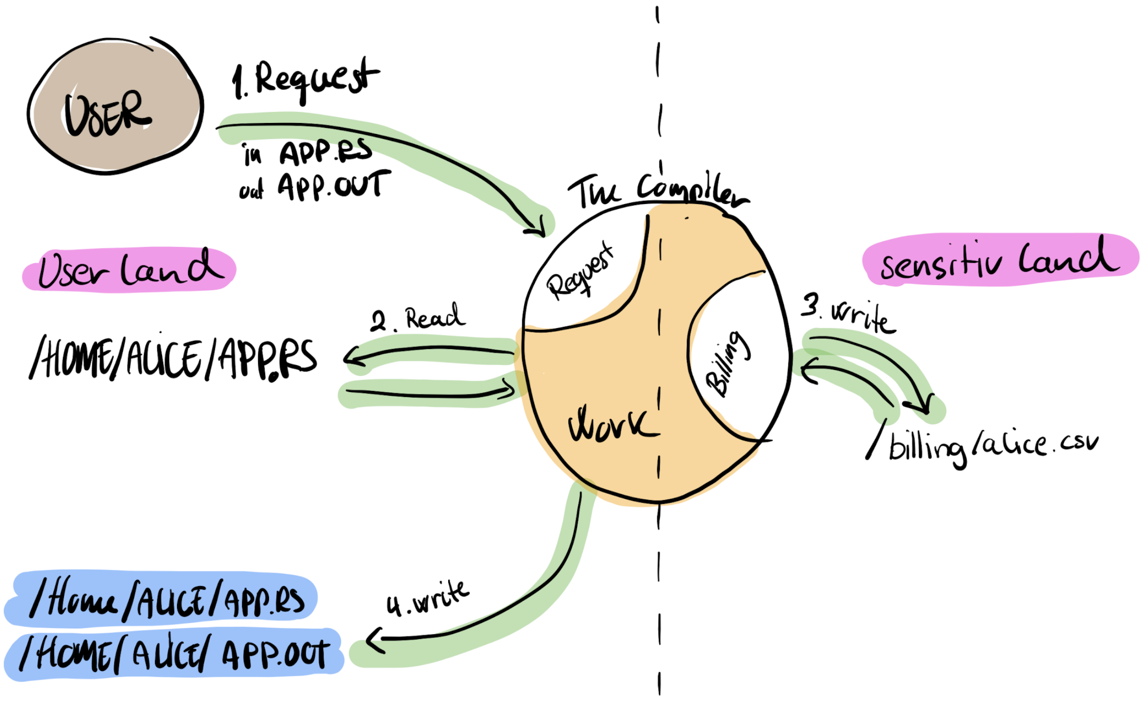


Figure 3.3: Confused Deputy: Well-intended Alice

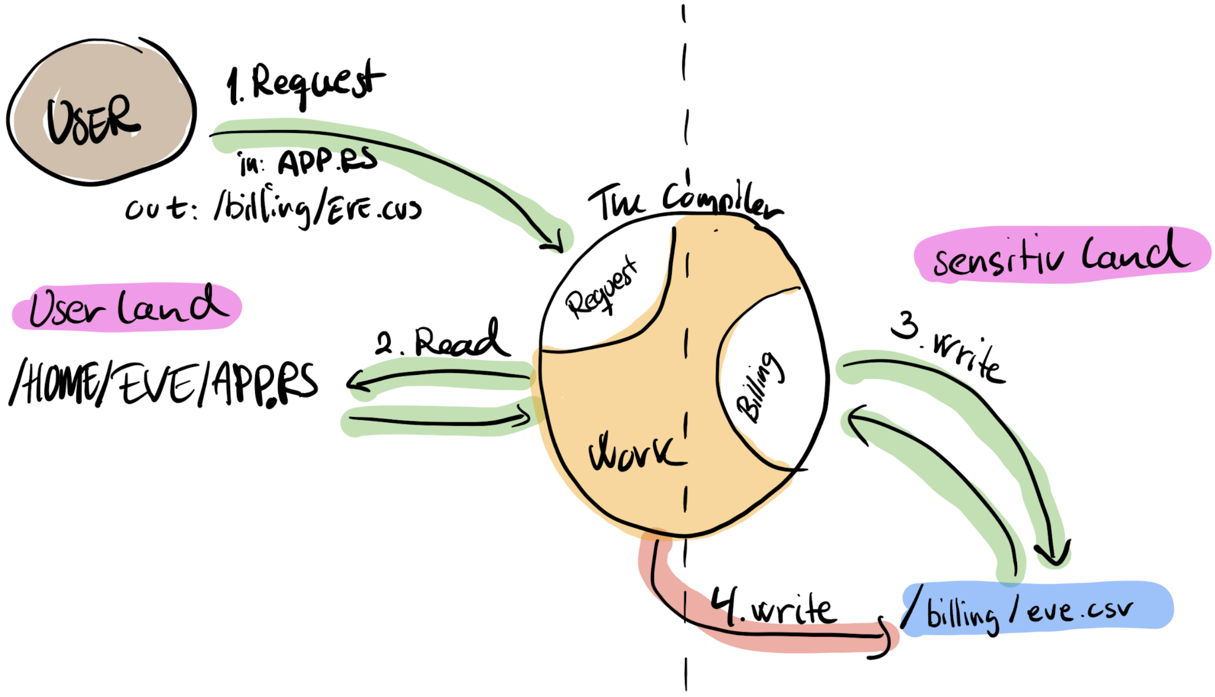
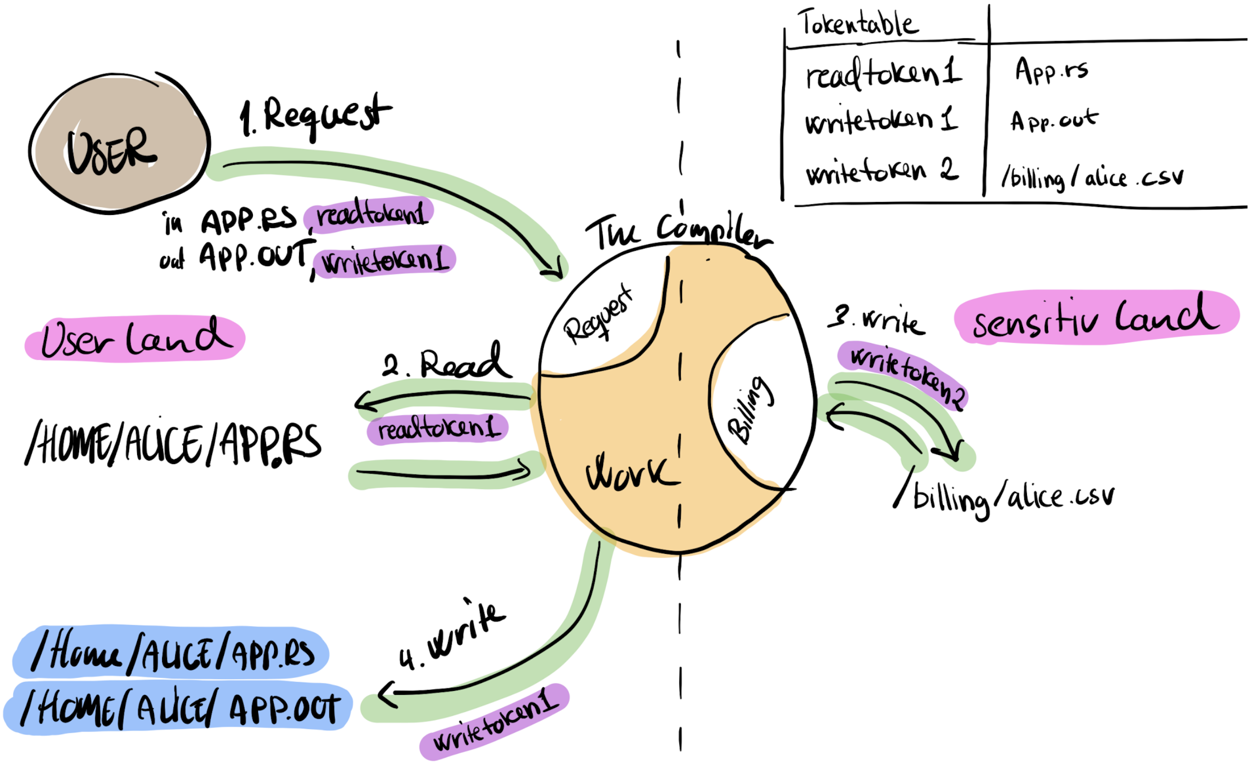


Figure 3.4: Confused Deputy: Ill-intended Eve

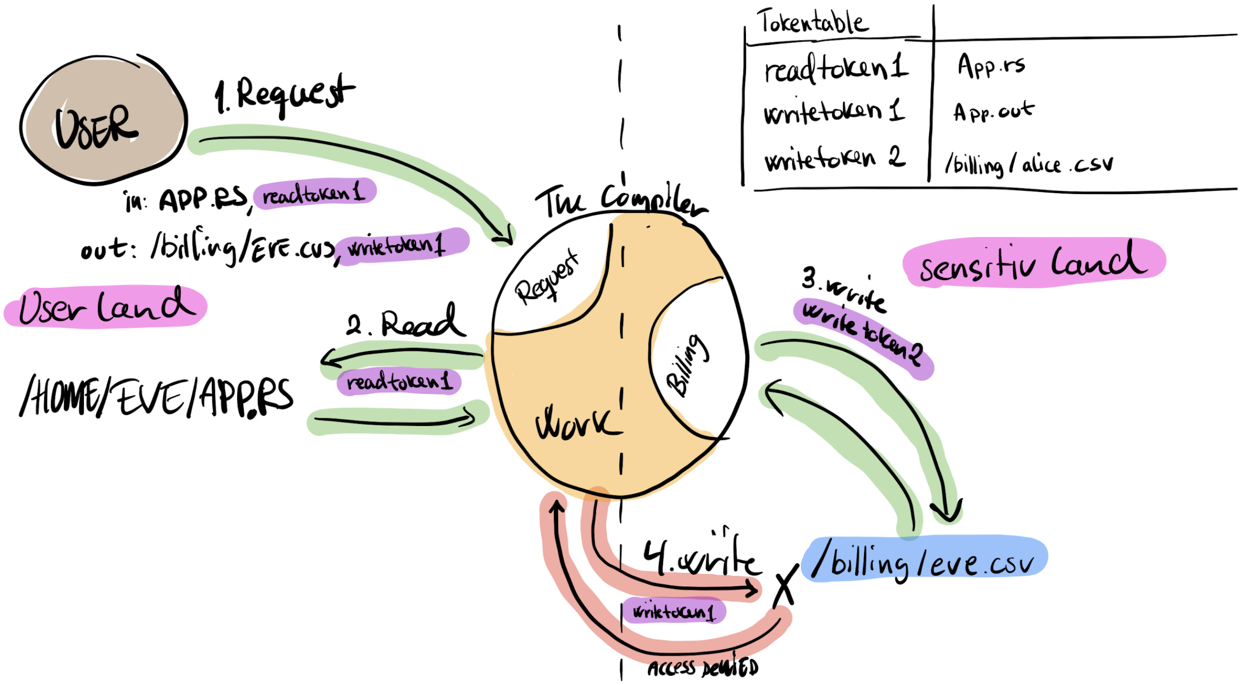
The following figures show the previous example with capabilities. First is our user Alice 3.5 that has two tokens for each of her files and passes these to the compiler allowing it to write to the filesystem for the specified files. Second is our user Eve 3.6 trying to confuse the compiler with an invalid token for the file Eve has specified for her output file, resulting in an access denied by the filesystem. The compiler also has a private token granting write access to the sensitive data area.

Hardy's example is for services and a filesystem residing inside an operating system, but compared to software design today, one will find a similar pattern in modern web application design. A web application runs in the browser and usually interacts with a RESTful API. These two components can be candidates for confused deputy, the browser through well-known attach vectors such as XSS and CSRF, and the RESTful API in further interacting with other RESTful APIs, e.g., microservice architecture and services, such as in Hardy's example.



Tokentable	
readtoken1	App.rs
writetoken 1	App.out
writetoken 2	/billing/alice.csv

Figure 3.5: Confused Deputy: Well-intended Alice with Capabilities



Tokentable	
readtoken1	App.rs
writetoken 1	App.out
writetoken 2	/billing/alice.csv

Figure 3.6: Confused Deputy: Ill-intended Eve with Capabilities

3.2.2 PoLP - Principle of Least Privilege

Principle of Least Privilege, also known as Principle of Least Authority, is the concept of only granting the needed privilege to a principal to let it achieve its designated job. The definition of a privilege is an allowed action on a resource object given to a principal, e.g., opening a network port for listening or reading a file from the filesystem. From our confused deputy problem 3.2.1, we can see that giving the compiler access to all files in the filesystem will be granting this principal too much access. Even if we change to the RBAC model, we will be prone to the same challenges since the roles must overlap, not limiting the compiler's access and granting access to too much in the filesystem.

Sandboxing is also a different way of achieving lesser privileges, but this approach does not strip the principal from its privileges, but instead, it limits the reach of the principal to be inside a sandbox. In our confused deputy example 3.2.1, sandboxing will not limit Eve 3.4 to overwrite the sensitive billing information since the compiler principal still has access to this file. Sandboxing is, therefore, not the same as PoLP.

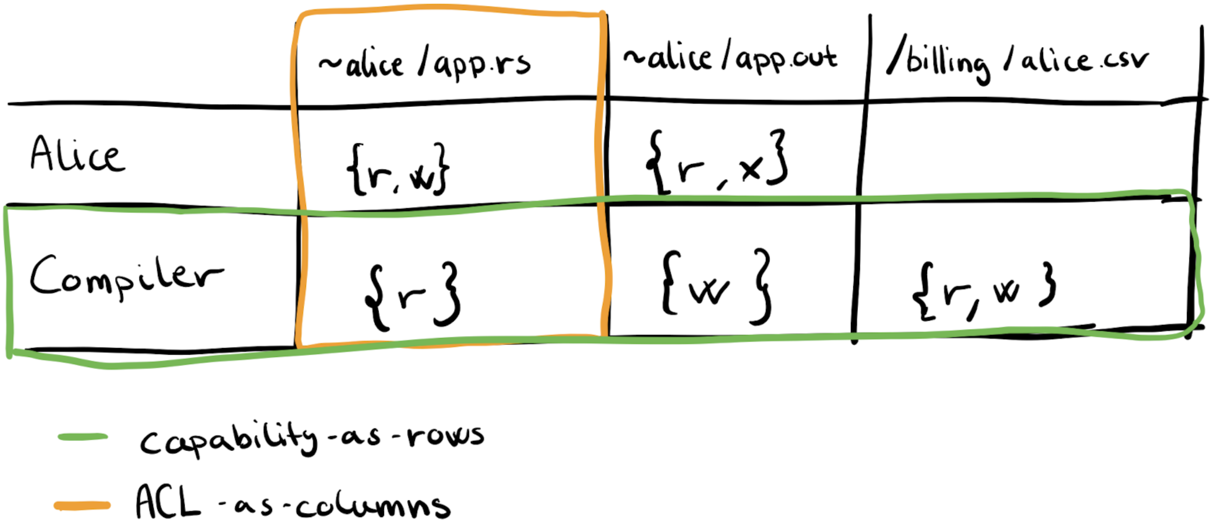
Capabilities can help achieve a very fine-grained PoLP. With capabilities, we can strip the principal for the compiler only to request one capability, to write billing information to the sensitive data area. The compiler now only has access to the sensitive data area and nothing else on the system. Whenever a user wants to request compile-time, they have to share their capabilities for the files they have access to, letting the compiler use these capabilities to access the necessary files to fulfill the user request, such as in Alice's well-intended example with capabilities 3.5.

PoLP is an essential part of securing systems and how we design our software. As Software Engineers, we write code in a context where we later apply restrictions such as security unless we already are using frameworks helping us. Rust is such an example, as the language promises memory safety through the borrow-checker but still does not restrict us from creating any software we want.

3.2.3 Myths around capabilities

Since the first mention of capabilities, there have been discussions and myths surrounding the understanding of capabilities. Miller's Capability Myths Demolished [27] lists three myths that we will briefly describe here. (1) the equivalence myth, (2) the confinement myth, and (3) the irrevocability myth.

The *equivalence myth* is the belief that ACLs are the same access matrix and that ACLs are columns and capabilities are rows in this matrix 3.7. As Miller points out, this matrix does not do capabilities justice, as the model the matrix represents is not flexible enough. Miller further visualizes this example by showing the same matrix as object references instead of a matrix 3.8. The critical part shown in this visualization is that the arrows in the ACL model point to the subjects (user) and that the visualized capabilities point to the resource subject from the holding subject (user). The visualization shows how inherently different the two models are and that comparing ACLs-as-columns and capabilities-as-rows is not correct.



	~alice/app.rs	~alice/app.out	/billing/alice.csv
Alice	{r, w}	{r, x}	
Compiler	{r}	{w}	{r, w}

Figure 3.7: Capabilities: ACLs as columns and Capabilities as rows

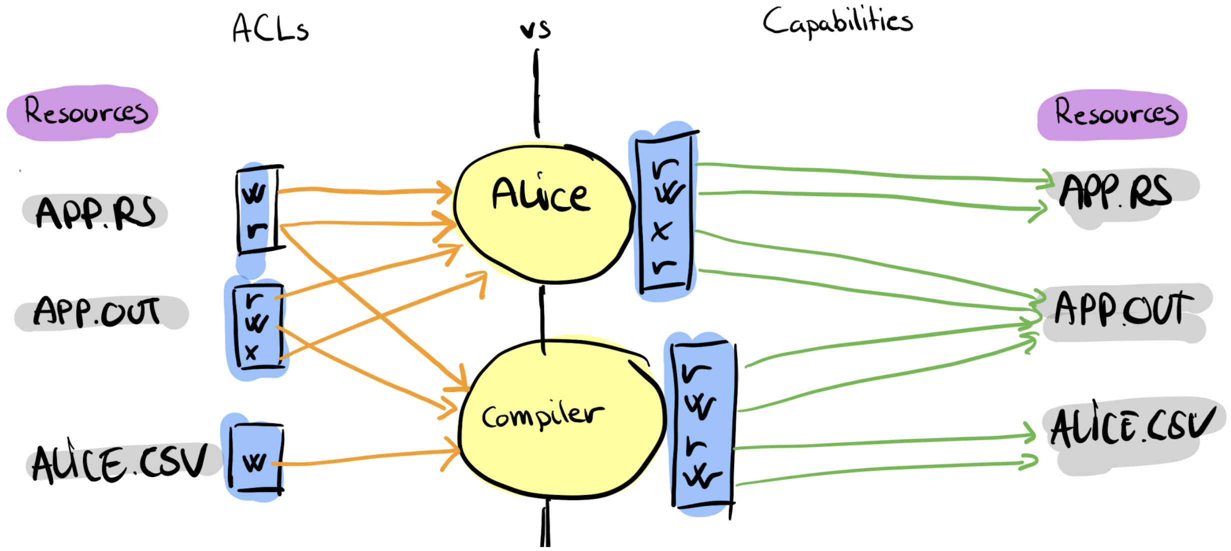


Figure 3.8: Capabilities: ACLs as columns and Capabilities as rows with object references

The *confinement myth* is about with whom and how to share capabilities between subjects in a capability-based system. The misconception is that Alice can share a capability with Bob, and then Bob can share this capability further without restrictions, e.g., with Eve. In this example, our subject Alice does not have to be authorized with both subjects Bob and Eve to share the "read" with Bob for Eve 3.9 (Unconfined). Trust is a critical part of capabilities, and to be able to share a capability with another subject, the sharing subject Alice, in this example, has to have an authorized path for both subjects to share a capability, such as in the figure 3.9 (Confined). Miller lists KeyOS as a project that has achieved confinement.

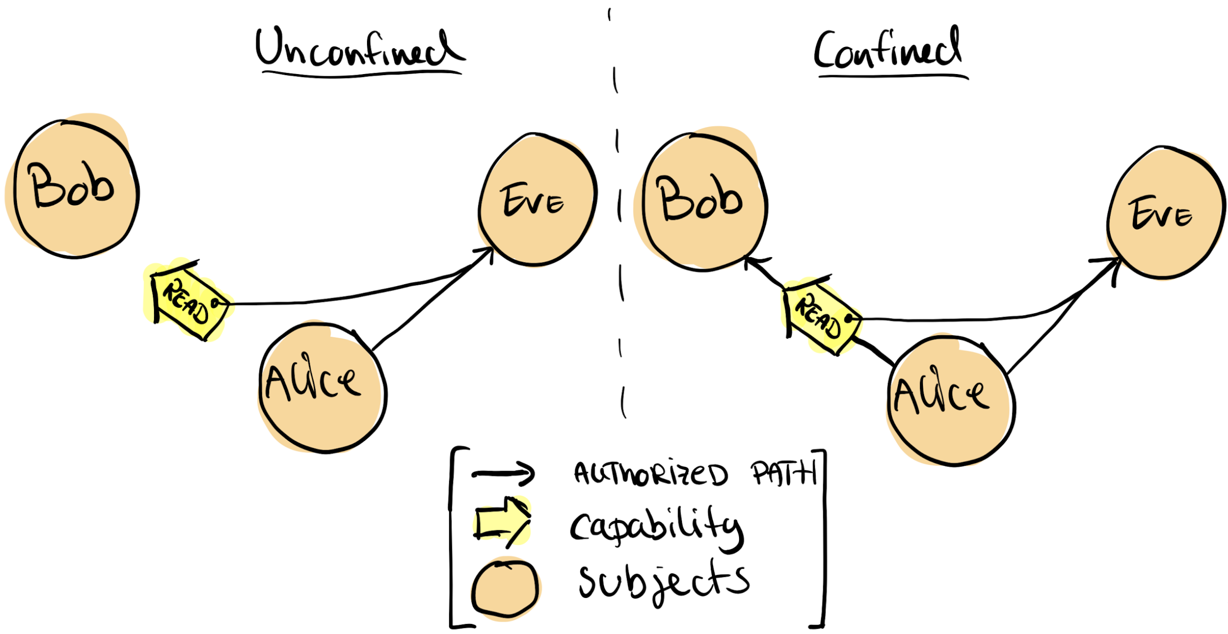


Figure 3.9: Capabilities: Sharing

The *irrevocability myth* is linked to the confinement myth and argues that one cannot revoke access in a capability-based system when sharing a capability with a subject. A capability is a reference with access privileges to a resource subject, and since the capabilities are with the capability holder, one cannot revoke this capability from the holder. Miller's solutions to revocation in a capability-based system are that we can compose new capabilities. One capability we call revokable, and a capability we call sharable, referencing the revokable capability. We share the shareable capability with Bob, letting Bob access the resource through the revokable. Once we want to revoke Bob's access, we revoke the revokable capability breaking the composition and restricting Bob's access, such as in figure 3.10.

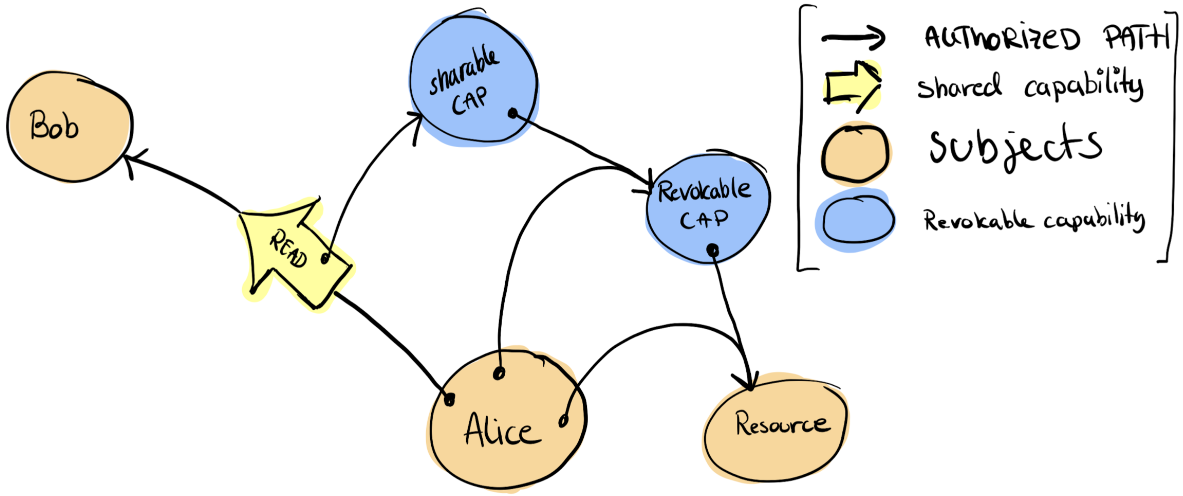


Figure 3.10: Capabilities: Miller's revokation factory [27, Figure 6]

A widespread misconception Miller describes is *capabilities-as-keys* and a very intuitive way of describing capabilities, although not entirely correct. An example of capabilities as keys is a user's story of collecting documents from a safe deposit box. The user arrives at the counter, asking the clerk (deputy) to collect our fictive box 123 and handing the clerk a copy of their key for this box, allowing the clerk to unlock and retrieve the box and hand it to the user to collect the requested documents. This example gives the main idea of capabilities but fails to show the other properties of capabilities and has led to widespread belief in the *irrevocability-* and *confinement-myth*.

Miller states that this misconception, *capabilities-as-keys*, fails to honor the Composability of Authorities- and Access-Controlled Delegation Channels-property. Composability of Authorities (Property E) [27, Figure 13] is that we view users and resources as subjects in a capability-based system, making access and authorization unified to create a network of authority relations, and such as the Confined part in figure 3.9. Access-Controlled Delegation Channels (Property F) [27, Figure 13] require treating all entities as subjects and that composability of subjects is a restriction of delegation of keys. E.g., if two subjects are not composed in a network, the delegation of a key is not feasible between the subjects.

Miller argues well against these myths and shows several cases of how others, e.g., KeyOS and confinement [27, p.5], have resolved these myths. Still, the myths are persistent, and there are misunderstandings regarding capabilities, especially the *capabilities-as-keys* metaphor.

Chapter 4

Tokens & Authorization Authorities

4.1 History of token security in APIs

This chapter will introduce the most widely used tokens used for session management in RESTful APIs. We will introduce HTTP Cookies, explain their use-case, and describe JSON Web Tokens (JWT) and Opaque Tokens. Further, we will mention the new kids on the block, Macaroons and Biscuits. We will also cover the delegation and authorization authorities OAuth and GNAP. GNAP is currently in the review process at IETF. Both support the most widely used tokens for session handling and authorization.

4.1.1 HTTP Cookie

In the early 1990s, the HTTP Cookie was created by Lou Montulli [2]. Montulli worked at Netscape Communications, the company responsible for the popular Netscape Navigator browser. There was a demand to manage sessions in the stateless protocol HTTP, and the browsers needed aid in managing the sessions. The first version of Cookie appeared in 1994, and the following year got patented in 1995, and soon after, all major browsers supported Cookies.

A Cookie consists of a string name, string value, and optional attributes stored in the browser and sent with HTTP requests to the targeted API, line 3 in following example 4.1. Cookies are small and comes in several types, that all have different properties and use cases.

Listing 4.1: Cookie: HTTP Request with Cookie

```
    GET /cookie_demo.html HTTP/2.0
    Host: www.kefo.no
    Cookie: session_id=supersecretid; Secure; HttpOnly; SameSite=Strict
```

The Session Cookie stores a session-id used to identify the user and session at the API. When users close their tab or browser, the browser deletes all Session Cookies. The Persistent Cookie stores long-term information for the user; this can be session ids or other preferences for the user when visiting specific web pages. Setting a Persistent Cookie expiration time and the browser will invalidate the Cookie.

Since these cookies persist, they have also been used to track user movement when visiting other web pages, such as integrated Facebook modules in specific web pages allowing visitors to share articles they read on Facebook. The Facebook module will read the user's cookies for Facebook details and submit information about the web page the visitor is visiting and other details to Facebook. Persisting sensitive session information and tokens for the user makes the user vulnerable to tracking and stolen sensitive information; since the persisted information is still accessible after the browser has reopened.

Over the years, new attributes, also known as flags, have been introduced for the Cookie standard to mitigate the attack surface for leaking sensitive information stored in the browser. A `Secure` flag was introduced to instruct the browser only to send this Cookie with an HTTP request if the request connection was over HTTPS. If the request connection is not secured, the browser will not add this Cookie to the HTTP request.

An `HttpOnly` flag has been added to limit cross-site scripting (XSS) and cross-site request forgery (CSRF). The HttpOnly flag tells the browser to restrict access to Cookies accessible through simple JavaScript running in the browser. XXS and CRSF attacks dependent on JavaScript will reduce the attack surface but not eliminate it just by setting the HttpOnly flag.

Later `SameSite` was introduced to mitigate CSRF attacks further. SameSite is also client-side protection, and it mitigates CSRF by restricting where the browser is sending cookies. It has three levels, `Strict`, `Lax` , and `None`. These tell the browser how strict it should be when evaluating the policy and if it should send cookies to third-party web pages. Cookies are beneficial but can be easily misused to id and track users across web pages. Cookies are still widely used to store important and functional user information, e.g., user personalizing and user sessions for their web experience.

4.1.2 JSON Web tokens - JWTs

JSON Web Tokens (JWT) was published in 2010 and has become the industry standard. IETF has drafted and documented JWT in the IETF OAuth Request for comments RFC7519 [22]. JWT's purpose is to transfer claims between two parties safely. It achieves this by encoding the header and payload with base64url encoding. The encoded header and payload, together with a dot symbol (.), are then used to create a cryptographic signature for the token, appending the signature to the end of the token 4.2.

Listing 4.2: JWT: Signature structure

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),
    "secret_key")
```

The result is a three-part string that we call JWT 4.3 used to submit as a token with the HTTP packet. The first part (line 1) of JWT is the header describing the algorithm used to sign the JWT. The second part (line 2) is the payload we would like to share, and this section is readable by anybody intercepting this token in transit. The signature is appended at the end (line 3). A common misconception is that JWTs are encrypted, and one cannot read the payload of the JWT. They are signed so no parties can tamper with the payload on an issued JWT, but everyone can extract the header and payload. Storing sensitive information in the payload is, therefore, not recommended. The following example 4.4 is the expanded version of our example token in 4.3.

Listing 4.3: JWT: Resulting Token

```
1  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9                // header
2  .eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Iktlbm5ldGgifQ // payload
3  .F9qNfWupLEACGLUysYgtcDbYNRQspzcgtpXBbaaUgas         // signature
```

Listing 4.4: JWT: Expanded Header + Payload

```
{ // header
  "alg": "HS256",
  "typ": "JWT"
},
{ // payload
  "sub": "1234567890",
  "name": "Kenneth"
}
```

JWTs are used with an authorization authority, such as OAuth 4.1.6 or GNAP 4.1.7, to transfer claims from the authority to the client to grant access to a resource server. A client adds the JWT to an HTTP Header called Authorization 4.5 and prefixes it with Bearer . It is named Bearer since the JWT is self-contained and carries claims for the resource service to evaluate.

Listing 4.5: JWT: HTTP Authorization Header with JWT

```
Authorization: Bearer eyJhbG..J9.eyJb..m5ldGgifQ.F9qN..baaUgas //shortened version
```

The resource server first validates the signature for the transferred Bearer token and then inspects the claims before granting access. There have been several attacks with Bearer tokens, where attackers have exploited the JWT signatures verification process in specific libraries [31]. This process has been called “Signature Stripping”, and the error occurs due to incorrect implementation of the signature verification process in the libraries and not because of the JWT standard.

Since Bearer tokens such as JWT are self-contained, stealing a Bearer token through XSS or CSRF can have severe consequences for the user. The resource server cannot distinguish from whom the token is arriving, granting access to the attacker on the resource server.

4.1.3 Macaroons

Researchers at Google created Macaroons in 2014 [4], and they designed Macaroons with the web and distributed systems in mind. Macaroons are pretty similar to JWTs as they solve the same problem of transferring claims between two parties, and both are signed so that no parties can tamper with the claims in transfer. Macaroon’s claims are different from JWTs claims because Macaroons present their claims as caveats. We can have one or multiple caveats inserted into a Macaroon, and these caveats are rules to be evaluated granting specific permissions. A caveat can specify, e.g., access to upload a picture to a specific destination and use authorization logic to grant access to resources. “Macaroons are credentials, not capabilities, because the possession of a macaroon is a necessary, but not a sufficient condition for granting authority.”[4] the authors write in their paper. We rely on tokens being capabilities for this thesis, but we could design and enhance with Macaroons in our implemented library and authorization authority.

4.1.4 Biscuits

Developed at Clever Cloud and announced in 2021 by Geoffroy Couprie [6], Biscuit is trying to combine the best of all worlds. Decentralized validation from JWTs 4.1.2, flexible rights management using an authorization logic language such as Macaroons 4.1.3, small size such as Cookies 4.1.1, and can be treated as Opaque Tokens 4.1.5 by an authorization authority. Biscuits are therefore called an "authentication and authorization token"[6]. It managed access to Clever Cloud's systems, and Macaroons did not fit their use case. Biscuits have much to offer and fit very well with our use thesis case since Biscuits are capabilities-based. Unfortunately, it lacks some implementations; it is ready for serverside Rust but not currently ready for a simple Javascript web client.

4.1.5 Opaque Tokens

Opaque tokens differ from the previously mentioned tokens. They do not carry any information between parties (JWT), they do not have any caveats (Macaroons), and they do not carry any security policy enforcement (Biscuits). Opaque tokens are only random strings that have meaning to the authorization authority. There are several ways an opaque token can be meaningful for the authorization authority, e.g., it can be a primary key in a token table listing the token's properties, such as a file descriptor table. The downside to this; each request with an opaque token has to validate the token at an authorization authority endpoint. In some designs, having to validate each token at the authorization authority will create a bottleneck and limit the application's responsiveness. For our proof of concept, we have used opaque tokens in our GNAP 4.1.7 implementation. We chose this approach since we do not need to transfer any claim with the token (JWT), and our token table is supposed to work in the same sense as a file descriptor table.

4.1.6 OAuth

OAuth has become the leading authorization and access delegation standard at version 2.0 and is now considered the industry standard. OAuth is defined as a framework in RFC6749 [18] for authorization and authentication.

The rise of mobile apps in 2010 has reshaped the security considerations required to interact with an authorization framework. The design of OAuth relies on the user being able to be redirected, and the assumption from OAuth is that the user is using a web browser. It is problematic for mobile apps, and there was no native support for it initially. OAuth has since been extended with different extensions to support the needs and demands in the market.

OAuth is designed to support four grant types: Authorization, Implicit Flow, Password and Client Credential, and Authorization Code flow. Implicit flow is the most used grant flow in OAuth. Since the initial RFC5849 [17] specification of the OAuth in 2010, the security landscape has changed a lot. At that time, the Implicit flow was the recommended way of interacting with JavaScript apps but now is considered insecure by default [5].

Since the specification has evolved to adapt to the security landscape, the Authorization Code flow is now the recommended flow. The Authorization Code flow has also been extended to mitigate leakage and other security concerns when using this flow. Proof Key for Code Exchange (PKCE) RFC7636 was added to Authorization Code to prevent Cross-Site Response Forgery (CSRF). Following that, RFC8252 extended RFC7636, PKCE for mobile. These extensions are just a small example of how the specification has evolved and how challenging it may be to work with the OAuth specification.

Aaron Parecki has a blog post [33] about how confusing OAuth 2.0 can be due to the number of extensions in OAuth 2.0, arguing for a new version of OAuth, version 2.1. Parecki is now sitting in a working group trying to address this challenge with OAuth 2.0, cleaning it up for OAuth 2.1. Parecki is also working with Justin Richer for the Grant Negotiation and Authorization Protocol (GNAP).

4.1.7 GNAP - Grant Negotiation and Authorization Protocol

GNAP is a new transactional protocol for delegating authorization. Justin Richer is still drafting this protocol [35] together with Parecki and Fabien Imbault. This protocol's primary goal is to learn from the evolution of the OAuth framework to create a better-suited protocol for authorization.

OAuth's challenges come from the fact that they rely on a web browser and cannot solve without breaking changes in their standard. GNAP is designed not to depend on a web browser for its authorization flows but to support a flow for a web-browser redirect. GNAP supports four flows, Redirect-based interaction, User-code Interaction, Async Authorization, and Software-only Authorization.

GNAPs transactional approach is a strength of GNAP, and through GNAP's Request and Response messages, it is easy to request a token for a set of attributes, a token for a capability, or request multiple tokens at once. The documentation for the GNAP message format is also well documented in an interactive webpage to help the developer understand the usage of the different types of messages. The current designed version of GNAP supports out-of-the-box RBAC, ABAC, and CBAC. Unfortunately, not all aspects of capability-based access control are drafted yet for the protocol, and examples of such are delegation and confinement. Through discussions with Richer, the protocol's design can support confinement and delegation, but it is not the focus of the working group at the current time.

Another aim of the GNAP protocol is that the specification should be so precise that the job from specification to implementation should not be too much of a challenge for the developer.

We will be implementing the Redirect-based interacting flow in GNAP for our capability-based proof of concept.

Chapter 5

Experiments and results

This chapter will describe the approaches we have taken and the results from trying to introduce a capability-based code design into Rust in experiment 1 and for a capability-based system in experiment 2. We will use the evaluation criteria listed in section 1.3. We will begin by defining the evaluation criteria, experiments, tools and code, the capability-based system, and the environment we used for our setup before getting into the experiments and results.

5.1 Evaluation criteria

We will go through three approaches (experiments 1a–c) to introduce a capability-based system for our developer in the Rust programming language and evaluate against the following evaluation criteria:

– It should be clear to the developer when a capability operation, such as Read, is needed.
– The code design should not increase code complexity.
– The code design should be generalizable into a library.

For experiment 2, we choose the experiment from 1a–c that meets most evaluation criteria. We will (2a) create a library, 2b and implement a capability-based system for a RESTful API and then evaluate the solution against the following criteria:

- Adhering to the code design should be easy for the developer.
- The code design should clarify what capabilities are needed.
- Should reduce the likelihood of any confused deputy in the RESTful API.

5.2 Defining our experiments

We have looked into three approaches to creating a pattern with Rust to give us a capability design. We studied native language features and evaluated them against the evaluation criteria. The definition of our experiments is as follows:

1a. Generic Type Parameters and Trait Bounds 5.5 We try to model a simple capability model with generic type parameters and trait bounds in a manner that can be generalized and reused easily for the developer.

1b. Capability field inspired from TypeState 5.6 Using inspiration from the TypeState library to create a capability field capable of holding the action a capability should be able to perform in the code.

1c. Using macros in Rust 5.7 We will look at macros in Rust and see how this can generate code for us, and look at previous work where Zack Mullay is discussing in a blog post about the capability design of code in Rust [30] and build on this knowledge to create a CapabilityApi.

2a. Designing and creating a library 5.8 After evaluating these approaches (1a–c), we picked the most suitable from the results of the evaluating criteria. We proceeded to design and create a library to use in the capability system we will build in experiment 2b.

2b. Capability-based system We built a complete capability-based system containing a RESTful API using our capability library, an implemented version of GNAP, and a web client. We evaluate the solution against the second criteria 5.1.

5.3 Tools and Code

5.3.1 Tools Setup

For development, we installed Rust stable and added the nightly version of Rust. Rust comes with great tooling, and we have added some tools to the toolkit to enhance our development experience. We mainly used the build-in `cargo` command to build and manage the project. We also installed cargo tools such as `sqlx` for SQL support and `cargo-watch` and `cargo-expand` to view generated code from macros. Since we are building a library with macros, we have also become dependent on Rust `nightly` features in the language installed, and `rust-toolchain` has this specified where `nightly` is required.

Further, we have used Docker to containerize all the system's components and used docker-compose to make the process easier for evaluating the proof-of-concept. Building the client, we have used NextJS, which requires npm installed. We have also containerized the client application with Docker.

We used VSCode 1.66 as IDE and Rust-Analyzer extension to get language features into VSCode. A MacBook Pro 2019 with a 2,4 GHz Quad-Core Intel Core i5 processor and 16Gb memory running macOS Monterey v12.3.1 was used during the development process.

5.3.2 Code repository

We have published code for the full proof-of-concept on GitHub under the following repo: `https://github.com/spydx/capability-poc/`. This repo consists of submodules, requiring this clone command when cloning: `git clone --recurse-submodules <repo>`.

There are several essential parts of this repository. In the listing 5.1 the critical directories from the root have been labeled. Everything in this repository is built manually with `cargo` and `npm` or through `docker-compose` from the root.

Listing 5.1: Code: Repository layout

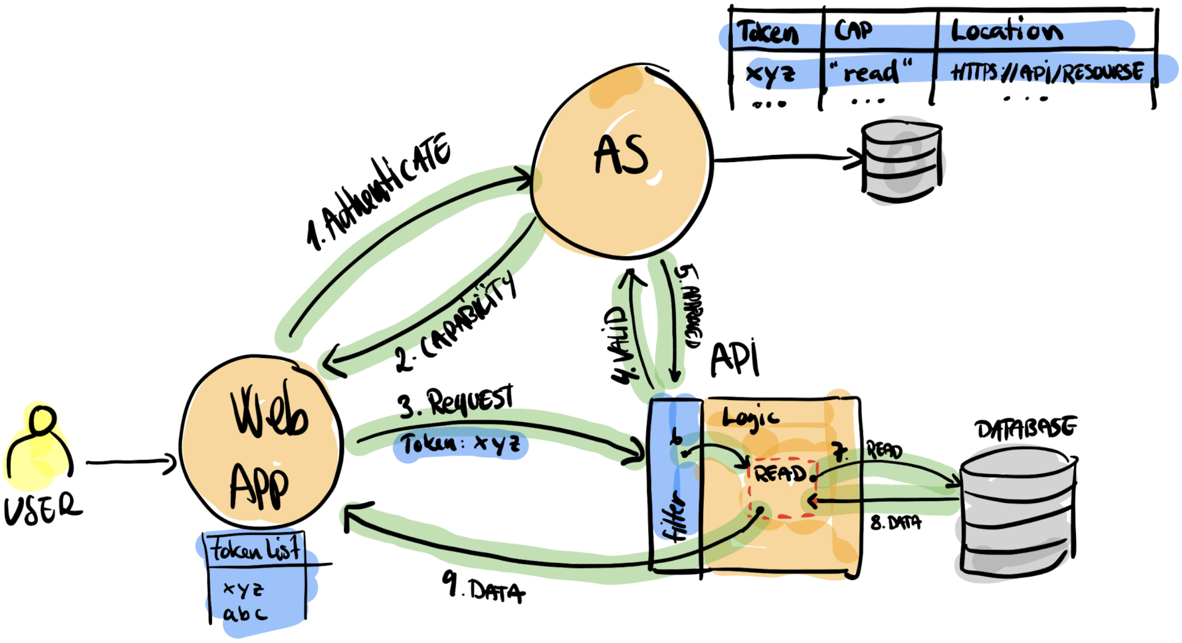
```
| - README.md
| - cap-client               // web client
| \ capabilities             // capability library
  | - capabilities-derive   // derive macros in the library
  | - src                   // additional code for the library
| - capabilityapi            // experiment 1c
| - docker-compose.yml       // system setup
| - dockerfiles              // buildfiles for docker
| - gnap                     // GNAP protocol and AS
| - gnap-cli                 // simple test client for GNAP
| - simple-api               // experiment 2 built with capabilities lib
| - trait-bounds             // code examples for trait-bounds, and typestate
```

5.4 Capability-based system

The experiments 1a–c described in 5.2 focues on the code design, 2a on Library and API Logic shown in section 5.4.2 and experiment 2b decribed in 5.2 focues on the overall picture described in section 5.4.1.

5.4.1 System overview

The desired capability-based system we would like to achieve is a system consisting of four components. The web app is responsible for holding the user's capabilities received from the authority server (AS) as tokens and using these tokens when interacting with the RESTful API. The AS stores and maintains the capabilities and tokens in a database. The RESTful API validates these tokens to a capability in a module called filter and passes the capability to the logic of the API. The logic handles the database connection and interacts with the data in the database. The interaction between all the components is in 9 steps, as shown in the following simplified figure 5.1.



Token	CAP	Location
xyz	"read"	HTTPS://API/RESOURSE
...	...	...

Figure 5.1: System: Sketch of a capability-based system

5.4.2 Filter and API Logic

The structure of the RESTful API will consist of two parts, the filter, and the logic, where the filter receives an HTTP request with a capability token, validates this token and retrieves the capability it represents and passes this to the logic. The logic then executes the desired code function that the capability authorizes access to and responds to the user request in a RESTful API manner, such as shown in the seven steps in figure 5.2

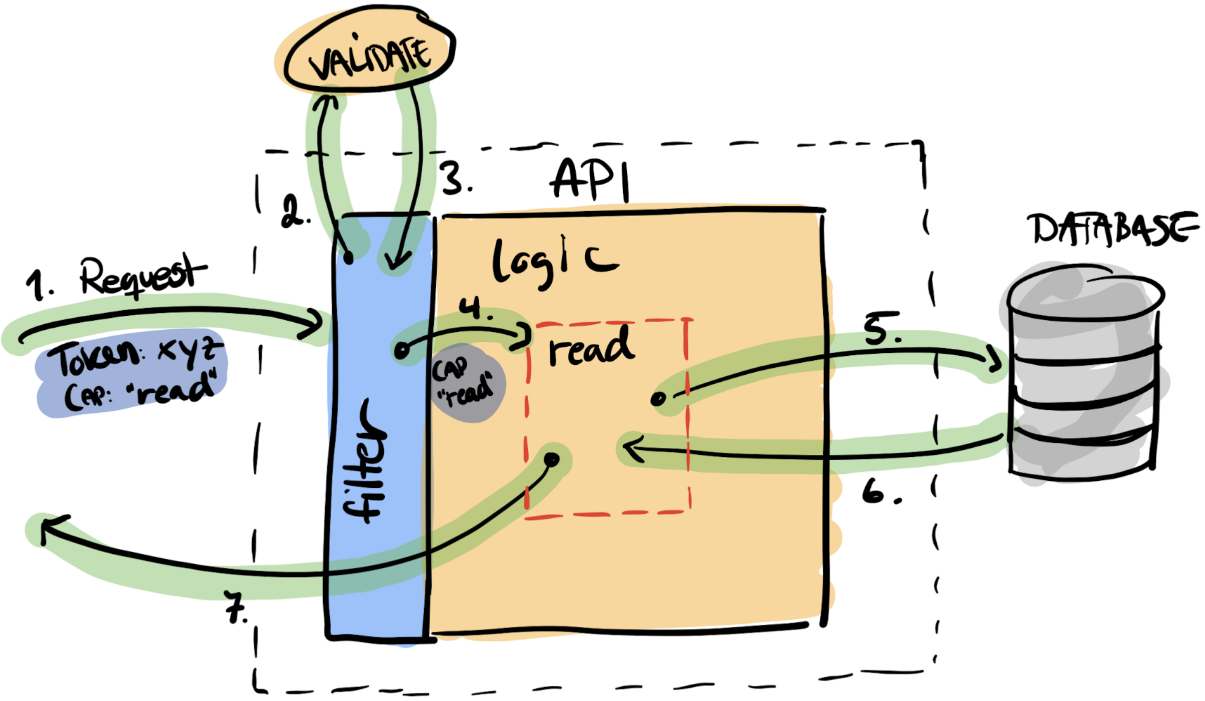


Figure 5.2: System: Sketch of the API layer

5.5 1a: Generic Type Parameters and Trait Bounds

In the approach, we tried to use the generic type parameters and trait bounds to create a design that will give the developer defining traits to implement a specific capability. We are bounding the trait to the type, enabling the developer to use functions, such as read, on a specific type when the developer has implemented the trait. Similar to the previously mentioned example, traits and bounds in section 2.1.

We started with the idea of a data structure, e.g., Person 5.2, that we should be able to create, store in the database, and read out from the database. We also created a trait that will represent our capabilities, create and read, for the struct 5.3. The trait `CreateRead<S>` contains our two capabilities, create line 2 and read line 3, and our developer has to implement this trait to be able to use the functions from this trait on our struct Person.

Listing 5.2: Traits: **struct**

```
struct Person {
    id: i64,
    firstname: String,
    lastname: String,
}
```

Listing 5.3: Traits: **capability read and create trait**

```
1 trait CreateRead<S> {
2     fn create(firstname: String, lastname: String) -> S;
3     fn read(id: i64) -> S;
4 }
```

Further, we wanted to store this information in the database and hence have a service struct maintaining the database connection 5.4. Also, for this service, we specify a trait requiring the type `T` to have `CreateRead<T>` implemented to accept only structs of type `T` with this trait implemented, such as line 2 in the example 5.5.

Listing 5.4: Traits: **service struct**

```
struct Service {
    con: Pool<Sqlite>,
}
```

Listing 5.5: Traits: **service trait**

```
1 #[async_trait]
2 trait DBCreateRead<T: CreateRead<T>> {
3     async fn read_db(&self, id: i64) -> T;
4     async fn create_db(&self, data: T) -> T;
5 }
```

Implementing these mentioned traits for the struct Person and Service, we can, from our main 5.6, create a Person struct on line 9 and create the Service on line 8. We can now call on the functions on line 10 to store the Person and on line 11 to read it from the database. For a complete implementation listing of this example, see Appendix A.1

Listing 5.6: Traits: **main**

```
1  async fn main() {
2      let con_str = "sqlite:bounds_persons.db";
3      let db: Pool<Sqlite> = SqlitePoolOptions::new()
4          .connect(con_str)
5          .await
6          .expect("Failed␣to␣create␣database");
7
8      let _service = Service { con: db };
9      let _p1 = Person::create("Kenneth".to_string(), "Fossen".to_string());
10     let create_res = Service::create_db(&_service, _p1).await;
11     let read_res = Service::read_db(&_service, create_res.id).await;
12     assert_eq!(create_res, read_res);
13 }
```

5.5.1 1a: Results

This approach works as it gives us a sense of capabilities needed to create and read from the database. It gives the developer a pattern to work out from and shows the developer's intention when reading the code. This code runs and is working as intended, and extending this example to be a RESTful API would give us a working example such as the intended system overview figure 5.1. We would get a token with capabilities over HTTPS to validate and run the correct functions, but we would lose the token information when running our functions. The advantage of this approach is that we use straightforward language features and do not have to implement libraries to follow this design.

There are, however, many other downsides to this approach. Our developers do not have to act according to the design, making it prone to circumvent our intentions. An example 5.7 is that since our struct Person is globally implementing CreateRead, creating the struct from its language initializer is not prohibited on line 1. Further, the developer can call `create_db` on line 6 with the newly created struct without limitation since the struct Person already implements the required trait, making the developer able to act outside its bounds.

Listing 5.7: Traits: **init struct**

```
let _p2 = Person {
    id: rand::random(),
    firstname: "Kenneth".to_string(),
    lastname: "fossen".to_string(),
};
let _p2_createdb = Service::create_db(&_service, _p2).await;
```

Also, when expanding to several different capability traits in this design, we would get code duplication. An example of this is wanting to create the capability traits CreateRead and CreateDelete. Because Rust requires us to implement functions create and read for the CreateRead trait, and the functions create and delete for the CreateDelete trait. Since Rust requires us to implement two create functions, we will get duplicate implementations leading to developer confusion and more code to maintain.

This approach may lead to more confusion since the design does not restrict the developer from following the design accordingly and the presence of code duplicates. Also, this design has no token information when executing functions, e.g., a read capability, not eliminating confused deputy in our code when acting with other principals such as the database.

5.6 1b: Capability field inspired from TypeState

This work is inspired from the work of José Duante and Antonió Ravara's "Retrofitting TypeState into Rust" [8]. TypeState enables structs to change behavior after running a function that changes the internal state of the struct. Their example is a Lightbulb, which has two states, on and off. When the Lightbulb is on, we can only turn it off, and vice versa. Extending TypeState to a capability, we could use the state of the struct to specify the capabilities it carries, e.g., it only has the read capability, and only the read function is available on the struct.

To get this code to work, we need to define a module line 1 for the struct to live inside. We would create the struct with the data fields we need and add a field describing the capability for the struct 6 inside this module. We are typing the struct with a capability type named Caps and are storing the capability in a field named cap in the example 5.8.

Listing 5.8: TypeState: **module and struct**

```
1  mod person {
2      use async_trait::async_trait;
3      use sqlx::{Pool, Sqlite};
4  
5      #[derive(Debug, PartialEq)]
6      pub struct Person<Cap: Caps> {
7          pub id: i64,
8          pub firstname: String,
9          pub lastname: String,
10         pub cap: Cap,
11     }
12     ...
13 }
```

Further, we use a private module inside the data structures module to create private traits inside to handle the state changes for the data structure and manipulate the capability we need 5.9.

Listing 5.9: TypeState: **private module and traits**

```
mod __private {
    pub trait Caps {}
}

pub trait Caps: __private::Caps {}
impl<__T: ?::core::marker::Sized> Caps for __T where __T: __private::Caps {}
#[async_trait]
impl __private::Caps for CreateRead {}
```

Then we can add the traits that we would want the developer to implement for the struct inside the module containing our data structure. For this example 5.10, we have added the trait `CreateReadCap` and a struct `CreateRead`. `CreateReadCap` is defined to contain two functions, create and read from the database.

Listing 5.10: TypeState: **Capability CreateRead trait**

```rust
#[derive(Debug, PartialEq)]
pub struct CreateRead;
#[async_trait]
pub trait CreateReadCap {
    async fn create(
        db: &Pool<Sqlite>,
        firstname: String,
        lastname: String,
    ) -> Person<CreateRead>;
    async fn read(db: &Pool<Sqlite>, id: i64) -> Person<CreateRead>;
}
```

After the developer has implemented the necessary traits for the Person struct on line 6 in listing 5.8. The developer can in the main function (listing 5.11) call to create a Person with the capability `CreateRead` on line 8, making available the create function on line 9 and read function on line 13 on the data structure. Implementation of the traits has been omitted and is part of the appendix B.1 as a complete implementation listing.

Listing 5.11: TypeState Example: **main**

```rust
1  async fn main() {
2      let connection_string = "sqlite:persons.db";
3      let database = SqlitePool::connect(connection_string)
4          .await
5          .expect("Failed␣to␣get␣database");
6
7      let p_created =
8          Person::<CreateRead>::
9          create(&database, "Kenneth".to_string(), "Fossen".to_string())
10         .await;
11
12     let p_read = Person::<CreateRead>::
13         read(&database, p_created.id)
14         .await;
15 }
```

5.6.1 1b: Results

This approach wraps everything into a module for each struct and organizes the code's public and private parts. The module contains a private module that maintains the capabilities for the struct, and it also contains the implementation of the essential traits for these capabilities, ensuring correct implementation. Supplying the `CreateDelete` struct to the Person struct will reveal only these functions for the developer, e.g., `Person::<CreateDelete>::read`, cleanly maintaining the object's capability and making it clear to the developer. Supplying the capability this way may seem unfamiliar to the developer, but it shows the developer's intent well in the code. The shown implementation does not encapsulate a unique token with its capabilities. However, since the capabilities are structs, we could extend the struct to carry the token to be passed further down the line, e.g., to functions and databases, making it compatible with a future capability database. The capability field approach is well-designed, cleanly putting everything into a module for the developer to maintain and relatively straightforward to use when developing.

On the other hand, this module has to contain everything regarding the struct. All imports, traits, implementations, structs, and functions must reside inside the module, which could lead to some confusion for the developer, making the learning curve for the design steeper. Manually maintaining this capability module for the struct can be a challenge. However, creating a sound library such as TypeState to manage the struct will relieve manual work for our developer. Also, the same problem occurs in this design as in E1a 5.5, that we will get duplicate implementations of code for certain combinations of capabilities, e.g., `CreateRead` and `CreateDelete` will have a duplicate `Create` function required by the traits. Due to the duplicate code issu¡

5.7 1c: Capabilities using macros

In this experiment, we need to provide some background in 5.7.1 on previous relevant work that we will use to describe the foundation for our example implementation in CapabilityApi 5.7.2. We will review the example implementation CapabilityApi with the same design in section 5.7.3, look at its results, then extract and design a library from the results of CapabilityApi in section 5.8 to use in the last experiment in section 5.9.

5.7.1 Previous work with macros

Inspiration for this work came from Zack Mullaly's discussions [29] and his work with macros [30]. Macros are a powerful feature in the Rust language that enables the developer to generate code from a simple pattern. An example of an earlier discussed macro that is very useful is the `vec!` macro (line 1 in listing 2.2), which uses the `macro_rules!` keyword in the background to generate the required code structure.

In Mullaly's discussions, he describes the following example 5.12 of how we can use traits and structs to guard and compose capabilities in Rust. He starts by defining a data structure, e.g., line 1, that he would like to store in a database. Then he creates structs that describe capabilities, e.g., line 7 `Save<T>(pub T)` and `Update<T>(pub T)`. These can hold the required data structure and describe the capability to perform. Continuing, he defines a trait that he implements the capability for the database. To save a user in the database, the developer needs to use the `handle_user_registration<DB>` function, where our database (DB) has the constraint `Capability<Save<User>>` implemented on line 23.

Listing 5.12: Macro: Mullaly's Capability example [30]: **impl Capability**

```
1  struct User {
2      pub email_address: String,
3      pub password_hash: String,
4      pub username: String,
5   }
6
7  struct Save<T>(pub T);
8  struct Update<T>(pub T);
9
10 struct SQLite {
11     db: Connection,
12 }
13
```

```
impl Capability<Save<User>> for SQLite {
    type Data = User;
    type Error = DatabaseError;

    fn perform(&self;, save_user: Save<User>) -> Result<User, DatabaseError> {
        // Execute a SQL query.
    }
}

fn handle_user_registration<DB>(db: &DB;, user: User) -> Result<User, DatabaseError>
    where DB: Capability<Save<User>>
{
    db.perform(Save(user))
}
```

Listing 5.13: Macro: Mullaly's macro example [30]

```
macro_rules! capability {
    ($name:ident for $type:ty,
        composing $({$operations:ty, $d:ty, $e:ty}),+) => {
        trait $name: $(Capability<$operations, Data = $d, Error = $e>+)+ {}

        impl $name for $type {}
    };
}
```

Mullaly continues to explain how his example can be generalized to a macro called **capability!** in listing 5.13 that he uses to compose new capabilities from these basic building blocks 5.12. Using the capability macro from 5.13, such as in example 5.14, generates a trait and impl and specifies the traits it requires in the design. On line 1 we invoke the macro **capability!** and it creates the traits needed for their implementation, e.g., line 6 and line 11. The developer can now implement these traits and use the earlier exemplified structure to require the data structure to be wrapped in capabilities when passed to functions performing actions against the database. This approach is similar to the earlier "Generics and TraitBounds" example 5.5. However, using a macro will create a capability to compose fine-grained traits for the developer, and we will implement this approach in the next section.

Listing 5.14: Macro: **capability!** macro use and result

```
1  capability!(CanChangeAndDeleteUserData for SQLite,
2              composing { Save<User>,   User, DBError },
3                          { Update<User>, User,   DBError },
4                          { Delete<User>, (),   DBError });
5
6  trait CanChangeAndDeleteUserData:
7      Capability<Save<User>, Data = User, Error = DBError>
8      + Capability<Update<User>, Data = User, Error = DBError>
9      + Capability<Delete<User>, Data = (), Error = DBError {}
10
11 impl CanChangeAndDeleteUserData for SQLite {}
```

5.7.2 Implemeting CapabilityApi with macro

We implemented a version of the capability macro into a project called CapabilityApi (found in the repository 5.3.2 under capabilityapi). The CapabilityApi follows the same design as Mullaly and uses the macro more or less as he described it. We changed the macro to support async operations, making us dependent on a particular `async_trait` crate because of using the `sqlx` crate when doing SQL queries. The rest we left intact such as Mullaly's example.

In the example 5.15 from our implementation of CapabilityApi, we focus only on one capability, `CanReadUserData` defined on line 2. This capability is accessible through an HTTP GET request to the URL `http://localhost:8080/users/{id}`, where `{id}` is a string identifier for the user. When the HTTP Server receives the request, the function `get_user` on line 4 get executed and calls `handle_find_user<DB>` on line 13. In the handler the Read capability is passed `db.perform(Read(name)).await` on line 17, and performs the implemented trait `Capability<Read<String>> for Database` on line 25 resulting in returing the requested data from the database to user through the initial function call.

We can now interact with our design through HTTP requests, successfully adhering to Mullaly's design with an HTTP Server. We did not extend CapabilityApi to use an HTTP request filter to interpret tokens to types, as this is considered a part of the next experiment 5.9. We did extend the HTTP server with a simple filter to grant access when receiving a Bearer token containing the string "Kenneth", tested and shown in listing 5.16.

Listing 5.15: Macro: Excerpt from: **CapabilityApi**

```
1  capability!(CanReadUserData for Database,
2  composing {Read<String>, User, DatabaseError});
3  // GET http://localhost:8080/users/{id}
4  pub async fn get_user(user: web::Path<String>, pool: web::Data<Database>) -> impl
       ↪ Responder {
5      let parsed_user: String = user.into_inner();
6      let db = pool.get_ref();
7      let u = handle_find_user(db, parsed_user)
8          .await
9          .expect("Failed␣to␣find␣user");
10     serde_json::to_string(&u).unwrap()
11 }
12
13 pub async fn handle_find_user<DB>(db: &DB, name: String) -> Result<User,
       ↪ DatabaseError>
14 where
15     DB: CanReadUserData,
16 {
17     db.perform(Read(name)).await
}

#[async_trait]
impl Capability<Read<String>> for Database {
    type Data = User;
    type Error = DatabaseError;

    async fn perform(&self, find_user: Read<String>) -> Result<Self::Data,
    ↪ Self::Error>  {
        let userid = find_user.0;
        let record = sqlx::query!(r#"SELECT␣*␣FROM␣users␣WHERE␣name␣=␣$1"#, userid,)
            .fetch_one(&self.db)
            .await
            .map_err(e e);

        let user = match record {
            Ok(r) => User {
                name: r.name.unwrap(),
                password: r.password.unwrap(),
            },
            _ => return Err(DatabaseError),
        };

        Ok(user)
    }
}

pub struct Database {
    pub db: Pool<Sqlite>,
}

pub struct User {
    pub name: String,
    pub password: String,
}
```

Listing 5.16: Macro: Testing Bearer token with: **CapabilityApi**

```
#!/bin/bash
URL="http://localhost:8080/users/"
HTTPCODE=‘curl -s -o /dev/null -s -w "%{http_code}\n" $URL‘
if [ $HTTPCODE = "401" ]; then
    echo "Access␣denied␣to␣$URL"
fi

echo "With␣Token␣for:␣$URL"
curl -s -H "Authorization:␣Bearer␣Kenneth" $URL | jq
```

5.7.3 1c: Results

The CapabilityApi example was easy to write and modular to fit most needs when structuring the code. One of the obvious challenges with this design is to make the developer adhere to the design and implement all the traits.

In CapabilityApi, the developer must create all the possible mutations needed in the capability macro. The design requires the developer to know and understand the structure before using the macro. Having no knowledge or struggling to understand the design, it is easy to circumvent the capabilities needed for interacting with the database intentionally or unintentionally. The design is unclear when a capability, such as Read, wraps a struct.

There are a few quirks, especially when accessing data in the Capability trait, where we have to use `find_user.0` to access the required string identifier, line 26. Mullaly's capabilities also do not carry the token of a capability but are extendable to do it.

The overall design looked promising, and we decided to generalize this structure and macro into a library to help the developer adhere to the design and make it clear when developing what capabilities are needed to be passed to functions to perform the required interaction.

5.8 2a: Designing and creating a library

In experiments 1a–c, the resulting main challenge is the amount of work the developer has to do to adhere to the design. It is also unclear when to use a capability and if the developer has implemented it for a particular struct. These issues are essential to address because if they are not, the developer may not see the use of such a library. We chose to use experiment 1c (CapabilityApi) 5.15 approach since it evaluated well against our evaluation criteria. We will call this library Capabilities and store the source code in the folder named `capabilities` in the repository 5.3.2.

5.8.1 Deciding design and developer interaction

From CapabilityApi, maintaining the code for the developer would become a challenge. To mitigate this, we had to think about how the developer should be interacting when using this capability design. We have seen several successful libraries in the Rust Community, e.g., Serde crate, that uses Procedural Macros with annotations to aid the developer. This approach is quite common in Rust, not only for external libraries but also for Rust's language features, such as deriving a debug trait on a struct, e.g., `#[derive(Debug)]`.

We decided that a similar approach might be an excellent way to remove some confusion and ease the developer's work to maintain the code design. We will create three procedural macros to be used to connect the design described in experiment 1c 5.15 together. These macros will do the job for the developer to adhere to the intended design.

5.8.2 The Library: capabilities

The library will contain all three macros and a filter. They are named `service`, `capabilities`, and `capability`. The first macro, `service`, will be responsible for wrapping the external entity we want to interact with a capability. In this example 5.17, we are annotating the main function to add a global service entity, such as in line 1, for our SQLite database. We added a custom name, "db", for the field name holding the database for our service instance, making it accessible to the developer by that name. The macro generates a hidden struct called `CapService` containing the field name “db” and a corresponding struct `CapServiceError` for error handling, shown in the example listing 5.18. There can only be one usage of this macro in our codebase, as it generates a static name for the service struct.

Listing 5.17: Lib: **service** usage

```
#[service(SqliteDb, name = "db")]
#[actix_web::main]
async fn main() -> Result<(), std::io::Error> {
    ...
}
```

Listing 5.18: Lib: **service** generated code

```
use sqlx::Pool;
use async_trait::async_trait;
pub type SqliteDb = Pool<Sqlite>; //custom type stored in the lib
pub struct CapService {
    db: SqliteDb,
}

pub struct CapServiceError;
```

The second macro, `capabilities`, is used on data structures that we want to pass to the service, and it also lists the capabilities we want to perform against the service. In the example listing 5.19, we can see on line 1 that we want the capabilities `Create`, `Read`, and `Delete` on the `Bowl` data structure, and we also have to specify the identifier field for the struct. The macro invokes an enhanced version of the capability macro that generates all the traits, such as in listing 5.20, connecting the struct to the `CapService` that the service macro created. The developer can now create functions that will issue the capability operation for the data structure against the service using the third macro.

Listing 5.19: Lib: **capabilities** usage

```
1 #[capabilities(Create, Read, Delete, id = "id")]
2 pub struct Bowl {
3     id: i64,
4     name: String,
5 }
```

Listing 5.20: Lib: **capabilities** generated code

```
pub trait CapCreateBowl: Capability<Create<Bowl>, Data = Bowl, Error =
    ↪ CapServiceError> {}
impl CapCreateBowls for CapService {}

pub trait CapReadBowl: CapabilityTrait<Read<Bowl>, Data = Bowl, Error =
    ↪ CapServiceError> {}
impl CapReadBowl for CapService {}

```

```
7 pub trait CapDeleteBowl: Capability<Delete<Bowl>, Data = Bowl, Error =
      ↪ CapServiceError> {}
8 impl CapDeleteBowl for CapService {}
```

The third macro is `capability`, and we append it to functions that we would like to perform a capability operation for a capability specified on the data structure. Each function can only implement one capability, such as in this example 5.21, `Create` for the `Bowl` data structure on line 9. The signature for the `create_db_bowl` has to be written by the developer, but the Rust compiler will warn if this signature is not compatible with the macro definition. This warning could be anything from returning other structs than `Bowl` and not using `CapServiceError` as an error value. As seen in this example, the capability provided is not limiting the SQL operations the developer can perform in the function as we perform both INSERT and SELECT. We will discuss this further in the design choices section for the library 5.8.3. The defined function `create_db_bowl` goes through a transformation when the macro is applied to it and spits out the indented design shown in example 5.22. The body of the function gets preserved and injected into the `perform` section of the `impl CapabilityTrait<Create<Bowl>> for CapService` on line 16. To reach the perform section of the code in 5.22 the design, we validate the capability received from the client against the correct capability for a `CapCreateBowl`. In the example, we need a `Create` capability to pass data further specified on line 9.

Listing 5.21: Lib: **capability** usage

```
1  #[capability(Create, Bowl)]
2  pub fn create_db_bowl(bowl: Bowl) -> Result<Bowl, CapServiceError> {
3      let _res = sqlx::query!(r#"INSERT␣INTO␣bowls␣(name)␣VALUES␣($1)"#, bowl.name)
4          .execute(&self.db)
5          .await
6          .expect("unable␣to␣create␣bowl");
7      let b = sqlx::query_as!(Bowl, r#"SELECT␣*␣FROM␣bowls␣WHERE␣name␣=␣$1"#, bowl.name)
8          .fetch_one(&self.db)
9          .await
10         .expect("Didn't␣fint␣any␣bowls");
11     Ok(b)
12 }
```

Listing 5.22: Lib: Excerpt from **capability** fn code generated

```
1 pub async fn create_db_bowl<Service>(
2     service: &Service,
3     param: Bowl,
4     cap: ::capabilities::Capability,
5 ) -> Result<Bowl, CapServiceError>
6 where
7     Service: CapCreateBowl,
8 {
 9     let valid = ::capabilities::Create { data: param };
10     if valid.into_enum().eq(&cap) {
11         service.perform(valid).await
12     } else {
13         Err(CapServiceError)
14     }
15 }
16 impl CapabilityTrait<Create<Bowl>> for CapService {
17     type Data = Bowl;
18     type Error = CapServiceError;
19     fn perform<'life0, 'async_trait>(&'life0 self, action: Create<Bowl>,
20     ) -> ::core::pin::Pin<
21         Box<dyn ::core::future::Future<Output = Result<Self::Data, Self::Error>>
22                 + ::core::marker::Send
23                 + 'async_trait,>,>
24     where 'life0: 'async_trait, Self: 'async_trait,
25     {
26         Box::pin(async move {
27             if let ::core::option::Option::Some(__ret) =
28                 ::core::option::Option::None::<Result<Self::Data, Self::Error>>
29             {
30                 return __ret;
31             }
32             let __self = self;
33             let action = action;
34             let __ret: Result<Self::Data, Self::Error> = {
35                 let bowl: Bowl = action.data;
36                 {
37                     // injected function block specified by the developer
38                 }
39             };
40             #[allow(unreachable_code)]
41             __ret
42         })
43     }
44 }
```

We need to map external capabilities to these internal functions in the last connecting piece to have a RESTful API. The library provides an HTTP filter mapping external capabilities from a client to these functions generated by the above macros. An HTTP request with a capability token will run the function with the corresponding capability. In the example 5.23, we are viewing the HTTP Handler for creating a new bowl. The filter has preprocessed the incoming capability from a Bearer token to the `Capability` on line 5. The filter gives us the correct capability to pass on to the `create_db_bowl` function under the name variable name cap. We proceed to validate, as shown in example 5.22 on line 10, the required capability against the received capability and throws an error if invalid. A full listing of the implementation and expanded code is available in the repository (see section 5.3.2) under the folder `simple-api` as a file named `simple_api.rs`. The filter implementation is explained in experiment 2b 5.9 because the implementation depends on the choices made for the capability system.

Listing 5.23: Lib: **filter** example

```
#[post("/bowls/")]
pub async fn create_new_bowl(
    json: web::Json<BowlsDTO>,
    svc: web::Data<CapService>,
    cap: Capability,
) -> impl Responder {
    let svc = svc.get_ref();
    let newbowl: Bowl = Bowl {
        id: 0,
        name: json.name.to_owned(),
    };

    println!("{:#?}", newbowl);
    println!("Cap:␣{:#?}", cap);
    match create_db_bowl(svc, newbowl, cap).await {
        Ok(bowl) => HttpResponse::Ok().json(bowl),
        _ => HttpResponse::BadRequest().json("{␣\"request\":␣\"bad␣request\"␣"),
    }
}
```

5.8.3 Decisions made when designing the library

When writing the procedural macros library, there was a decision only to develop this library to a proof-of-concept (PoC) for capabilities. We implemented only a few capabilities named `Create`, `Read`, `Update`, `Delete`, `ReadAll`, `DeleteAll`, and `UpdateAll` to not let the scope get too big.

Also, we decided that these capabilities should not carry the unique token that arrives from the web client. There are no existing databases that support capabilities when querying for data, so there is no need to preserve this information. The capabilities are extendable for this feature, with some work for future databases that support capabilities.

We are not restricting the developer when writing an SQL Statement inside a capability function block, e.g., a DELETE SQL statement inside a function that only has a `Create` listed as their capability, such as mentioned when explaining the capability macro in section 5.8.2. The decision not to restrict this is because we would want to have a database that supports receiving capabilities such that the mentioned case is not a problem.

Further, we have used static names for the `CapService` and `CapServiceError`, restricting the developer to only have one service in their API. We could add a feature for the developer to name the struct himself or keep the default name when needed. We wanted to avoid too many new parameters for this PoC library and not increase the learning curve.

Lastly, all the macro-generated structs, traits, implementations, and fn blocks are in the same scope where they are used, meaning that we can only create a single file API for the developer. A single file API will suffice for the PoC, but it will become a problem for larger projects and needs a resolution before being used in such a manner. The implementation for Simple–api is this way; everything is stored in the file main.rs and works as intended.

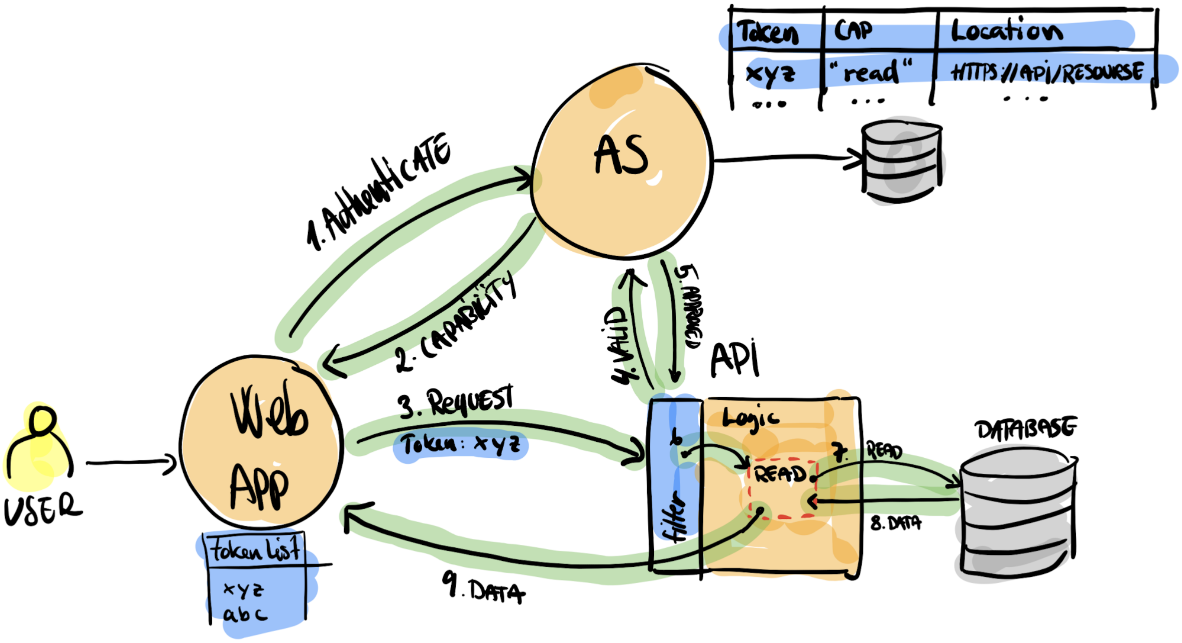
5.8.4 2a: Results

We created a library with three procedural macros to implement the design we used in CapabilityApi. The macros in the library do most of the work for the developer and reduce the design burden. The developer only needs to understand how the macros are connected to use this library. When using, e.g., the capabilities macro on a struct, we must implement the missing traits before our code compiles. Later extending the capabilities for a struct is done by adding the capability operation, e.g., `Update`, to the `capabilities` macro for the struct and implementing the corresponding capability function block. Together with the implementation-specific filter, the macros give the developer an easy task adhering to the indented capability design and harnessing its features described in API Overview figure 5.2

Currently, there are several limitations to how to use this library, and they are all mentioned earlier in this section 5.8.3. The most significant limitation for developers is that they cannot structure the code exactly the way they want. We have only tested the library with simple data models. Complex models such as a Person data structure also containing a list of Addresses could pose a problem and might need an extension of the library. These issues need to be verified and solved in the future development of the library. Also, the macros are not validating that the code block's content is doing the correct capability operation, e.g., a function annotated with Update and the developer writes a DELETE SQL statement. Having capability tokens such as Biscuits and a database that handles data querying with such tokens could solve this.

5.9 2b: Putting it together to a RESTful capability system

In experiment 2b, we are looking to integrate the previously created library 5.8 from 2a and a client with an authorization server (AS) in a distributed system, such as in the following figure 5.3. We will first cover a few design choices to limit the implementation scope to get a proof-of-concept. Then we will shortly describe the work we did on GNAP, describe the interactions in the capability system, and visualize the interactions flow between the different components when trying to achieve the described capability system in section 5.4 and shown in figure 5.3.



Token	CAP	Location
xyz	"read"	HTTPS://API/RESOURSE
...	...	...

Figure 5.3: Sketch of a capability-based system

5.9.1 Decisions made before implementing the capability system

OAuth is the dominant authorization authority framework in the industry, but it has issues that they are trying to resolve in the next iteration of the OAuth version. Justin Ritcher has been working with OAuth and has seen these problems and created OAuth.XYZ, now named GNAP, to address OAuths issues. To avoid the complex nature of OAuth and remove the requirement for a browser, we have chosen to use a minimal implementation of GNAP as the authorization authority server (AS). We have only implemented the Redirect-based Interaction flow to reduce the scope, as it is the most suitable flow for a web client. GNAP relies on the Signing HTTP Messages standard [3], and this standard is still in revision, and there are no working implementations in Rust. We decided not to implement this central security feature in GNAP since we are not evaluating the security for GNAP but are instead harnessing the power of GNAP's transactional approach. In the Tokens & Authorization Authorities (chapter 4), we reviewed several token types that can carry capabilities, and we have chosen for simplicity to use Opaque Tokens 4.1.5. An Opaque Token is a random string token that does not serve any meaning except at the AS. Future development of the library should consider switching to Biscuits 4.1.4 or similar tokens that are capability capable.

5.9.2 Implementing a RESTful capability system

Since GNAP is still in draft at Internet Engineering Task Force (IETF) and only has reference implementation in Java, we decided to extend an implementation we found in a repository on GitHub from the user dskyberg at `https://github.com/dskyberg/gnap`. We cloned and detached the repository when evaluating this GNAP implementation. We found the implementation is partial and decided to extend it with Opaque Token generation, token introspection, an authentication- and authorization page, and completed the Redirect-based Interaction flow for the protocol. These features result in a minimum viable solution for testing a capability system.

The Redirect-based Interaction flow seen in listing 5.4, which we implemented following the GNAP draft for the AS, consists of 11 steps to make up a transaction.

(1) The user Starts a Session with the client, and (2) the client initiates a GrantRequest 5.24 Requesting access from the AS. The request create an initial transaction holding the claim (line 4–17).

(3) The AS responds with an Interaction Request containing a unique ID supplied to our (4) client to redirect the user for (5) authentication and (6) authorization.

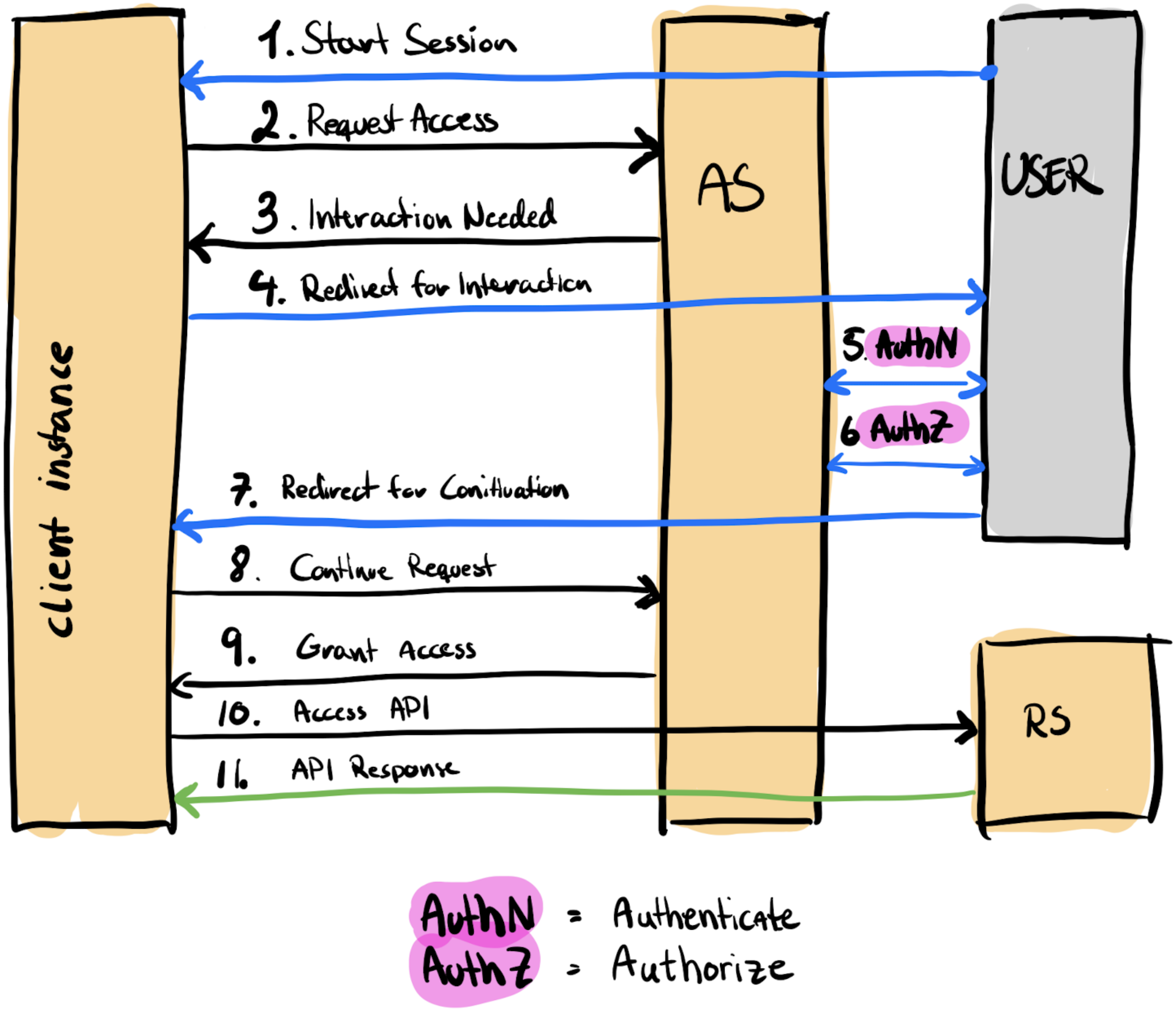


Figure 5.4: GNAP: Redirect-based Interaction flow sketch from specification [35]

(7) The AS will continue the process by redirecting the user back to the client with a continuation id. (8) The client uses the supplied id in a Continuation Request sent to the AS.

(9) The AS issues the granted access as capability tokens as shown in listing 5.25. The Opaque token on 5, in this example, (10) is used when requesting to create a resource on the RS, and (11) the RS responds accordingly.

Listing 5.24: GNAP: **GrantRequest** JSON from step (2)

```
{
 "access_token": [
    {
      "label": "create_bowls",
      "access": [
        {
          "type": "waterbowl-access",
          "actions": [
            "create"
          ],
          "locations": ["http://localhost:8080/bowls/"]
        }
      ],
      "flags": [
        "bearer"
      ]
    },
  ],
  "client": "7e057b0c-17e8-4ab4-9260-2b33f32b2cce",
  "interact": {
    "start": ["redirect"],
    "finish": {
      "method": "redirect",
      "uri": "localhost:8000/gnap/auth",
      "nonce": "e744d1f0-e601-455f-a696-57c6b0a21280"
    }
  }
}
```

Listing 5.25: GNAP: **Granted Access Token** JSON from step (9)

```
{
  "access_token": [
      {
        "label": "create_bowls",
        "value": "Y6M65CQM2X4GJN8KY6FH8D8AHD28JX3G4WZP8JBEM77BTC5", //Opaque Token
        "access": [
          {
            "type": "waterbowl-access",
            "actions": [
              "create"
            ],
            "locations": ["http://localhost:8080/bowls/"]
          }
        ],
        "flags": [
          "bearer"
        ]
      },
    ],
  }
```

We used the implementation Simple–api, that we created in section 5.8, as the resource server (RS) for this experiment. The RS uses the created capabilities library to adhere to the design and require, e.g., the capability `Create` to create a Bowl resource on the RS. To make this interaction easier to work with, we also created a simple javascript client with NextJS. The client follows the Redirect-based interaction flow with the AS and stores the issued capabilities within the client. For each action we would like to achieve at RS, (1) the client sends the correct capability token with the HTTP request to the RS. When the RS receives an HTTP request with the token, (2) it introspects the token by passing it to the introspection point on the AS. (3) If the token is valid, the AS responds with a capability operation and resource path. (4) The capability operation is passed from the filter and evaluated at the resource paths function, such as shown in figure 5.5. (5)(6)(7) only executes if the capability is correct and can perform the correct operation.

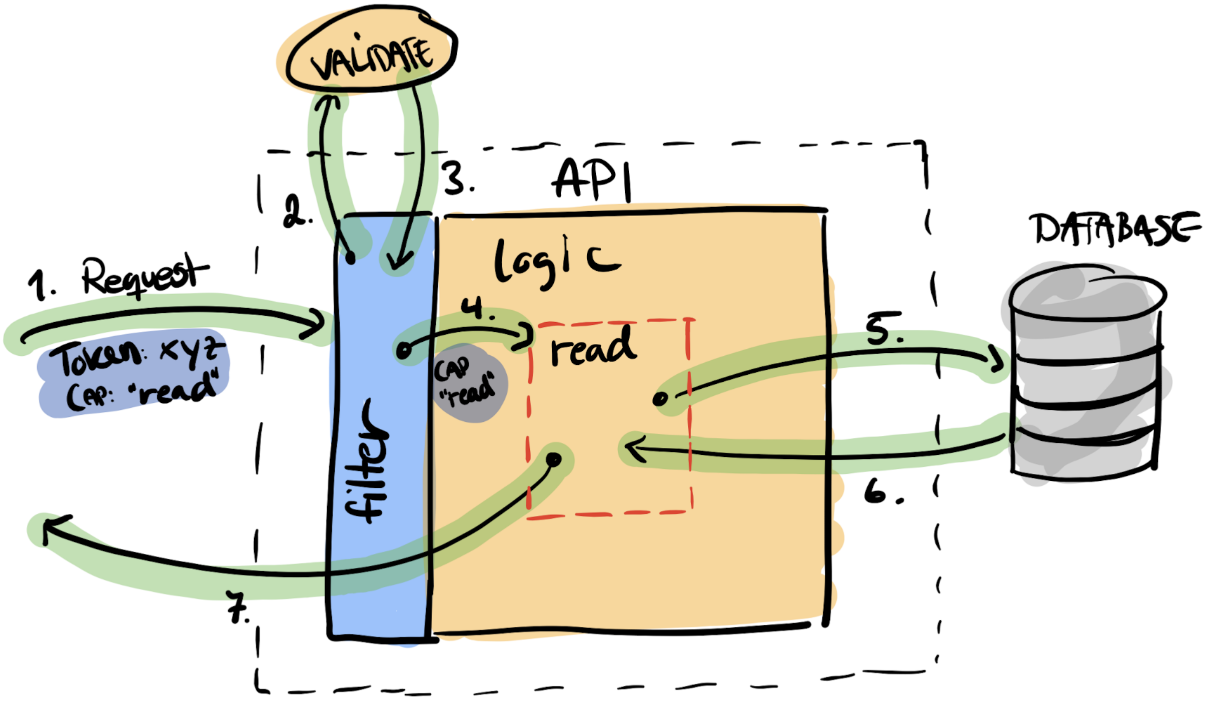


Figure 5.5: System: Sketch of the API layer

We will not cover more implementation details of the GNAP protocol here. However, the entire GNAP source code is accessible in the `gnap` folder located in the shared repository for the proof-of-concept 5.3.2.

5.9.3 2b: Results

We have successfully created a capability system we described and created a sketch of in section 5.4. The client holds the capabilities, the AS issues capabilities with tokens, action, and a reference, and the RS maintains these capability restrictions through the created capabilities library. The GNAP message format in a GrantRequest, contains enough information to represent capabilities as object references and can be stored in the client, such as the sketch from Capability Myths 3.8, and used when interacting with the RS. The capabilities library aid the developer when creating the RS, as we previously evaluated. The capabilities are coded into the RS dictating the required capability when interacting with data. The filter helps the RS interact with the AS to evaluate the capability tokens received from the client to the proper function calls in the RS.

We have only achieved the minimal features of a capability-based system. The current design only allows the client to request a token with one capability action, such as the GrantRequest for Create in listing 5.24. GNAP allows for more capability actions, e.g., the actions array contains the actions create and read, but our design in the library does not handle this case. Challenges such as delegation and confinement (Capability Myths 3.2.3 figure 3.9) are not solved. GNAP's initial transaction design supports delegation and confinement, but none of the drafts for GNAP specify these properties; this is future work. The entire solution has not been tested on a large scale, e.g., with multiple APIs and stress testing. Also, it lacks a complete GNAP implementation and a management solution of the possible capabilities and users in the system.

Chapter 6

Discussion, Conclusion and Future work

In this chapter, we will take a second look at our research goals before we discuss the findings in chapter 5. We will interpret the results and how they fare against our evaluation criteria and discuss the capability-based system solution we created. This solution will provide the basis for answering our research questions from section 1.3 in our conclusion before we mention future work in the last part of this chapter.

6.1 Discussion

Our research goal is to implement CBAC in a capability-based system with RESTful APIs to limit confused deputy problems. Previous research has shown that we can reduce possible confused deputies between two principals acting together by harnessing the flexibility of capabilities.

We decided to approach this using the Rust programming language, focusing on using built-in language features to create a design pattern that could hold capabilities inside a RESTful API. To achieve a capability-based system, we need an authorization authority, and we chose GNAP because it supports capabilities. The design pattern and GNAP would give us a capability-based system that supports CBAC.

We have shown two things through these experiments (1a–c and 2a–b). (1) We have composed code in Rust, a non-capability language, into a design pattern and library that enabled functions to require capability types to execute within a RESTful API. (2) Combining the RESTful API built with the created library with a GNAP AS in a system enables a client to request capability tokens from the AS and use capability tokens to access the RS, just as our RESTful API, to run functions in the code.

We were uncertain that using Rust would make this library possible as it is a non-capability language and evaluates poorly in Hillert's thesis [20]. However, surprisingly it has powerful language abstractions that enabled us to create a design and implement this library to ease the developer's job in adhering to the created capability design within the RESTful API.

The implemented library enables the developer to specify the available CRUD operations for a data structure in the RESTful API. It aligns with the PoLP and shows the implemented operations for the developer in the RESTful API. Issuing a valid token from the AS with an unimplemented operation for a data structure will not result in any execution of functions in the API. It is restricting users from "acting outside their intended permissions" [11].

The myths Miller [27] describes have made it easier to see the difference between classic RBAC and ABAC compared to CBAC. The illustration 3.8 shows us that the user principals are holding the capabilities and not the resources. The capability-based system we have created aligns with this visualization of the system as expected. The user authorizes the AS to delegate the user's capabilities to a client that uses these capability tokens to interact with the RS principal.

Hardy [19] argued that such a capability-based system would reduce confused deputy problems if not removing them, and we believe that this approach carries the same properties. Also, GNAP needs an extension to support the last two properties, (1) delegation and (2) confinement, which have not been implemented in this system.

We mentioned that this work would be an addition to the work of Yarygina. Appending this capability-based system to her implementation of MissFire [40] will enable secure communication between authenticated services and restrict function calls between each service. Requests between the different RESTful APIs in an MSA have to be equipped with a capability token to flow to other services.

We focused on design patterns that enabled capabilities that only use Rust's built-in language features that we could generalize to a library. We have shown through the creation of the library that there is an approach that can resemble CBAC without having native support for object capabilities in the language. However, there have been recent discussions in the Rust community to enhance Rust's approach to capabilities and solve the "context problem", such as these blog posts [25]and discussion [24]. If the Rust standard accepts these discussed additions, it could lead to a different approach that may further enhance a library, such as the created capabilities library in 2a 5.8

The library is also developed only as a proof-of-concept, and as mentioned 5.8, there are several limitations due to the library's design. The current state of the library does not make it ready for production, but it is a start at approaching CBAC in a RESTful API.

Since the focus was not on the security of GNAP but on the transactional behavior of GNAP and support for capabilities it provides, we skipped essential parts of the implementation mentioned in 5.9.1. There remains work to be done further to enhance this capability system, and one of them is to ensure the GNAP security mechanisms work as intended in this setup. Also, the draft we leaned on for implementing GNAP has expired, and a new version of the document is in place. Other standards that we need to rely upon, e.g., Signing HTTP Messages, are also currently in draft. The workgroup responsible for these drafts also just had a hackathon where they uncovered several minor implementation challenges in the reference implementation and walked away with new knowledge to add to GNAP [34]. Using such an early version of a protocol might be risky as it is subject to change or will not make it out of the design phase. There seems to be a community around GNAP that wants it to succeed. It might even take up the competition with today's industry-standard OAuth 2 due to GNAP's transactional approach to authorization.

Also, delegation and confinement are the two last parts of the capability system that we need to implement, and they were excluded from the scope due to the share size of the project. To achieve delegation and confinement, GNAP needs to be extended.

We represent capabilities in this system through Opaque Tokens. The tokens do not carry any information, except capability operation and a resource reference, but not in the same manner as JWTs or Biscuits, and were chosen for simplicity. However, we believe the unique capability token should carry more information for fine-grained access control. In the capability-based system, we created a capability that represents a table. We believe a capability in this system should represent one row in the database instead. We chose the capability-as-table approach since we believe the database should evaluate the token to return the data for the capability. However, there is no such database or extension that evaluates a capability to row to the writer's knowledge at the time of writing. It is also why the designed library does not carry the capability token into functions.

6.2 Conclusion

This research explores how we can achieve CBAC in a RESTful API using Rust and GNAP. Through our experiment, we constructed a system so we can confidently answer the research questions listed in 1.3.

RQ1: "Can we structure our code or utilize Rust's ecosystem to achieve CBAC within a RESTful API?"

Through experiment 1a–c, we explored a few approaches to incorporating capabilities into the code itself. Two approaches showed some weaknesses that we would not want in our design, but 1c, with the help of macros, we built a complete RESTful API that allowed us to guide the developer to better decisions when creating the API. We successfully structured the code and generalized it into a library through experiment 2a. Further, we used this library in experiment 2b.

RQ2: "Can CBAC give fine-grained access control and avoid confused deputy problems in a RESTful API?"

CBAC can, such as ABAC, give us fine-grained access control on actions when interacting with the RESTful API. The library allows us to specify the actions available to the system's user that the developer implements. The current implementation of the library has coarse-grained access control, and the access represents access to a table instead of a row of data. We chose not to implement fine-grained, as we believe this would require a capability-based database that supports representing a capability as a row and not a table.

We implemented a minimal version of a GNAP AS. Due to the lack of implementation of standards required, e.g., Signing HTTP Messages for Rust, we did not implement security features such as HTTP signing of messages the GrantRequest. Therefore we cannot document removing any confused deputy, even though the theory of a capability-based system suggests this.

RQ3: "Could GNAP help us realize capability-based fine-grained access control in a RESTful API?" The flexible format of Grant Requests and Responses in GNAP is undoubtedly helping us construct capability-based access control. We are leaning on the current standard that allows us to request multiple tokens with tiny scopes that give us a coarse-grained access control. We could narrow this down to fine-grained access control with a capability-based database.

We looked at different design patterns used internally in the API with Rust. We extracted the design pattern to a library using Rust's language features, making the developer's job easier. Further, we constructed a capability-based system using GNAP, the RESTful API, and created a minimal client that would use capabilities to access the RESTful API. We have created a capability-based system for the modern backend. The design adheres to the properties of a capability-based system and can serve as a basis for future research on this topic.

6.3 Future work

We now have proof-of-concept of a capability-based system within a RESTful API-style architecture, and it can form the basis of future research regarding GNAP and CBAC in RESTful APIs. Listed in no particular order are a few ideas to build on top of this research we can suggest:

- When we created the library, we made some design choices that did not scale well in other projects. Sorting these issues listed in 5.8.3 and supporting complex data models would lead to a more adaptable library that can be tested for more extensive use cases and measure performance against a similar ABAC solution.

- Another interesting approach would be to evaluate how this capability-based system handles a complex microservice architecture. We have created a capability-based system and only evaluated it as a single RESTful API and not against other architecture, such as event-driven architectures.
- We build a minimal implementation of a client for the proof-of-concept. Researching how a client would fare when each capability token is an object reference in a complex model would help discover potential issues and limits and lead to best practices or usability studies for clients in a capability-based system.
- A limitation of this thesis of CBAC in a capability-based system is that we lack a database that supports receiving capability tokens. A capability database would receive a capability token, and this would represent a row of data in a table. There are many challenges in this scenario, such as connecting the API and the database? Who owns the tables? How does this work with complex models? Having such a database could let an API represent a database connection with a capability token. Then as a proper capability-based system, letting the API pass capability tokens from the user to the database to return data would further bring APIs closer to PoLP and reduce confused deputies such as SQLi.
- We need tokens to support querying data from a capability-based database. Would Biscuits suffice for this purpose? Can they be extended to serve this purpose? Also, how would Biscuits or similar tokens fare in this capability-based system that we created?
- GNAP is in development, and there are only a few libraries implementations for clients and servers. This work is mainly done within the working group. However, they need help researching different security aspects, such as fully implementing all the flows with Signing HTTP Message support and similar projects that would help the standard greatly toward finalization.
- Extending GNAP to support the two last capability myths, delegation, and confinement, would require an extension of the GNAP core protocol into a new RFC.

Bibliography

[1] Wikipedia:10,000 most common passwords - wikipedia.
URL: https://en.wikipedia.org/wiki/Wikipedia:10,000_most_common_passwords.

[2] HTTP cookie.
URL: https://en.wikipedia.org/w/index.php?title=HTTP_cookie&oldid=1081477230. Page Version ID: 1081477230.

[3] Annabelle Backman, Justin Richer, and Manu Sporny. HTTP message signatures.
URL: https://datatracker.ietf.org/doc/draft-ietf-httpbis-message-signatures. Num Pages: 82.

[4] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society. ISBN 978-1-891562-35-8. doi: 10.14722/ndss.2014.23212.
URL: https://www.ndss-symposium.org/ndss2014/programme/macaroons-cookies-contextual-caveats-decentralized-authorization-cloud/.

[5] John Bradley, Andrey Labunets, Torsten Lodderstedt, and Daniel Fett. OAuth 2.0 Security Best Current Practice.
URL: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16.

[6] Geoffroy Couprie. Biscuit, the foundation for your authorization systems - Clever Cloud, April 2021.
URL: https://www.clever-cloud.com/blog/engineering/2021/04/12/introduction-to-biscuit/.

[7] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, April 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2019.01.001.
URL: https://www.sciencedirect.com/science/article/pii/S0164121219300019.

[8] José Duarte and António Ravara. Retrofitting typestates into rust. In *25th Brazilian Symposium on Programming Languages*, SBLP'21, page 83–91, 2021. doi: 10.1145/3475061.3475082.
URL: https://doi.org/10.1145/3475061.3475082.

[9] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. 2(2):115–150. ISSN 1533-5399. doi: 10.1145/514183.514185.
URL: https://doi.org/10.1145/514183.514185.

[10] Kenneth Fossen. Exploring Capability-based security in software design with Rust, 6 2022.
URL: https://github.com/spydx/capability-poc.

[11] OWASP Foundation. A01 Broken Access Control - OWASP Top 10:2021, .
URL: https://owasp.org/Top10/A01_2021-Broken_Access_Control/.

[12] OWASP Foundation. A04 Insecure Design - OWASP Top 10:2021, .
URL: https://owasp.org/Top10/A04_2021-Insecure_Design/.

[13] OWASP Foundation. A05 Security Misconfiguration - OWASP Top 10:2021, .
URL: https://owasp.org/Top10/A05_2021-Security_Misconfiguration/.

[14] OWASP Foundation. A07 Identification and Authentication Failures - OWASP Top 10:2021, .
URL: https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/.

[15] OWASP Foundation. OWASP Top 10:2021, .
URL: https://owasp.org/Top10/.

[16] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. A capability-based security approach to manage access control in the Internet of Things. *Mathematical and Computer Modelling*, 58(5):1189–1205, September 2013. ISSN 0895-7177. doi: 10.1016/j.mcm.2013.02.006.
URL: https://www.sciencedirect.com/science/article/pii/S089571771300054X.

[17] Eran Hammer-Lahav. The OAuth 1.0 protocol.
URL: `https://datatracker.ietf.org/doc/rfc5849`. Num Pages: 38.

[18] Dick Hardt. The OAuth 2.0 authorization framework.
URL: `https://datatracker.ietf.org/doc/rfc6749`. Num Pages: 76.

[19] Norm Hardy. The Confused Deputy, December 2003.
URL: `https://web.archive.org/web/20031205034929/http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html`.

[20] Jessica Hillert. *A Comparison of the Capability Systems of Encore, Pony and Rust.*
URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-395655`.

[21] T. Inoue, H. Asakura, H. Sato, and N. Takahashi. Key Roles of Session State: Not against REST Architectural Style. pages 171–178. IEEE, 2010. ISBN 978-1-4244-7512-4. doi: 10.1109/COMPSAC.2010.64. ISSN: 0730-3157.

[22] Michael Jones, John Bradley, and Nat Sakimura. JSON web token (JWT).
URL: `https://datatracker.ietf.org/doc/rfc7519`. Num Pages: 30.

[23] Niel Madden. *API Security in Action.* Manning Publications, November 2020. ISBN 978-1-61729-602-4.
URL: `https://www.manning.com/books/api-security-in-action`.

[24] Tyler Mandry. Contexts and capabilities in Rust - Tyler Mandry.
URL: `https://tmandry.gitlab.io/blog/posts/2021-12-21-context-capabilities/`.

[25] Tyler Mandry. Blog post: Contexts and capabilities in Rust - language design, December 2021.
URL: `https://internals.rust-lang.org/t/blog-post-contexts-and-capabilities-in-rust/15833`.

[26] Ross Mason. How REST replaced SOAP on the web: What it means to you.
URL: `https://www.infoq.com/articles/rest-soap/`.

[27] Mark S Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished. page 15.

[28] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[29] Zack Mullaly. Using "Capabilities" to design safer, more expressive APIs in Rust.
URL: `https://www.reddit.com/r/rust/comments/7rmgxo/using_capabilities_to_design_safer_more/`.

[30] Zack Mullaly. Zack Mullaly, January 2018.
URL: `https://web.archive.org/web/20180120000131/http://www.zsck.co/writing/capability-based-apis.html`.

[31] Parthipan Natkunam. JWT signature stripping attack: A practical primer.
URL: `https://medium.com/geekculture/jwt-signature-stripping-attack-a-practical-primer-2d8f9ca00c2f`.

[32] Stack Overflow. Stack Overflow Developer Survey 2021, 2021.
URL: `https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021`.

[33] Aaron Parecki. It's Time for OAuth 2.1.
URL: `https://aaronparecki.com/2019/12/12/21/its-time-for-oauth-2-dot-1`.

[34] Justin Richer. The GNAPathon.
URL: `https://justinsecurity.medium.com/the-gnapathon-57ee110508ac`.

[35] Justin Richer, Aaron Parecki, and Fabien Imbault. Grant negotiation and authorization protocol.
URL: `https://datatracker.ietf.org/doc/draft-ietf-gnap-core-protocol`. Num Pages: 168.

[36] Kirsten S. Cross Site Request Forgery (CSRF) | OWASP Foundation.
URL: `https://owasp.org/www-community/attacks/csrf`.

[37] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975. ISSN 1558-2256. doi: 10.1109/PROC.1975.9939.

[38] Ekaterina Shmeleva. How Microservices are Changing the Security Landscape. December 2020.

URL: https://aaltodoc.aalto.fi:443/handle/123456789/97601. Accepted: 2020-12-20T18:14:19Z.

[39] Prabath Siriwardena and Nuwan Dias. *Microservices Security in Action.* Manning Publications, July 2020. ISBN 978-1-61729-595-9.
URL: https://www.manning.com/books/microservices-security-in-action.

[40] Tetiana Yarygina. *Exploring Microservice Security.* The University of Bergen, October 2018. ISBN 978-82-308-3665-1.
URL: https://bora.uib.no/handle/1956/18696. Accepted: 2018-11-02T14:55:22Z.

Appendix A

Trait and Bounds example

A.1 Full code for Trait and Bounds example

Listing A.1: Trait and bounds example: `trait-bounds\example\bounds.rs`

```rust
use async_trait::async_trait;
use rand;
use sqlx::sqlite::SqlitePoolOptions;
use sqlx::{Pool, Sqlite};
use std::fmt::Debug;

#[derive(Debug, PartialEq)]
struct Person {
    id: i64,
    firstname: String,
    lastname: String,
}

trait CreateRead<S> {
    fn create(firstname: String, lastname: String) -> S;
    fn read(id: i64) -> S;
}

impl CreateRead<Person> for Person {
    fn read(id: i64) -> Person {
        Person {
            id: id,
            firstname: "kenneth".to_string(),
            lastname: "fossen".to_string(),
        }
    }

    fn create(firstname: String, lastname: String) -> Person {
        Person {
            id: rand::random(),
            firstname: firstname,
            lastname: lastname,
        }
    }
}

struct Service {
    con: Pool<Sqlite>,
}

#[async_trait]
trait DBCreateRead<T: CreateRead<T>> {
    async fn read_db(&self, id: i64) -> T;
    async fn create_db(&self, data: T) -> T;
}

#[async_trait]
impl DBCreateRead<Person> for Service {
    async fn read_db(&self, id: i64) -> Person {
        let r = sqlx::query!(
            r#"SELECT␣id,␣firstname,␣lastname␣FROM␣person␣WHERE␣id␣=␣$1"#,
            id
        )
        .fetch_one(&self.con)
        .await
        .expect("Failed␣to␣query␣database");

        Person {
            id: id,
            firstname: r.firstname,
            lastname: r.lastname,
        }
    }

    async fn create_db(&self, data: Person) -> Person {
        let _r = sqlx::query!(
            r#"INSERT␣INTO␣person␣(id,␣firstname,␣lastname)␣VALUES␣($1,␣$2,␣$3)"#,
            data.id,
            data.firstname,
            data.lastname
        )
        .execute(&self.con)
        .await
        .expect("Failed␣to␣insert␣Person␣into␣database");
        data
    }
}

#[tokio::main]
async fn main() {
    let con_str = "sqlite:bounds_persons.db";
    let db: Pool<Sqlite> = SqlitePoolOptions::new()
        .connect(con_str)
        .await
        .expect("Failed␣to␣create␣database");

    let _service = Service { con: db };

    let _p1 = Person::create("Kenneth".to_string(), "Fossen".to_string());

    let create_res = Service::create_db(&_service, _p1).await;
```

```rust

    let read_res = Service::read_db(&_service, create_res.id).await;

    let _p2 = Person {
        id: rand::random(),
        firstname: "Kenneth".to_string(),
        lastname: "fossen".to_string(),
    };
    let _p2_createdb = Service::create_db(&_service, _p2).await;

    assert_eq!(create_res, read_res);
    println!("Create<{:#?}>␣->␣Read<{:#?}>", create_res, read_res);
}
```

Appendix B

TypeState example

B.1 Full code for TypeState example

Listing B.1: TypeState example: `trait-bounds\example\typestate.rs`

```rust
use async_trait::async_trait;
use person::*;
use sqlx::{Pool, Sqlite, SqlitePool};

mod person {
    use async_trait::async_trait;
    use sqlx::{Pool, Sqlite};

    #[derive(Debug, PartialEq)]
    pub struct Person<Cap: Caps> {
        pub id: i64,
        pub firstname: String,
        pub lastname: String,
        pub cap: Cap,
    }

    #[derive(Debug, PartialEq)]
    pub struct CreateRead;
    #[async_trait]
    pub trait CreateReadCap {
        async fn create(
            db: &Pool<Sqlite>,
            firstname: String,
            lastname: String,
        ) -> Person<CreateRead>;
        async fn read(db: &Pool<Sqlite>, id: i64) -> Person<CreateRead>;
    }
    /*pub struct Delete;
    pub trait DeleteCap {
        fn delete(db: &Pool<Sqlite>,) -> Person<Delete>;
    }
    pub struct CreateDelete;
    pub trait CreateDeleteCap {
        fn create(db: &Pool<Sqlite>,firstname: String, lastname: String) -> Person<CreateDelete>;
        fn delete(db: &Pool<Sqlite>,) -> Person<CreateDelete>;
    }
    pub struct CreateUpdate;
    pub trait CreateUpdateCap {
        fn create(db: &Pool<Sqlite>,firstname: String, lastname: String) -> Person<CreateUpdate>;
        fn update(db: &Pool<Sqlite>) -> Person<CreateUpdate>;
    }
    pub struct CreateReadUpdate;
    pub trait CreateReadUpdateCap {
        fn create(db: &Pool<Sqlite>,firstname: String, lastname: String) -> Person<CreateReadUpdate>;
        fn read(db: &Pool<Sqlite>) -> Person<CreateReadUpdate>;
        fn update(db: &Pool<Sqlite>,fistname: String, lastname: String) -> Person<CreateReadUpdate>;
    }
    */
    mod __private {
        pub trait Caps {}
    }

    pub trait Caps: __private::Caps {}
    impl<__T: ?::core::marker::Sized> Caps for __T where __T: __private::Caps {}
    #[async_trait]
    impl __private::Caps for CreateRead {}
    /*
    impl __private::Caps for Delete {}
    impl __private::Caps for CreateDelete {}
    impl __private::Caps for CreateUpdate {}
    impl __private::Caps for CreateReadUpdate {}
    */
}

#[async_trait]
impl CreateReadCap for Person<CreateRead> {
    async fn create(db: &Pool<Sqlite>, fistname: String, lastname: String) -> Person<CreateRead> {
        let person = Person::<CreateRead> {
            id: rand::random(),
            firstname: fistname,
            lastname: lastname,
            cap: CreateRead,
        };

        sqlx::query!(
            r#"INSERT INTO person (id, firstname, lastname) VALUES ($1, $2, $3)"#,
            person.id,
            person.firstname,
            person.lastname
        )
        .execute(db)
        .await
        .expect("Failed to write to database");
        person
    }

    async fn read(db: &Pool<Sqlite>, id: i64) -> Person<CreateRead> {
        let r = sqlx::query!(
            r#"SELECT␣id,␣firstname,␣lastname␣FROM␣person␣WHERE␣id␣=␣$1"#,
            id
        )
        .fetch_one(db)
        .await
        .expect("Failed␣to␣read␣database");

        Person::<CreateRead> {
            id: r.id,
            firstname: r.firstname,
            lastname: r.lastname,
            cap: CreateRead,
        }
    }
}

#[tokio::main]
async fn main() {
    let connection_string = "sqlite:persons.db";
    let database = SqlitePool::connect(connection_string)
        .await
        .expect("Failed␣to␣get␣database");

    let p_created =
        Person::<CreateRead>::create(&database, "Kenneth".to_string(), "Fossen".to_string()).await;

    let p_read = Person::<CreateRead>::read(&database, p_created.id).await;

    println!("{:#?}", p_created);
    println!("{:#?}", p_read);
    assert_eq!(
        p_created, p_read,
        "The␣to␣persons␣{:#?}␣and␣{:#?}␣are␣not␣equal",
        p_created, p_read
    );
}
```