UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Module 09: High Level Networking

Advanced Programming

Muhammad Yusuf Khadafi

Editor: Teaching Team (Lecturer and Teaching Assistant)

Advanced Programming

Semester Genap 2024

Fasilkom UI

Author: *Muhammad Yusuf Khadafi*
Email: m.yusuf03@cs.ui.ac.id

Table of Contents

Contents

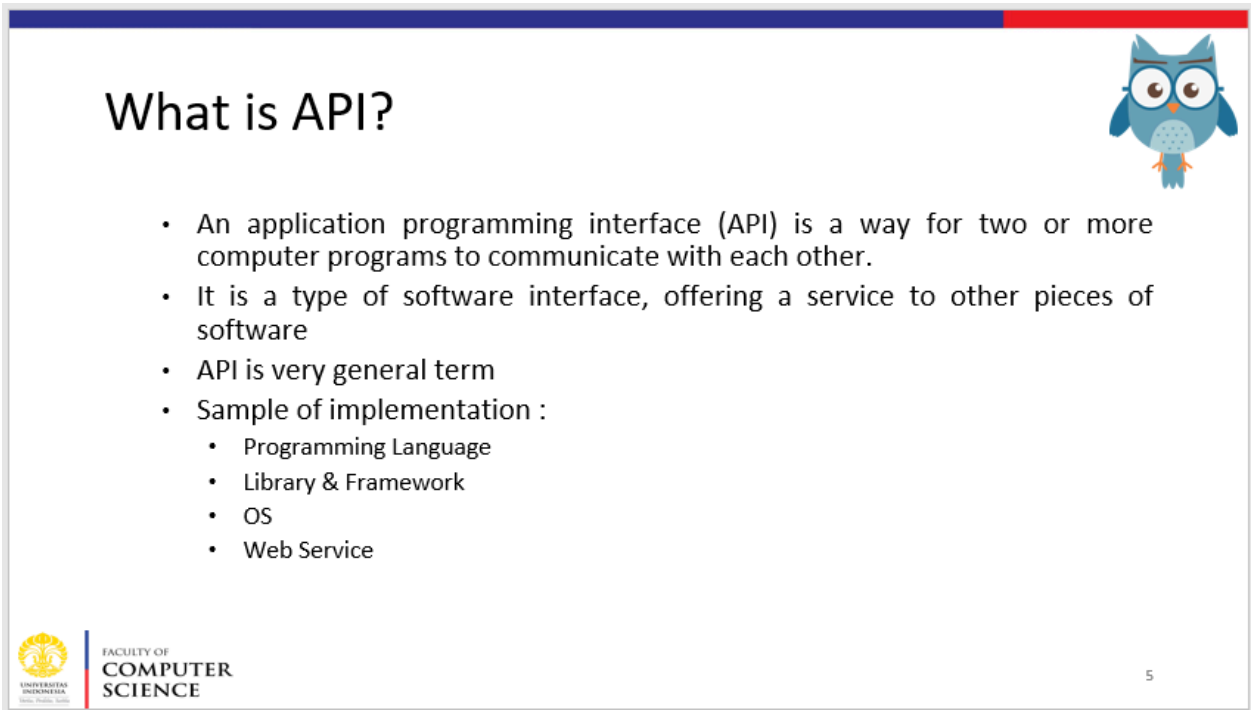
Learning Objectives

1. Fundamentals of REST API and gRPC: Students will learn the basic principles of REST API, including its key constraints and principles, and be introduced to gRPC, understanding its advantages in performance and efficiency over traditional REST APIs.
2. Implementation Techniques: Students will gain practical skills in designing and developing RESTful services and gRPC services, focusing on creating, reading, updating, and deleting data, and utilizing Protocol Buffers for gRPC.
3. Compare and Contrast REST API and gRPC: Students will critically analyze and compare the use cases, benefits, and limitations of REST APIs and gRPC. They will discuss scenarios where one might be preferred over the other based on factors such as ease of implementation, performance requirements, and compatibility with existing systems.

References

1. Masse, Mark. (2011). REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media.
2. Indrasiri, Kasun. (2020). gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes. O'Reilly Media.
3. https://medium.com/swlh/grpc-fundamental-and-concept-93414d7956df

What is REST API



The concept of an Application Programming Interface, or API, is essential in modern software development. Defined as a set of protocols and rules, an API enables communication between disparate software applications. This interface allows for seamless data and functionality exchange, promoting integration and operational efficiency among various digital services and applications.

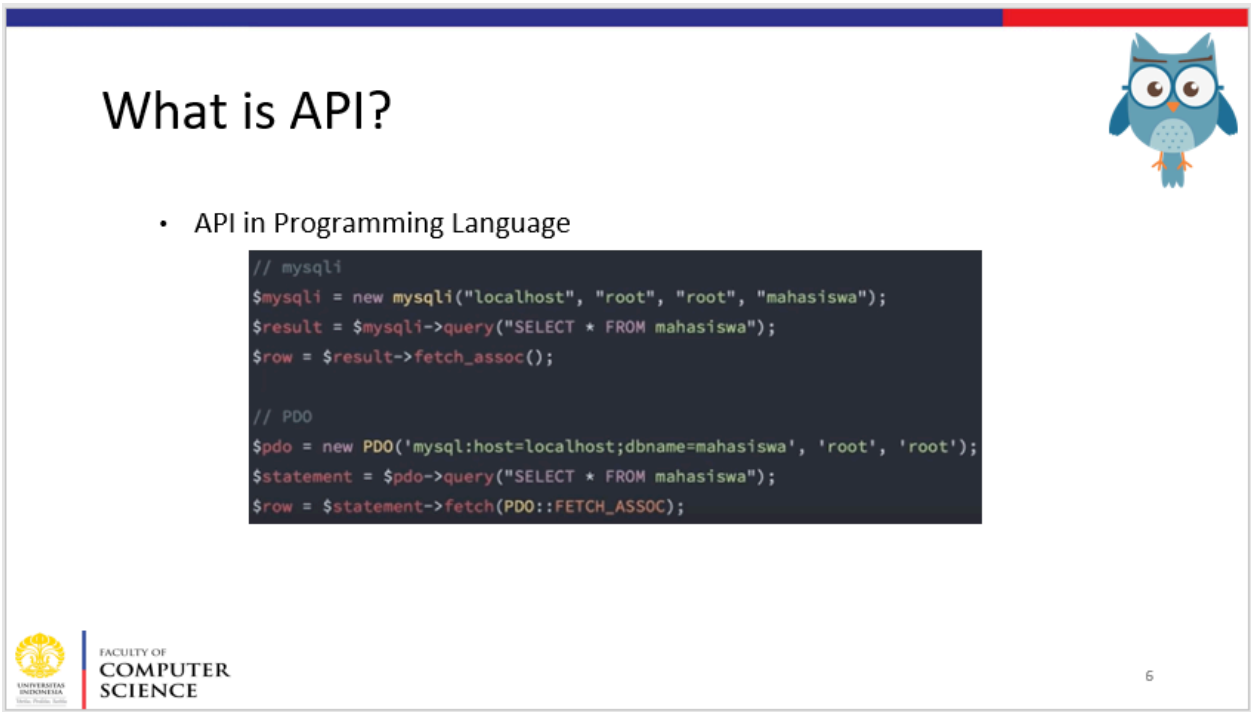
APIs function as strategic intermediaries, providing specific services to software components without exposing the underlying programming logic. Their crucial role in developing scalable and flexible digital ecosystems is significant, as APIs are instrumental across various technological domains. The key implementations of APIs include:

Programming Languages: APIs enable developers to utilize predefined functions for interactions with the operating system or other applications.

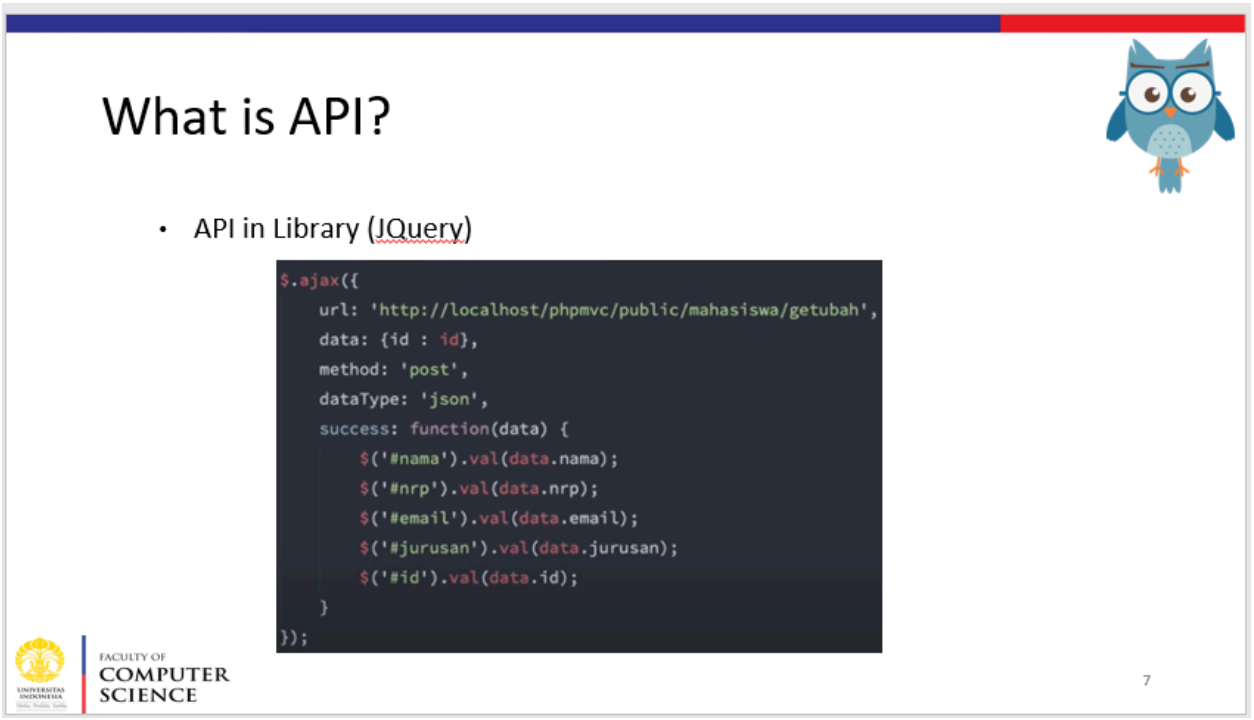
Libraries and Frameworks: These contain sets of APIs that simplify complex programming tasks, enhancing the efficiency of the development process.

Operating Systems: APIs facilitate the execution of operating system-level operations, such as file reading and writing, read contact, accessing hardware device like camera, etc.

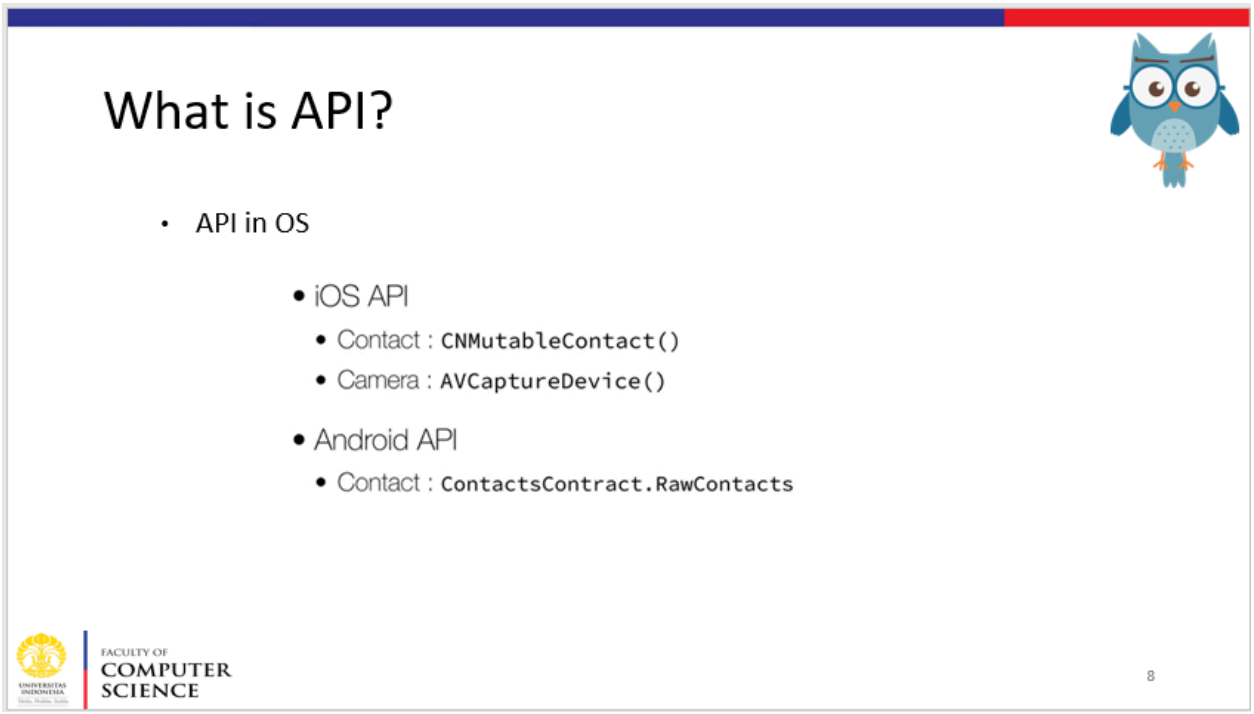
Web Services: Web APIs connect web-based applications, allowing them to exchange data over the internet, typically in JSON or XML formats.



APIs in programming languages provide a set of functions, procedures, and protocols that developers can use to build software applications. For example, the standard library in programming language provides numerous APIs for tasks like Database Connection, file I/O, network communication, and data manipulation. Additionally, programming languages often have APIs for interacting with hardware devices, such as graphics cards or sensors. These APIs abstract the underlying complexities, allowing developers to focus on application logic rather than low-level details.



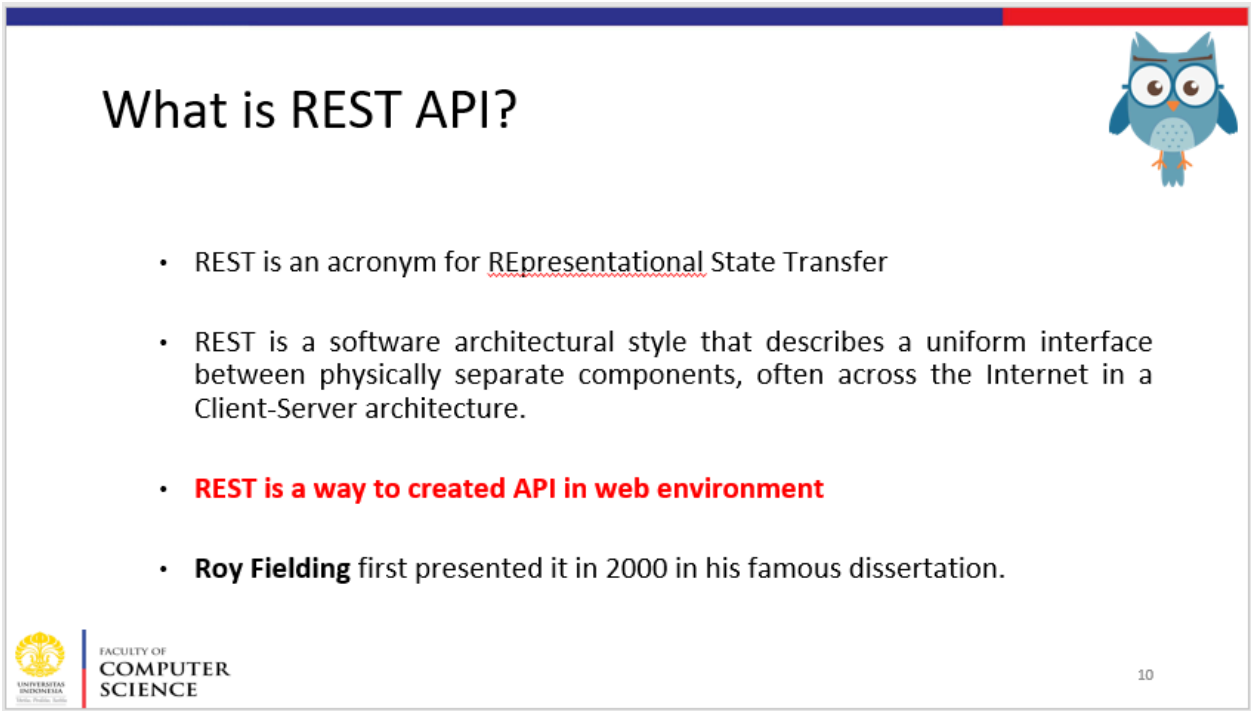
jQuery, a popular JavaScript library, provides an Ajax API that simplifies asynchronous HTTP requests and handling responses in web applications. With jQuery's Ajax API, developers can easily perform tasks like fetching data from a server without refreshing the entire webpage, updating parts of a webpage dynamically, and handling various types of HTTP requests (GET, POST, etc.). This API abstracts away the complexities of raw XMLHttpRequests and offers a more streamlined approach to handling asynchronous communication in web development.



In the realm of mobile operating systems like iOS and Android, APIs serve as the bridge between the hardware and software layers, enabling developers to create rich and interactive applications. Both iOS and Android provide comprehensive sets of APIs tailored to their respective platforms, offering functionalities for accessing device hardware (camera, GPS, sensors), user interface components (UI controls, gestures), and system services (phone contact, notifications, background tasks). These APIs empower developers to harness the full potential of the underlying operating system and create immersive mobile experiences.

SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are two widely used approaches for building web services, each with its own set of APIs. SOAP APIs typically adhere to a standardized XML-based messaging protocol and provide functionalities for defining remote procedure calls, data serialization, and error handling. In contrast, REST APIs leverage HTTP methods (GET, POST, PUT, DELETE) and represent resources as URIs, offering a lightweight and flexible approach to web service development. REST APIs are commonly used in modern web applications due to their simplicity, scalability, and compatibility with various client platforms and programming languages.

What is REST API



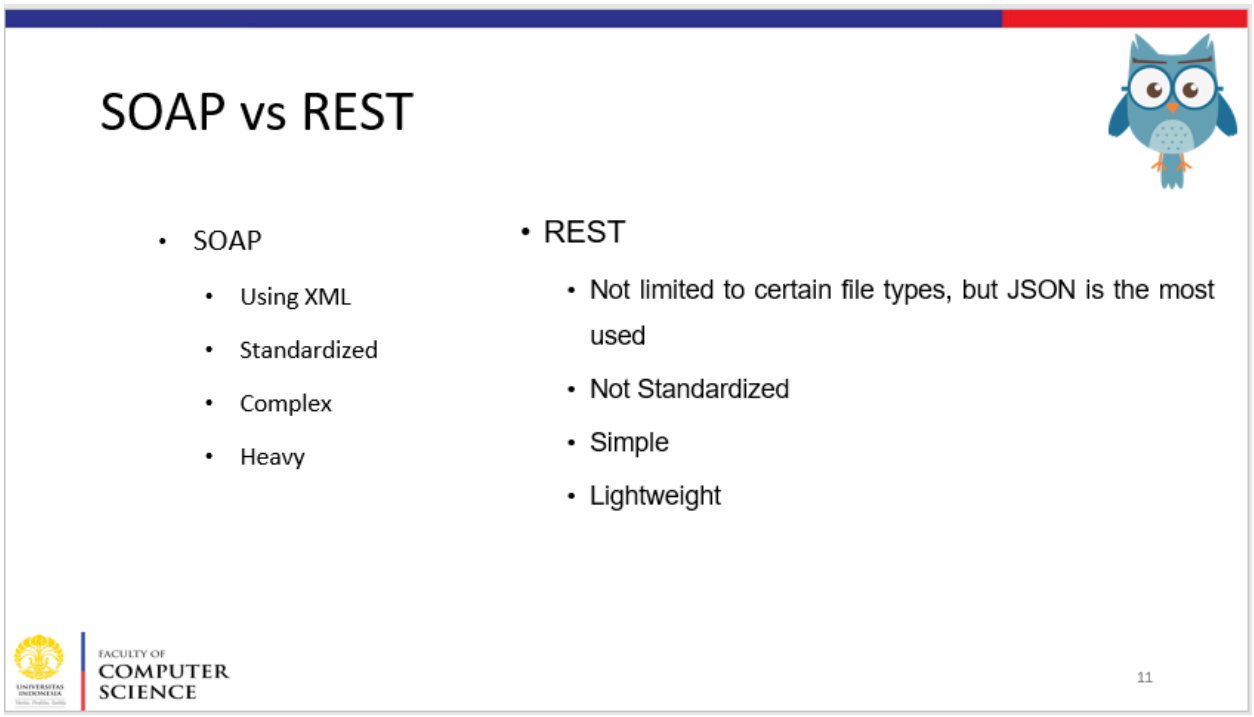
REST standing for Representational State Transfer, is a software architectural style that revolutionized the way web services are designed and implemented. It defines a set of principles for building scalable and maintainable APIs in web environments. At its core, REST emphasizes a uniform interface between client and server components, facilitating communication over the Internet in a distributed client-server architecture.

Introduced by Roy Fielding in his seminal dissertation in 2000, REST has since become a cornerstone of modern web development. Unlike traditional approaches, which often rely on tightly coupled protocols and interfaces, REST promotes loose coupling and interoperability between heterogeneous systems.

One of the key concepts of REST is resource representation, where resources are identified by URIs (Uniform Resource Identifiers) and manipulated using standard HTTP methods such as GET, POST, PUT, and DELETE. This allows for a stateless and scalable architecture, where clients can interact with server resources without the need for server-side session management.

Overall, REST offers a powerful and flexible framework for designing APIs that are well-suited to the distributed nature of the web. Its simplicity, scalability, and interoperability have made it the de facto standard for building web services, driving innovation and enabling seamless integration between diverse systems and platforms.

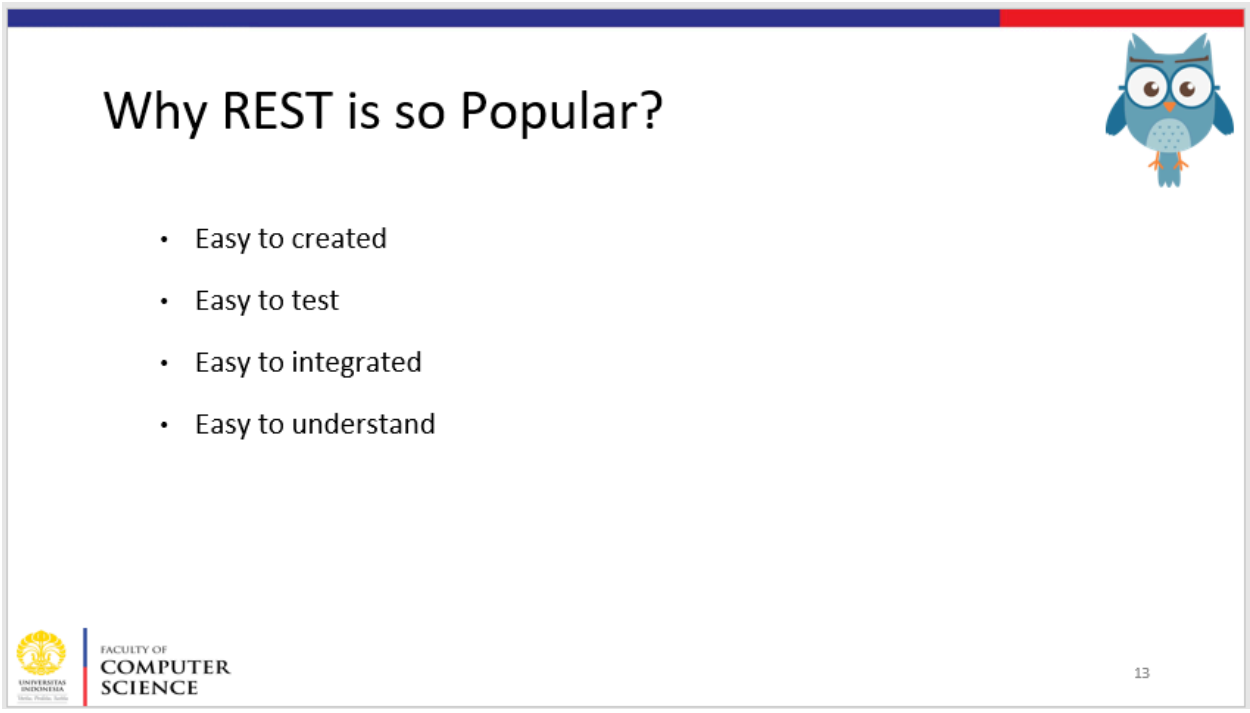
SOAP vs REST



SOAP and REST are two contrasting approaches for building web services, each with its own characteristics and advantages. SOAP, utilizing XML (eXtensible Markup Language), offers a standardized protocol for communication between distributed systems. However, SOAP tends to be complex and heavyweight due to its extensive specifications and standardized messaging formats.

On the other hand, REST, which stands for Representational State Transfer, is more flexible in terms of data formats, often using JSON (JavaScript Object Notation). Unlike SOAP, REST is not standardized, allowing for greater freedom in designing APIs. REST is celebrated for its simplicity and lightweight nature, making it easier to develop and maintain APIs compared to SOAP. While SOAP is ideal for enterprise-level applications requiring strict standards compliance, REST is preferred for its simplicity and adaptability to modern web development practices.

Why REST is so Popular



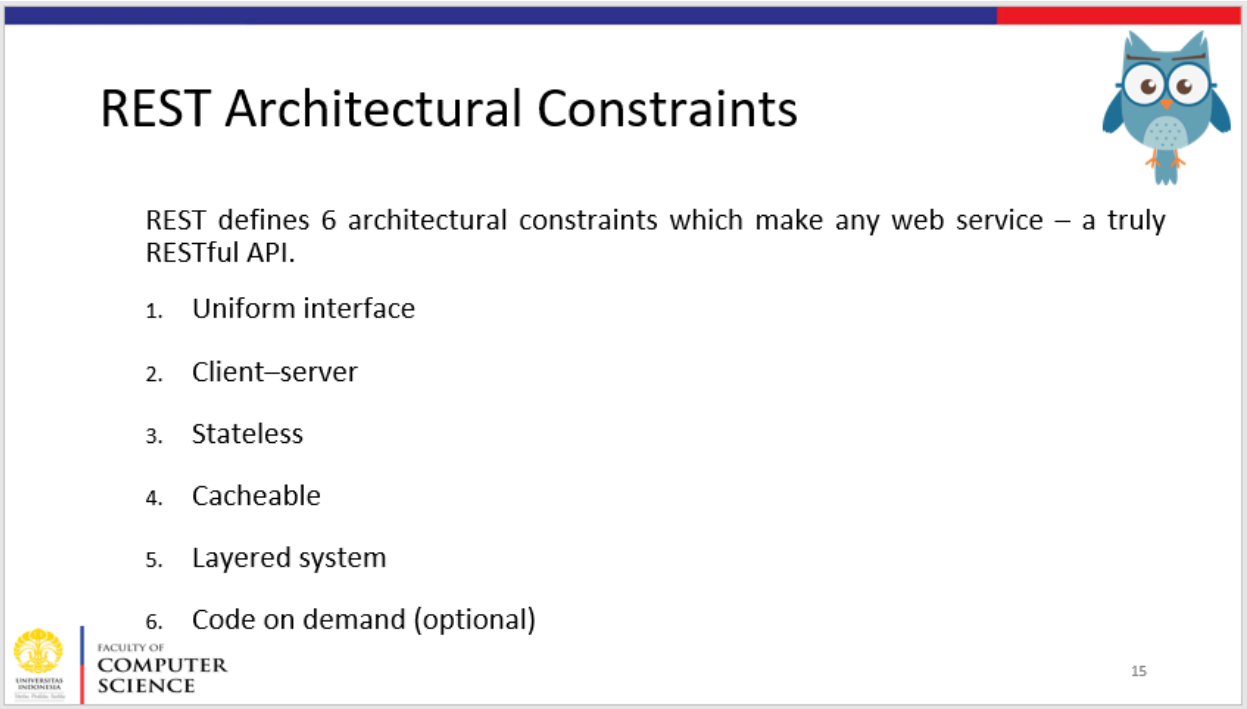
REST has gained immense popularity in the realm of web development due to several key reasons. Firstly, it is remarkably easy to create RESTful APIs, thanks to its straightforward architectural principles and HTTP-based approach. Developers can quickly design and implement APIs using standard HTTP methods like GET, POST, PUT, and DELETE.

Moreover, REST APIs are easy to test, as they leverage standard web protocols and tools for validation and debugging. This simplifies the testing process and ensures the reliability and stability of the API endpoints.

Integration with existing systems and platforms is seamless with REST, thanks to its lightweight and flexible nature. RESTful APIs can be easily integrated into diverse environments, allowing for interoperability between different systems and services.

Furthermore, REST's simplicity and clarity make it easy to understand for both developers and consumers of APIs. Its intuitive design principles, such as resource representation and stateless communication, contribute to its widespread adoption and popularity in the development community. Overall, REST's ease of creation, testing, integration, and understanding have solidified its position as the preferred choice for building web APIs.

REST Architectural Constraints

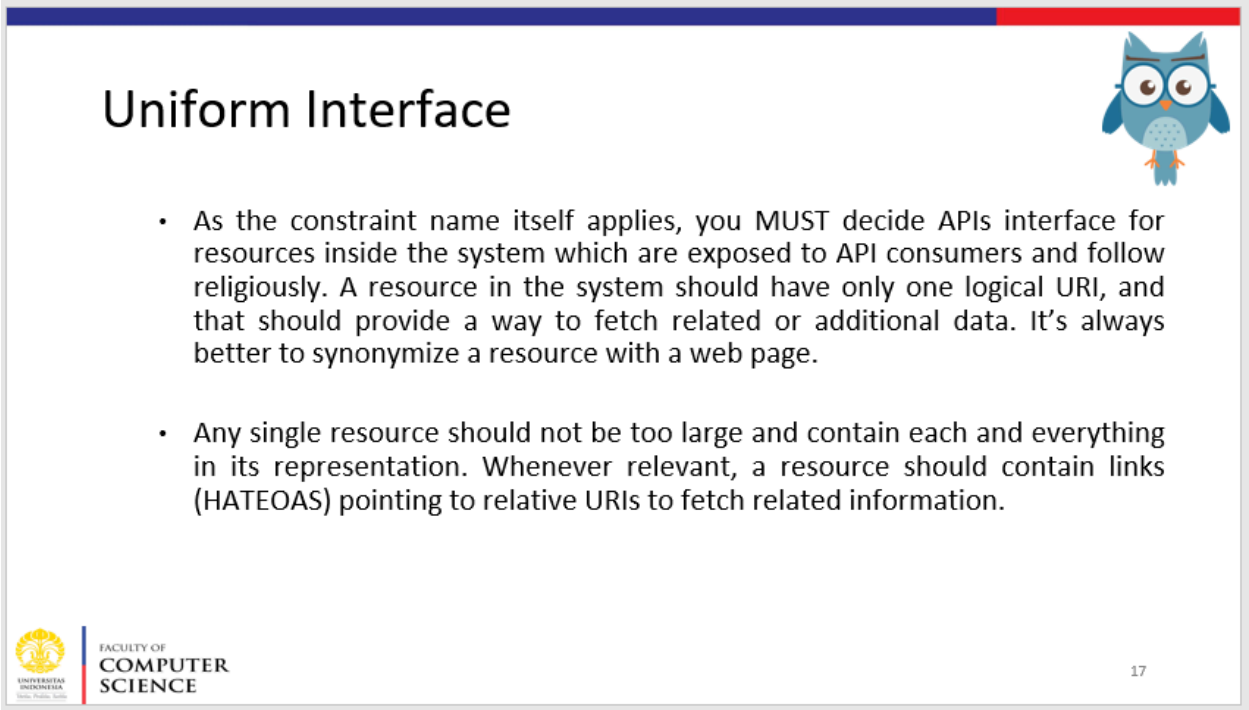


REST, or Representational State Transfer, sets forth six architectural constraints that characterize a truly RESTful API, distinguishing it from other approaches.

1. Uniform interface: This constraint ensures consistency across interactions by utilizing standard methods and resource representations, promoting simplicity and predictability in API design.
2. Client-server: Separating the client from the server facilitates independent evolution of both components, enabling scalability and parallel development.
3. Stateless: Stateless communication means each request from the client to the server contains all necessary information, eliminating the need for server-side session management and improving scalability and reliability.
4. Cacheable: Responses from the server can be cached to enhance performance and reduce latency, promoting efficiency in distributed systems.
5. Layered system: Intermediaries such as proxies and gateways can be inserted between clients and servers without impacting the overall architecture, enhancing scalability, security, and flexibility.
6. Code on demand (optional): This optional constraint allows for the transfer of executable code, such as JavaScript, which can enhance client functionality dynamically

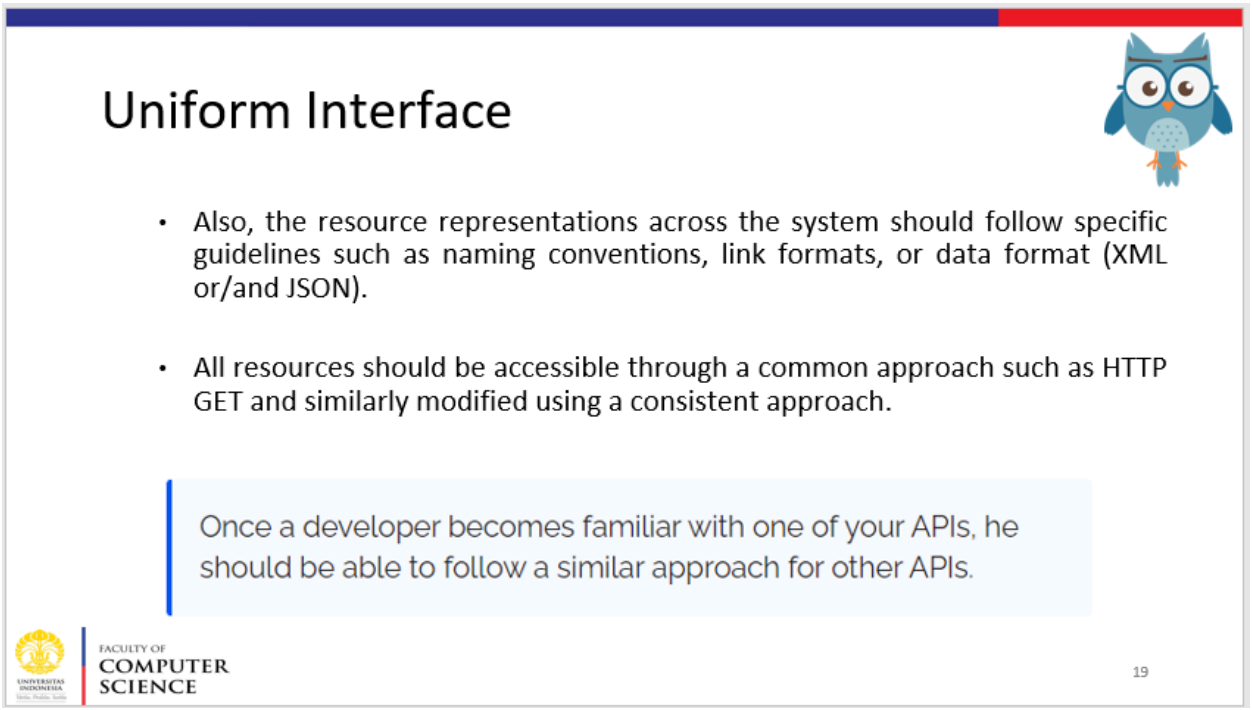
By adhering to these architectural constraints, developers can create RESTful APIs that are scalable, maintainable, and interoperable across diverse environments.

Uniform Interface



The Uniform Interface constraint in REST API design emphasizes the importance of consistency and clarity in defining the interfaces for resources within the system. Each resource should have a single logical URI, ensuring simplicity and predictability for API consumers. This uniformity simplifies resource identification and retrieval, akin to navigating web pages via URLs.

Furthermore, resources should be designed to be self-descriptive, providing only relevant information in their representations. It's crucial to avoid bloating resources with unnecessary data, promoting efficiency and maintainability. Instead, resources should include links (HATEOAS - Hypermedia as the Engine of Application State) pointing to related URIs, enabling clients to discover and navigate the API dynamically.

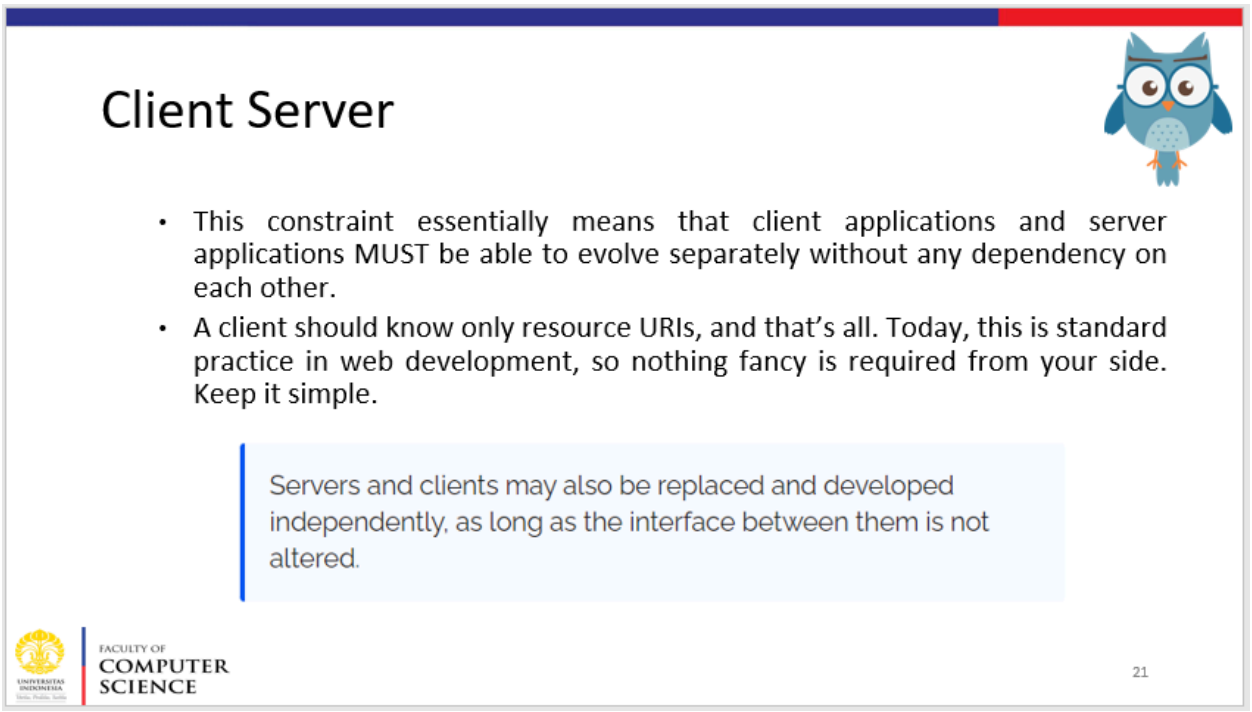


The Uniform Interface constraint in REST API design encompasses various aspects to ensure consistency and ease of use across the system. Resource representations should adhere to established guidelines for naming conventions, link formats, and data formats (such as XML or JSON), promoting uniformity and clarity in API usage.

Additionally, all resources should be accessible through standardized HTTP methods like GET, enabling predictable and intuitive interactions. Similarly, modifications to resources should follow a consistent approach, simplifying development and reducing cognitive load for developers.

By enforcing a uniform interface, RESTful APIs enable developers to quickly understand and navigate different APIs within the system. Once familiar with one API, developers can easily apply the same principles and practices to interact with others, fostering efficiency and reducing the learning curve. This consistency enhances developer productivity and promotes a seamless experience across the API ecosystem.

Client Server

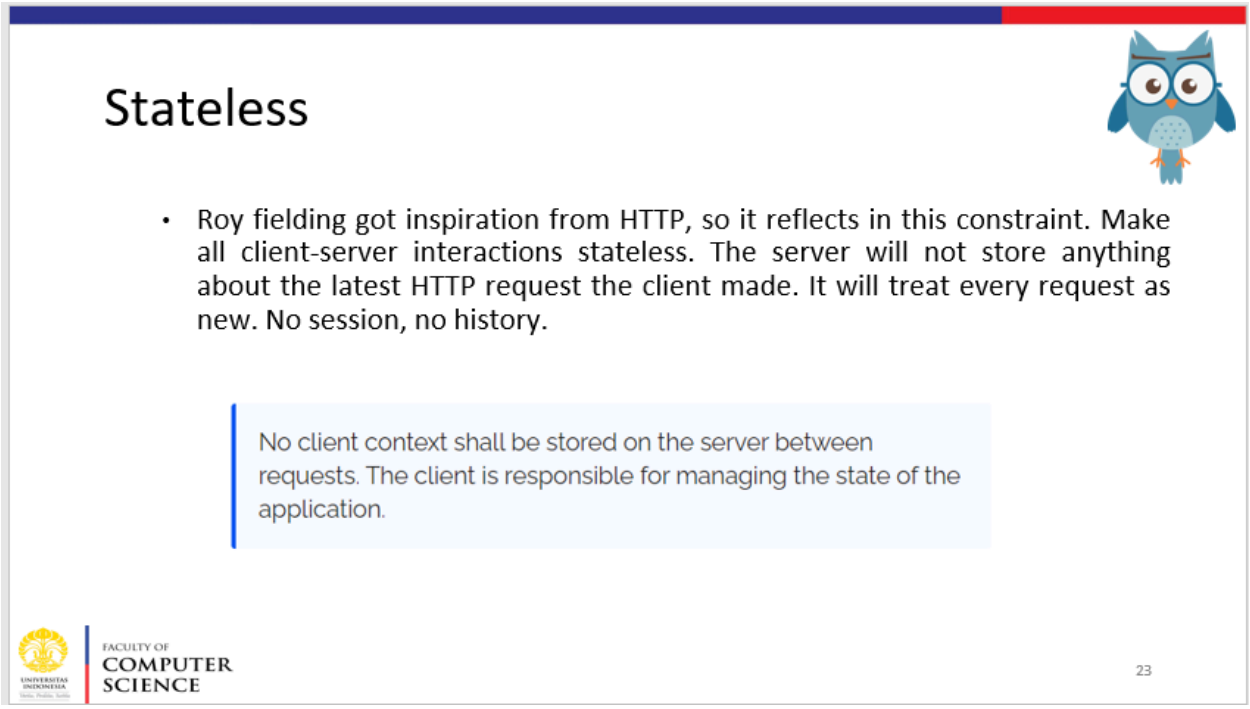


The Client-Server constraint in REST API design underscores the importance of decoupling client and server components to enable independent evolution and development. It mandates that client applications and server applications operate independently, with no interdependency between them. Clients should only need to know resource URIs, simplifying interactions and promoting flexibility.

In modern web development, adhering to this constraint is standard practice, requiring simplicity rather than complexity. Both servers and clients can be replaced or upgraded independently, as long as the interface between them remains unchanged. This fosters agility and scalability in system development, allowing components to be modified or updated without affecting other parts of the system.

By following the Client-Server constraint, RESTful APIs facilitate modular and scalable architectures, enabling seamless evolution and innovation in both client and server applications while ensuring interoperability and reliability in the overall system.

Stateless

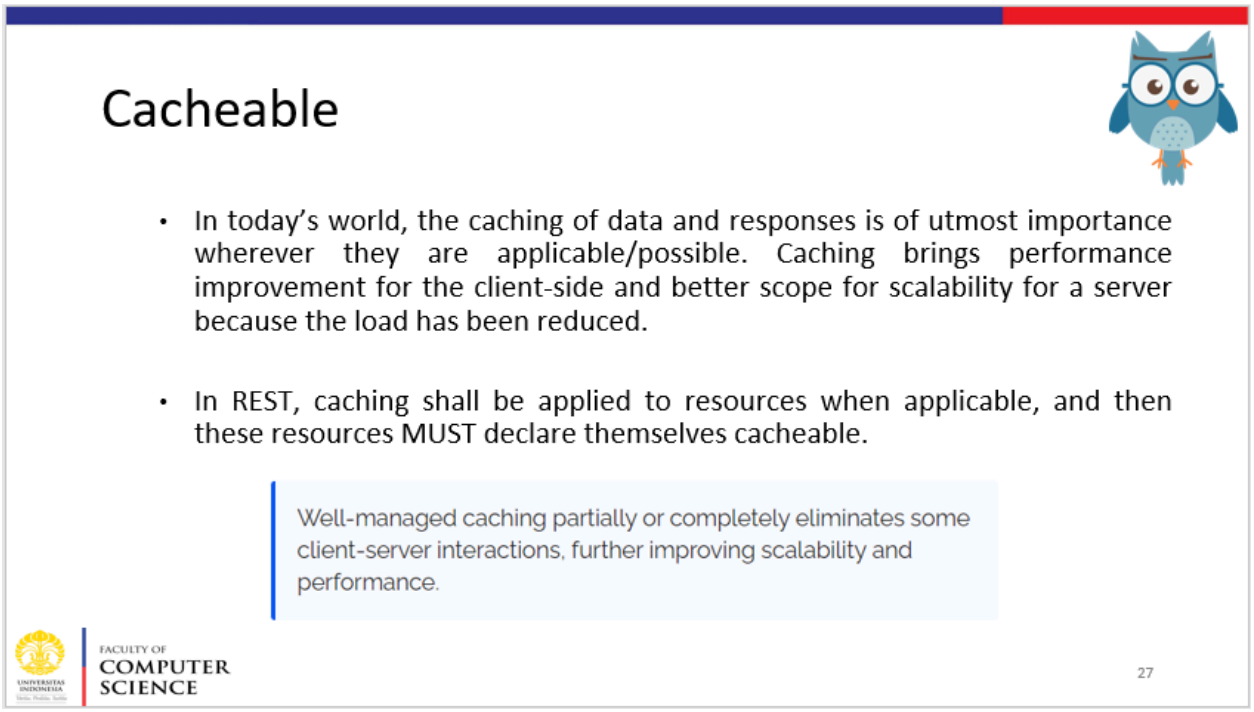


The Stateless constraint in REST API design, inspired by HTTP principles as outlined by Roy Fielding, mandates that all client-server interactions be devoid of server-side state retention. In essence, each request from the client to the server must contain all necessary information for the server to fulfill the request without relying on any prior context or state.

This means that servers should not store any information about previous client requests, resulting in a clean slate for each interaction. Consequently, there are no sessions or historical data retained on the server, simplifying server-side management and enhancing scalability. Instead, clients are responsible for managing the state of their applications, ensuring autonomy and flexibility.

By adhering to the Stateless constraint, RESTful APIs promote simplicity, scalability, and reliability in distributed systems. They facilitate easier load balancing and fault tolerance, as servers can handle requests independently without needing to manage client state, ultimately leading to more robust and efficient systems.

Cacheable



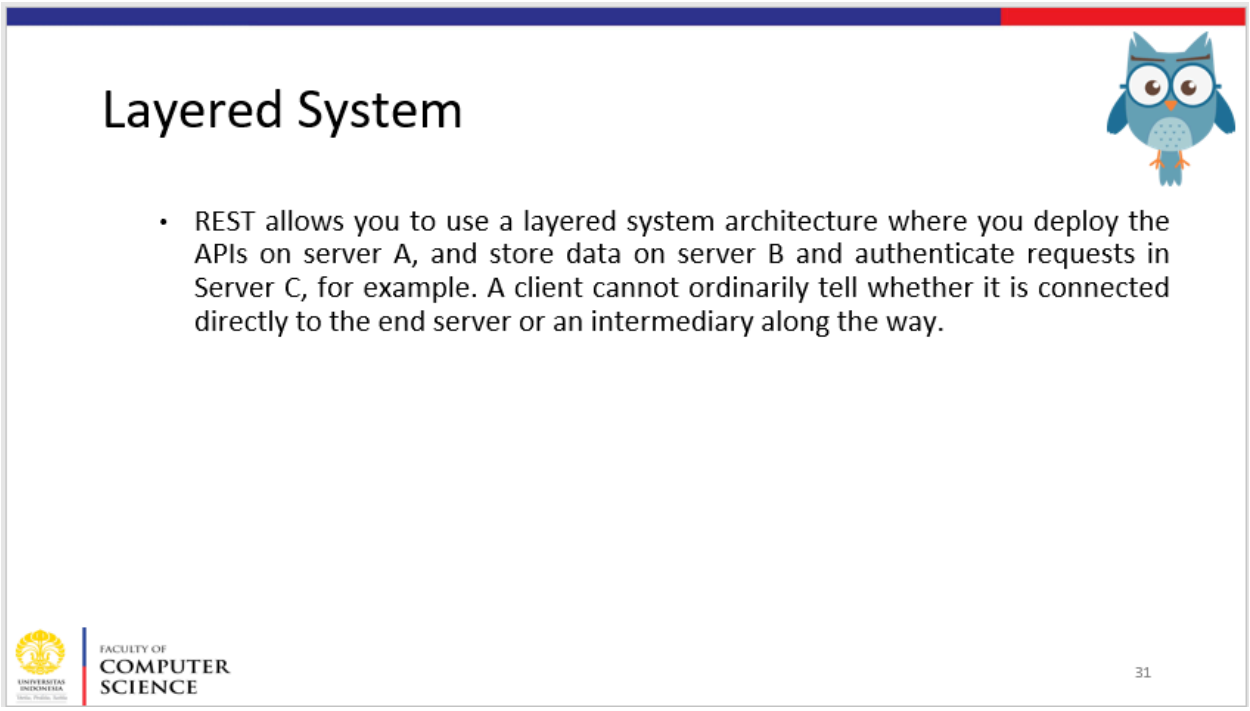
The Cacheable constraint in REST API design emphasizes the significance of caching data and responses to enhance performance and scalability in modern web applications. Caching plays a crucial role in reducing latency and improving responsiveness for clients by storing frequently accessed resources locally.

In RESTful architectures, resources should be designed to be cacheable whenever applicable. This means that resources must declare themselves as cacheable, allowing clients and intermediary servers to cache them according to specified caching policies.

Effective caching can significantly reduce the need for repeated client-server interactions, as cached resources can be served directly from the cache without requiring requests to the server. This not only improves the performance and responsiveness of client applications but also reduces the load on the server, enabling better scalability and resource utilization.

By adhering to the Cacheable constraint, RESTful APIs can achieve optimized performance and scalability, resulting in a more efficient and responsive system overall.

Layered System



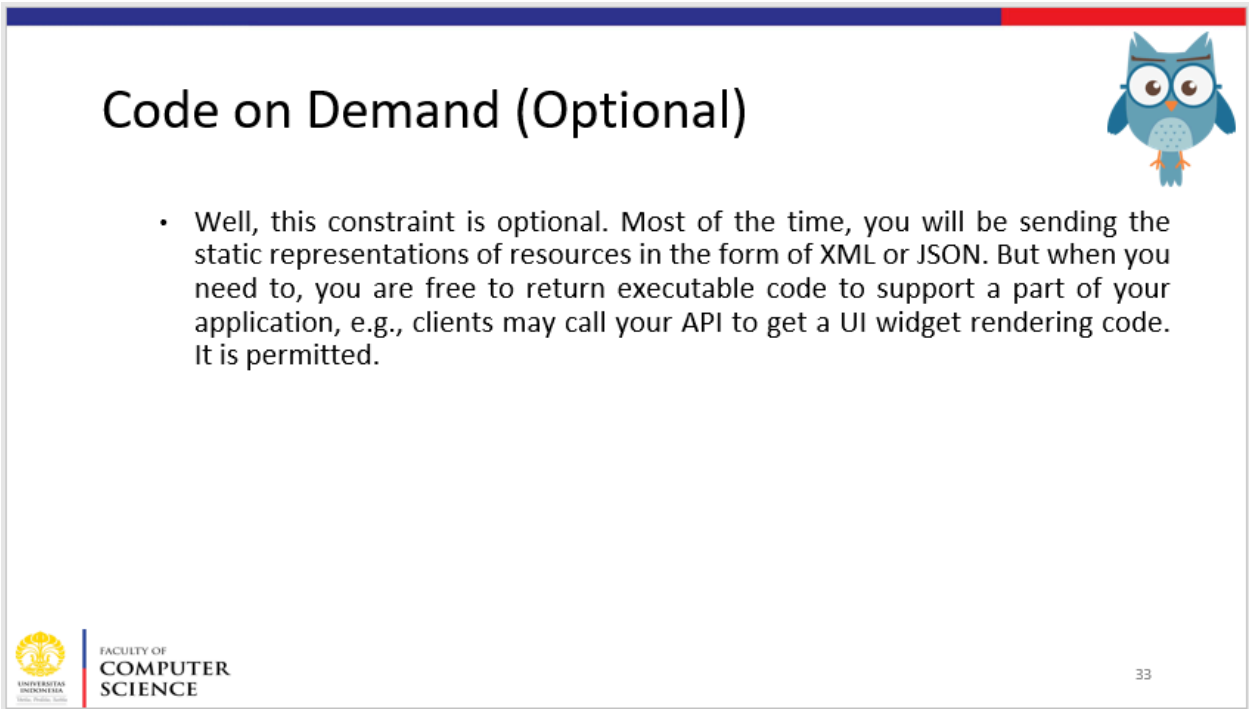
The Layered System constraint in REST API design enables the deployment of APIs within a layered architecture, offering flexibility and scalability in system design. With this constraint, different components of the system can be distributed across multiple layers or tiers, each responsible for specific functionalities.

For instance, in a layered system, APIs may be deployed on Server A, while data storage resides on Server B, and request authentication is handled by Server C. Clients interact with the system without necessarily knowing the underlying architecture or the specific servers involved. This separation of concerns allows for easier maintenance, scalability, and security enhancements, as each layer can be managed and updated independently.

Additionally, the layered architecture enables the insertion of intermediaries such as proxies or gateways between clients and servers. This provides opportunities for load balancing, caching, security enforcement, and other optimizations without impacting the overall system architecture or client-server interactions.

Overall, the Layered System constraint in RESTful APIs facilitates modular and scalable architectures, promoting flexibility and interoperability while enhancing system reliability and performance.

Code on Demand

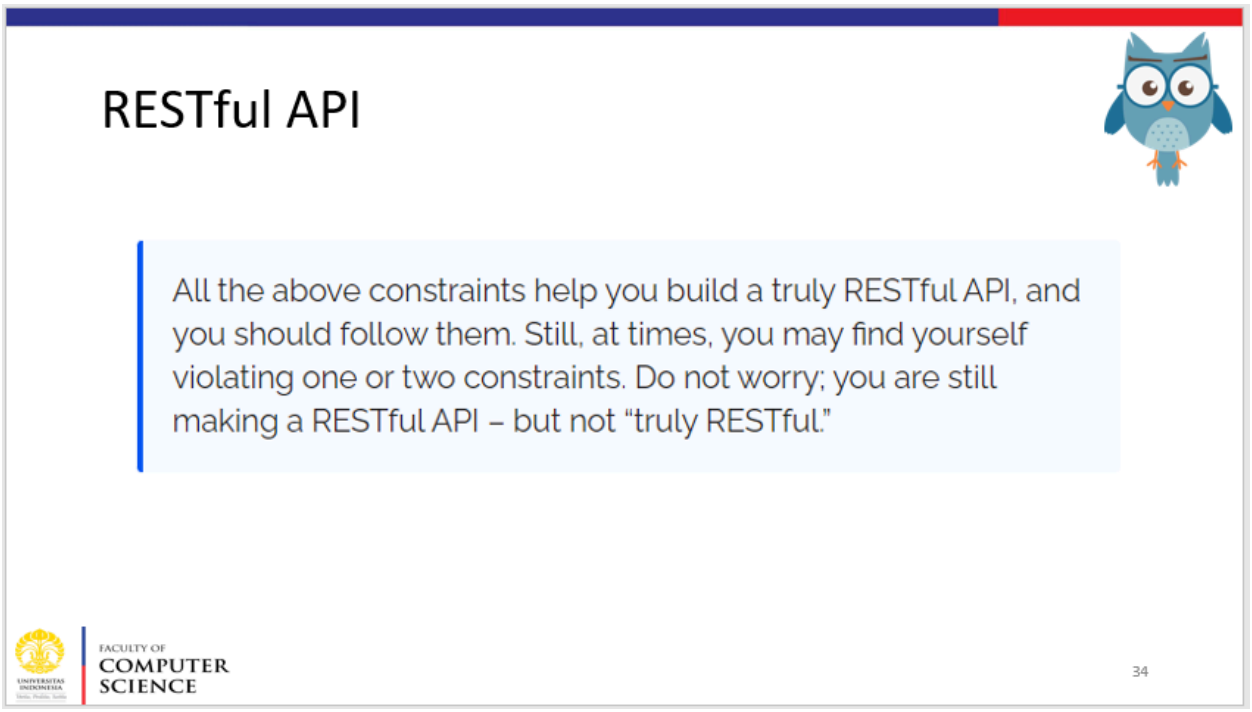


The "Code on Demand" constraint in REST API design offers flexibility by allowing servers to transmit executable code to clients upon request, although it's optional and not commonly used. While REST typically involves exchanging static representations of resources, such as XML or JSON, this constraint permits servers to provide executable code snippets to clients when necessary.

For example, clients might request UI widget rendering code or other dynamic functionalities from the server. By enabling servers to transmit executable code, clients can enhance their capabilities without having to implement all functionalities locally.

Although not widely utilized due to security concerns and the complexity it introduces, the "Code on Demand" constraint provides an additional avenue for extending the functionality of client applications dynamically. It grants developers the freedom to leverage server-side capabilities to support specific parts of their applications, further enriching the client-server interaction in RESTful architectures.

RESTful API

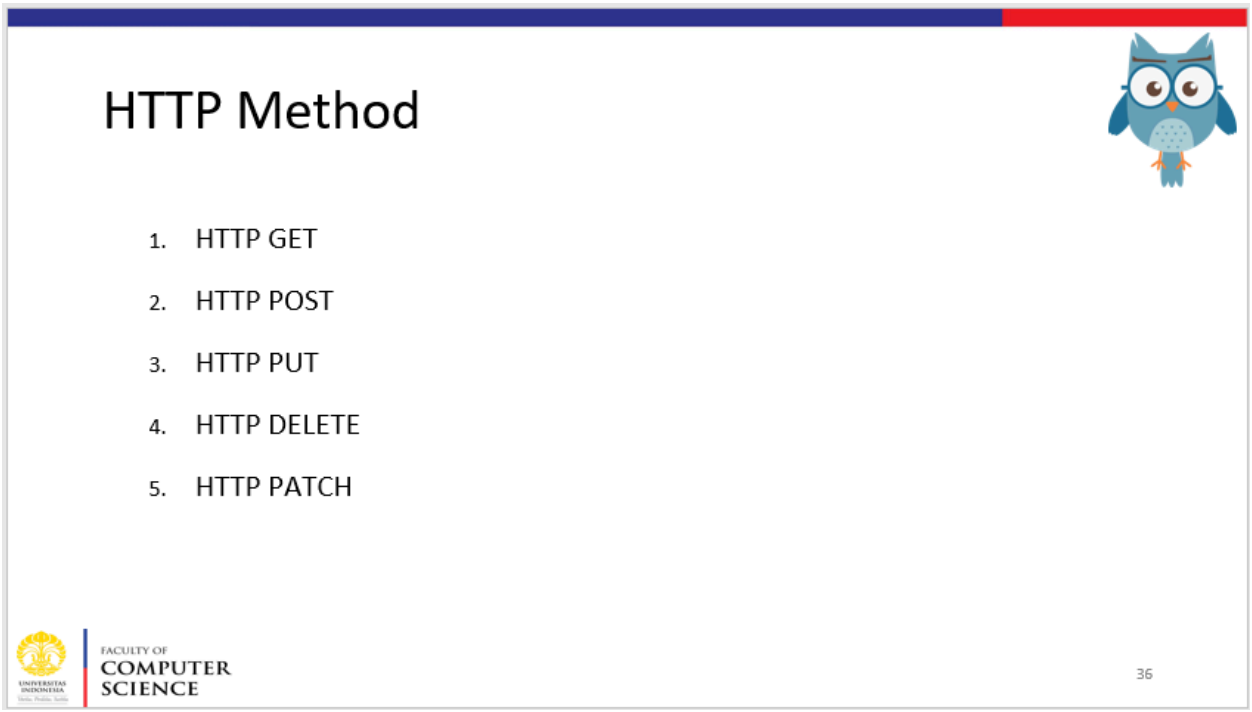


RESTful APIs adhere to a set of architectural constraints, as discussed previously, to ensure interoperability, scalability, and simplicity. While striving to follow all constraints is ideal, it's not uncommon to deviate from them occasionally. Despite this, even if one or two constraints are not fully met, the API can still be considered RESTful, albeit not entirely adherent.

Understanding that absolute adherence to all constraints may not always be feasible, developers should prioritize the constraints that align most closely with their project's goals and requirements. Flexibility within the REST architectural style permits adjustments to accommodate specific use cases without sacrificing the essence of RESTfulness.

By striving to uphold the core principles of REST, developers can still derive many benefits, such as improved scalability, maintainability, and interoperability, even if the API isn't strictly adherent to all constraints. The key is to strike a balance between following the constraints and meeting the practical needs of the application or system being developed.

HTTP Methods

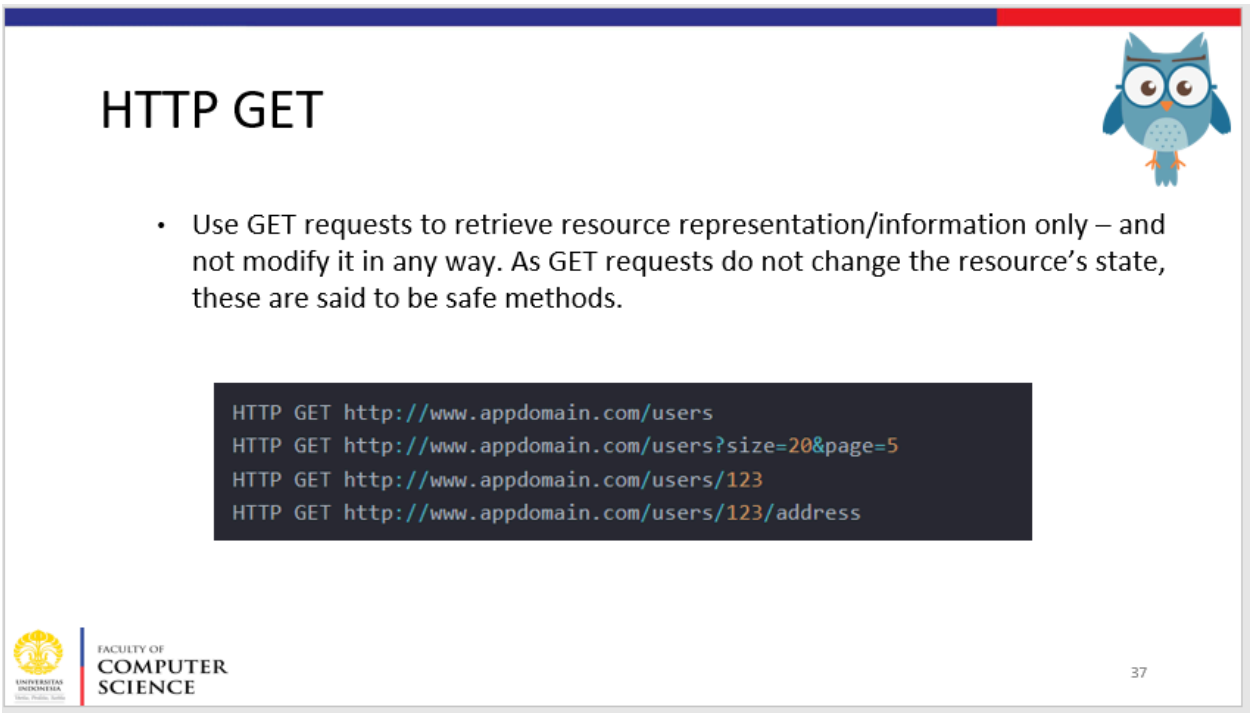


HTTP methods play a pivotal role in RESTful API design, defining how clients interact with server resources.

- HTTP GET: Retrieves data from a server.
- HTTP POST: Submits data to be processed by the server.
- HTTP PUT: Updates data on the server, typically replacing existing data.
- HTTP DELETE: Removes data from the server.
- HTTP PATCH: Partially updates data on the server, making modifications to specific fields.

Each method serves a distinct purpose, enabling versatile and efficient interactions between clients and servers in RESTful architectures.

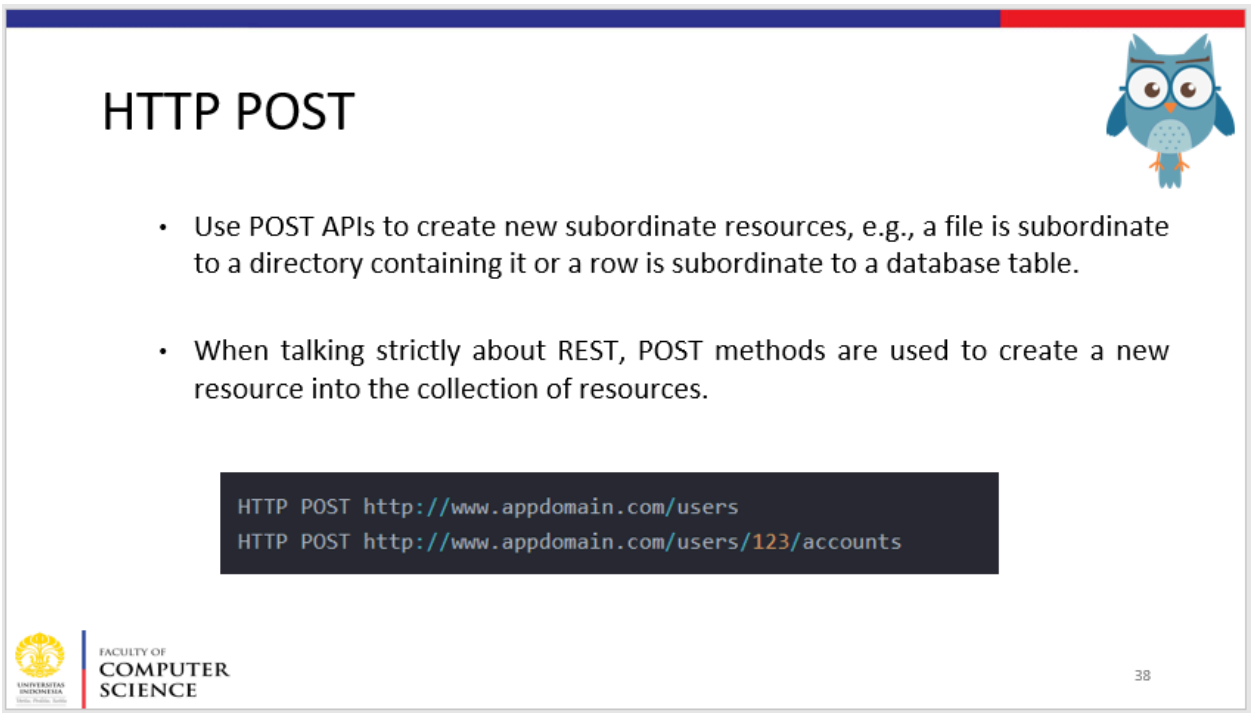
HTTP GET



HTTP GET requests are fundamental in RESTful API design, employed to retrieve resource representations or information from a server without altering its state. GET requests are considered safe methods as they do not modify the resource's state. Instead, they solely retrieve data, making them suitable for operations that do not involve updates or modifications.

GET requests are commonly used for fetching data such as retrieving user profiles, fetching product information, or accessing articles. They are straightforward and idempotent, meaning multiple identical GET requests will yield the same result. This simplicity and predictability make HTTP GET requests ideal for scenarios where clients only need to access information from the server without altering any resource state.

HTTP POST

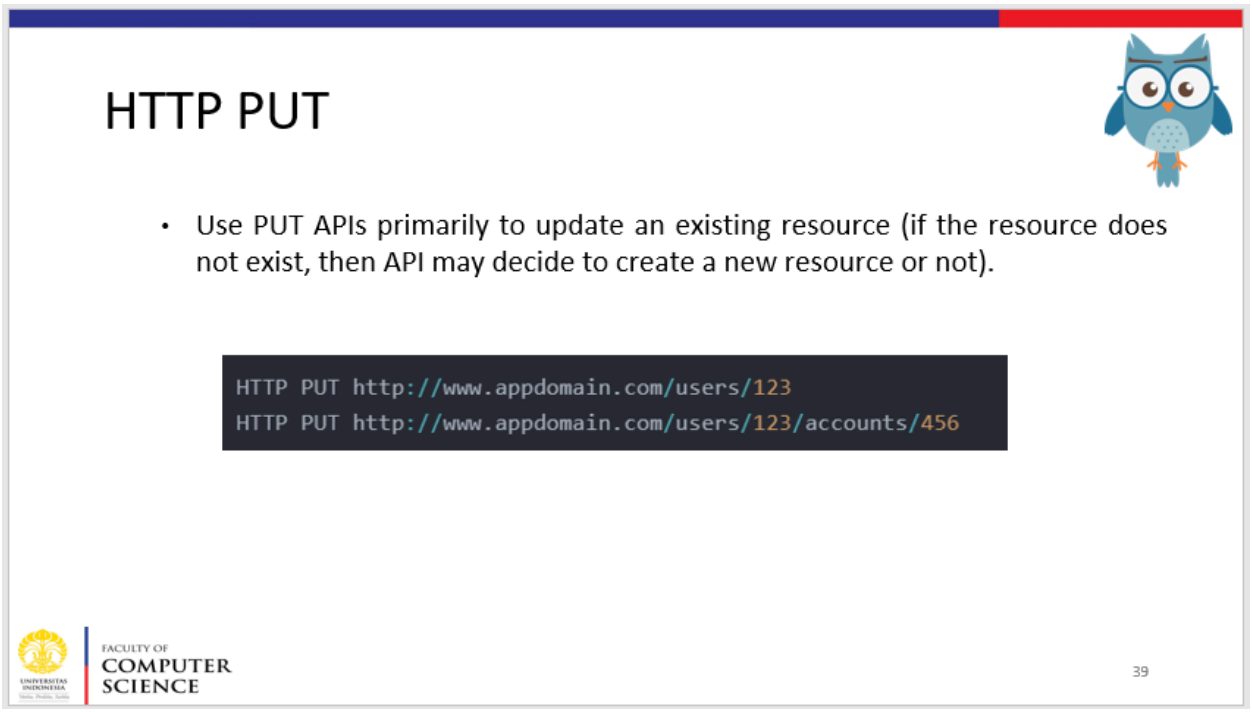


HTTP POST requests are crucial in RESTful API design, serving to create new subordinate resources within a server. Subordinate resources are entities that are logically related to a parent resource, such as a file within a directory or a row within a database table.

In the context of REST, POST methods are primarily utilized to add a new resource to a collection of resources. For instance, when a user submits a form to create a new blog post or adds a new product to an online store, a POST request is sent to the server.

By leveraging POST requests, clients can initiate the creation of new resources, enabling dynamic expansion and management of collections within the server. This functionality enhances the flexibility and scalability of RESTful APIs, facilitating the addition of new data to the system.

HTTP PUT

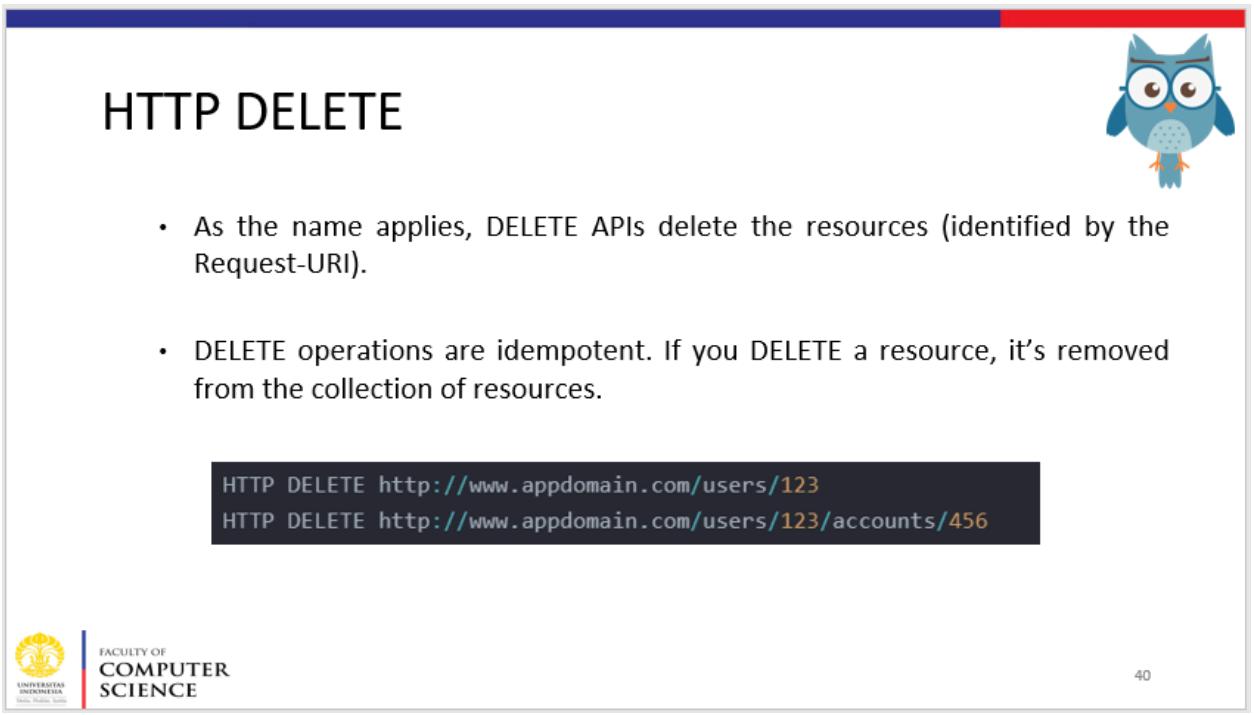


HTTP PUT requests are integral in RESTful API design, primarily utilized for updating existing resources on a server. When a client sends a PUT request, it indicates the intention to modify an existing resource. If the specified resource does not exist, the server may choose to create a new resource based on the provided data, although this behavior can vary depending on the API implementation.

PUT requests are idempotent, meaning that making multiple identical PUT requests will have the same effect as making a single request. This characteristic ensures predictability and consistency in resource updates.

Typically, PUT requests are used for operations such as updating user profiles, modifying existing articles, or changing product information in an e-commerce system. By leveraging PUT requests, clients can efficiently update resource representations on the server, facilitating dynamic data management and synchronization within the application.

HTTP DELETE



HTTP DELETE requests are vital in RESTful API design, serving to remove resources identified by the Request-URI from the server. When a client sends a DELETE request, it signals the intention to delete the specified resource.

DELETE operations are idempotent, meaning that multiple identical DELETE requests will result in the same outcome as a single request. If a resource is successfully deleted, it is removed from the collection of resources on the server.

DELETE requests are commonly used for operations such as deleting user accounts, removing articles or posts from a database, or eliminating files from a storage system. By leveraging DELETE requests, clients can efficiently manage resources on the server, facilitating data cleanup and maintenance within the application.

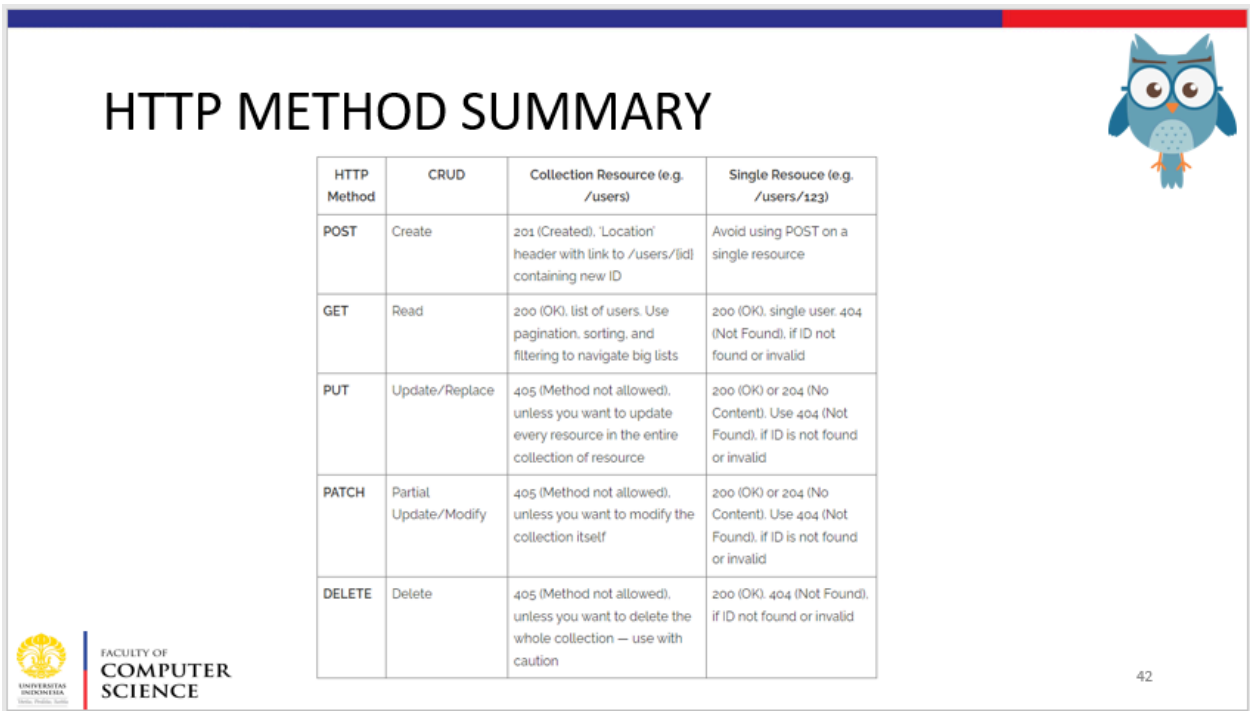
HTTP PATCH

HTTP PATCH requests are essential in RESTful API design, utilized to make partial updates to existing resources on a server. Unlike PUT requests, which are used to replace a resource in its entirety, PATCH requests are suitable for making specific modifications to a resource without affecting its entire representation.

PATCH requests are ideal when only certain fields or attributes of a resource need to be updated, providing a more efficient and targeted approach to resource modification. By sending a PATCH request with the desired changes, clients can update specific aspects of a resource without having to resend the entire resource representation.

Common use cases for PATCH requests include updating individual fields of a user profile, modifying specific properties of an article, or adjusting certain attributes of a product. Leveraging PATCH requests allows for more granular control over resource updates, enhancing flexibility and efficiency in RESTful API interactions.

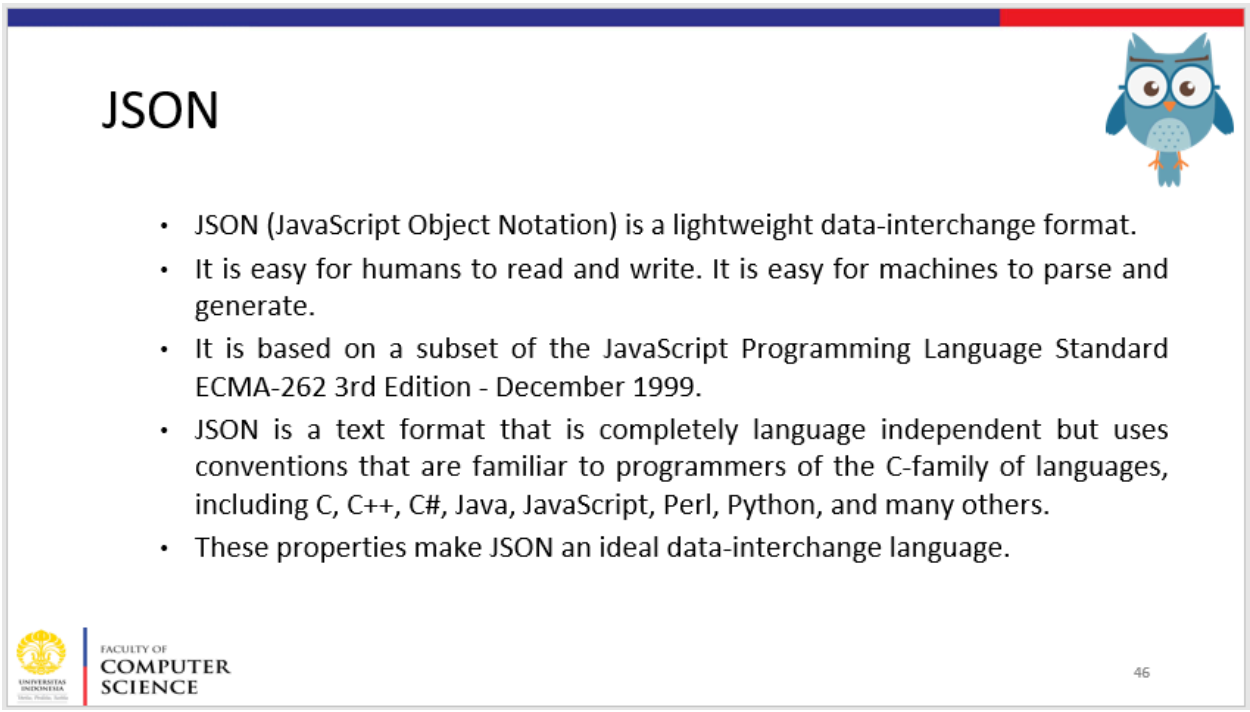
HTTP METHOD SUMMARY



- POST (Create):
    - Used for creating new resources in a collection.
    - Returns 201 Created status code along with the 'Location' header containing the link to the newly created resource (e.g., /users/{id}).
    - Avoid using POST on single resource endpoints as it may lead to confusion or inconsistency in API design.
- GET (Read):
    - Retrieves resource representations from the server.
    - For collection endpoints (e.g., /users), returns a list of resources.
    - For individual resource endpoints (e.g., /users/123), returns the representation of a single resource.
    - Pagination, sorting, and filtering mechanisms help navigate through large datasets efficiently.
- PUT (Update/Replace):
    - Updates or replaces existing resources with the provided data.
    - Typically used on single resource endpoints to update resource representations.
    - Returns 200 OK or 204 No Content upon successful update.
    - Avoid using PUT on collection endpoints unless the intention is to replace every resource in the collection.
- PATCH (Partial Update/Modify):

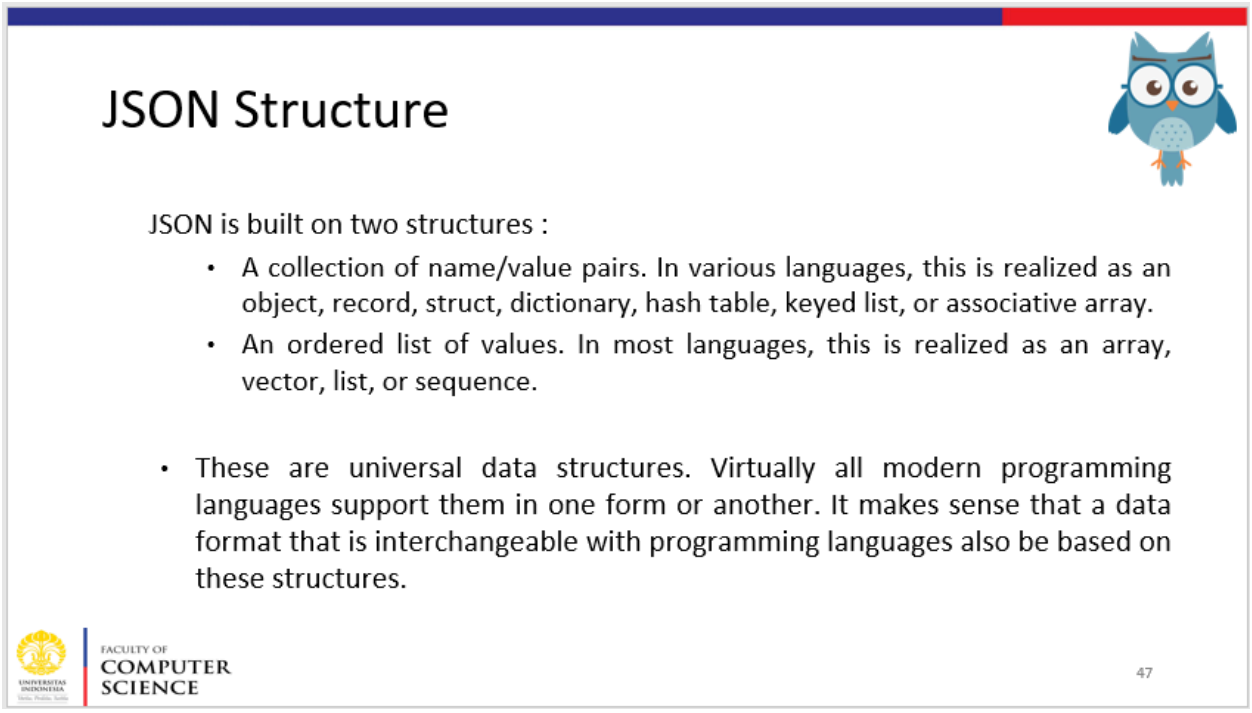
    - Partially updates existing resources with the provided data.
    - Used on single resource endpoints to modify specific fields or attributes of a resource.
    - Returns 200 OK or 204 No Content upon successful update.
    - Avoid using PATCH on collection endpoints unless modifying the collection itself.
- DELETE (Delete):
    - Removes resources from the server.
    - Use with caution, especially on collection endpoints, as it may result in the deletion of multiple resources.
    - Returns 200 OK upon successful deletion.
    - Returns 404 Not Found if the specified resource ID is invalid or not found.

JSON



JSON (JavaScript Object Notation) serves as a lightweight data-interchange format, valued for its readability and simplicity. Designed for both human and machine comprehension, JSON facilitates parsing and generation with ease. It's rooted in a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition from December 1999, ensuring compatibility with JavaScript environments. Despite its JavaScript origins, JSON is language-independent, adopting conventions familiar to programmers of C-family languages like C, C++, C#, Java, JavaScript, Perl, and Python. Its text-based format allows for seamless integration across various platforms and systems. JSON's versatility and familiarity make it an optimal choice for data interchange, enabling efficient communication between different programming languages and environments. This widespread adoption has solidified JSON as a cornerstone of modern web development and API integration, facilitating interoperability and streamlined data exchange in software development.

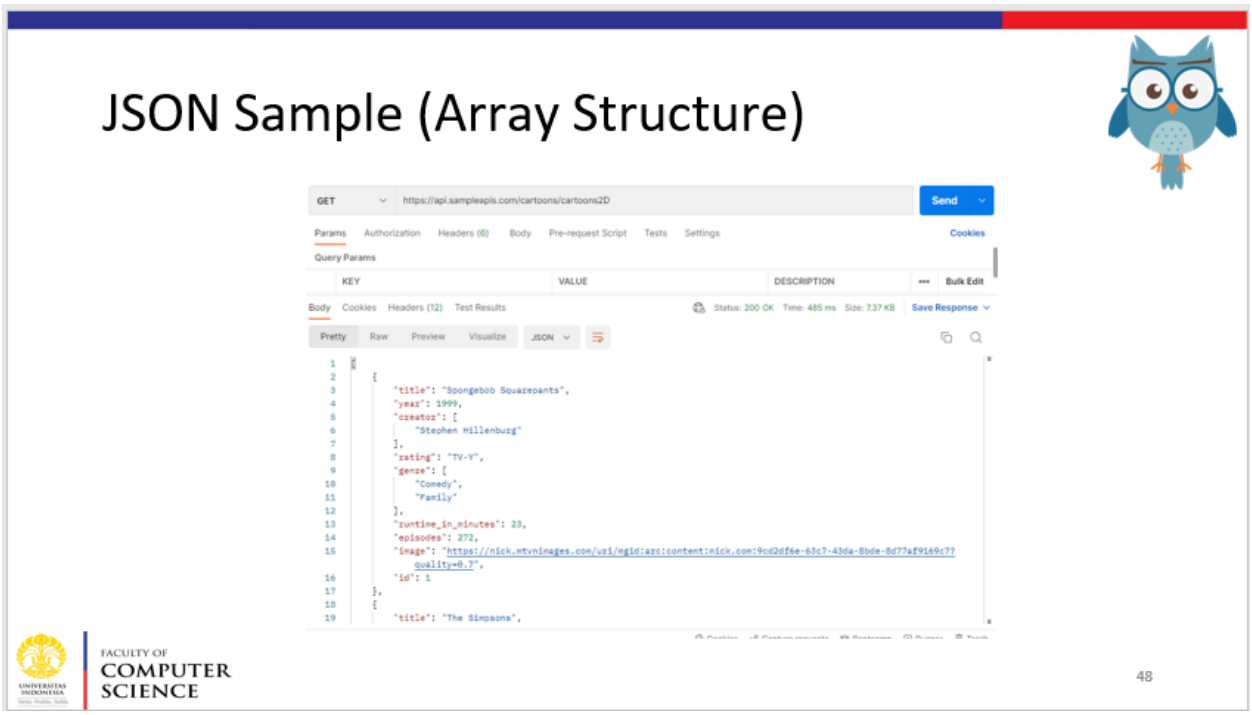
JSON Structure



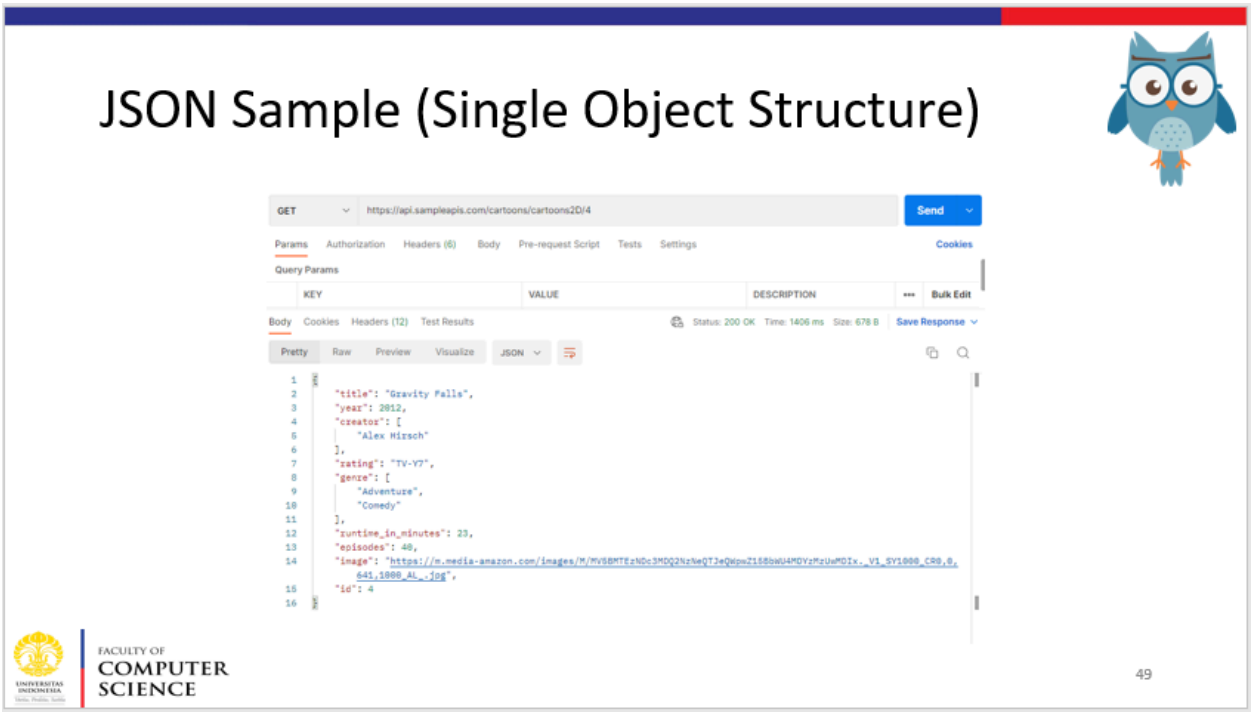
JSON structures are based on two fundamental components: a collection of name/value pairs and an ordered list of values. The name/value pairs represent properties or attributes associated with an object, while the ordered list of values represents elements within an array. These structures are universal and widely supported across modern programming languages, appearing as objects, records, structs, dictionaries, hash tables, keyed lists, associative arrays, arrays, vectors, lists, or sequences depending on the language.

The versatility of these structures allows JSON to seamlessly integrate with various programming languages, facilitating interoperability between data formats and software systems. By aligning with these universal data structures, JSON ensures compatibility and ease of use across different platforms and environments. This universal adoption underscores the significance of JSON as a standard data-interchange format in modern web development and software engineering practices.

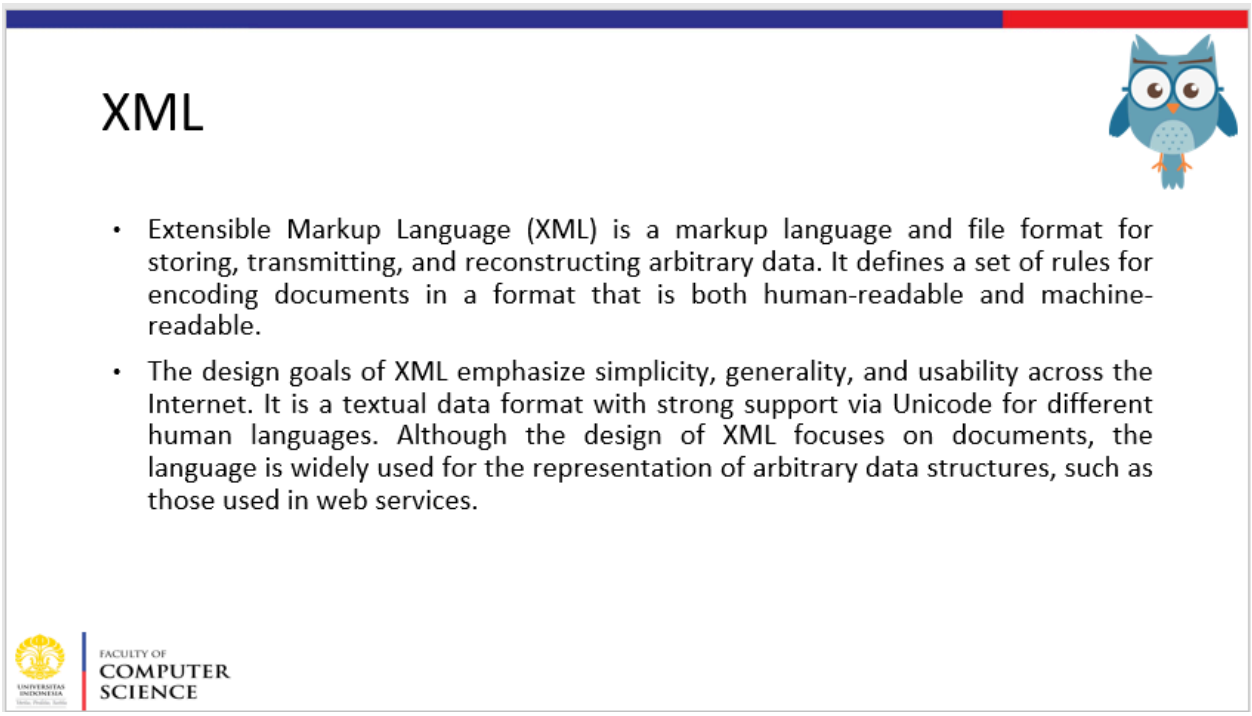
Array Structure



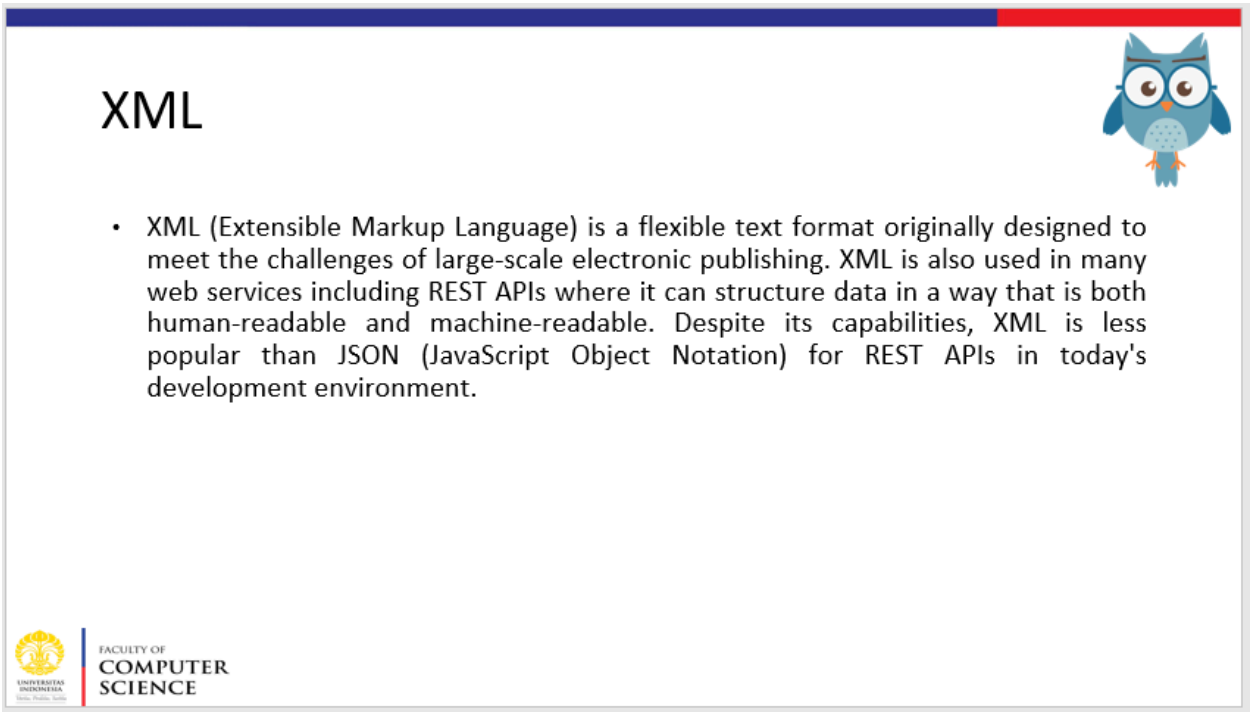
Single Object Structure



XML

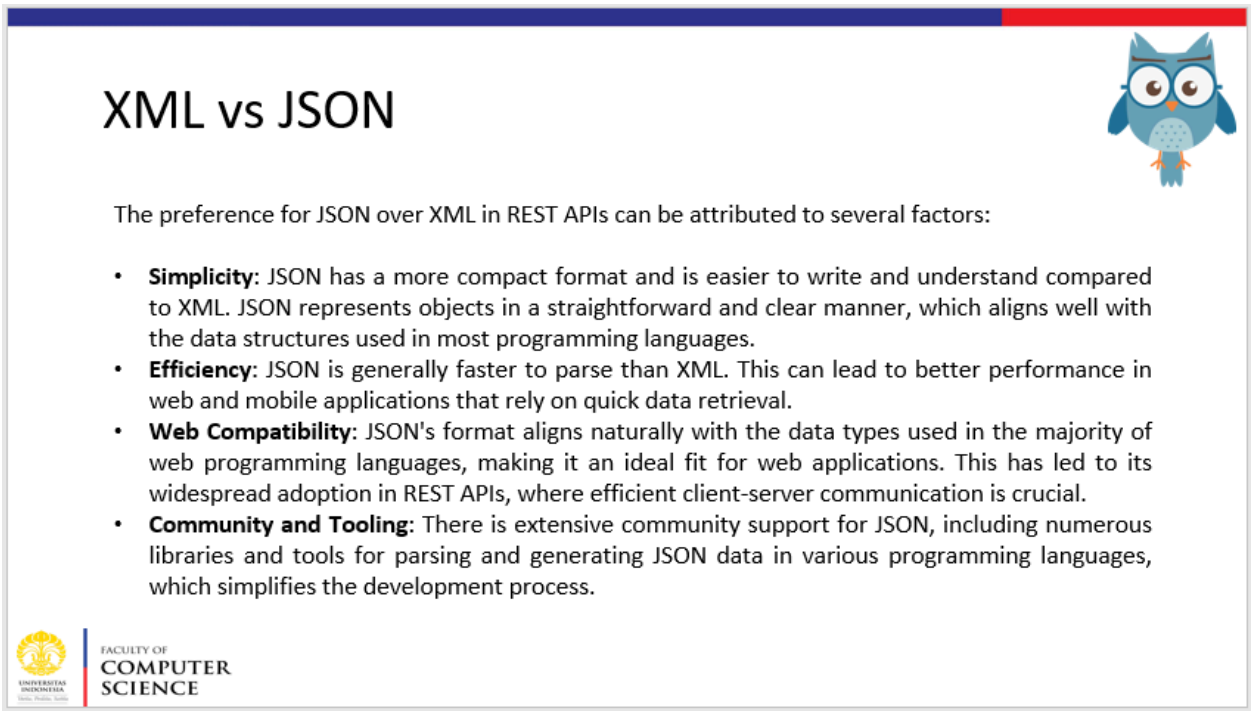


Extensible Markup Language (XML) serves as a versatile markup language and file format, catering to the storage, transmission, and reconstruction of diverse data sets. It establishes a framework of rules for encoding documents, ensuring readability for both humans and machines. XML's design principles prioritize simplicity, generality, and internet-wide usability, making it a preferred choice for various applications. Being a textual data format, XML boasts robust support through Unicode, enabling seamless integration of different human languages. While initially geared towards document representation, XML finds extensive utilization in depicting arbitrary data structures, including those prevalent in web services. Its widespread adoption highlights XML's significance as a foundational technology in data interchange, facilitating interoperability and adaptability across diverse software systems and communication protocols. Whether handling documents or complex data structures, XML continues to play a pivotal role in modern computing environments, contributing to efficient information exchange and system interoperability.



XML emerges as a flexible text format initially crafted to address the demands of extensive electronic publishing endeavors. Its versatility extends to various web services, including REST APIs, where it facilitates the organization of data in a format accessible to both humans and machines. Despite its adeptness, XML encounters reduced popularity compared to JSON (JavaScript Object Notation) within contemporary development landscapes. JSON's rise to prominence stems from its lightweight nature, ease of use, and seamless integration with modern web technologies. JSON's concise syntax and straightforward data representation make it a preferred choice for REST APIs, aligning well with the trend towards simpler, more agile development practices. However, XML's robust structure and extensive feature set still render it relevant in specific use cases, particularly those necessitating complex data structures or compatibility with legacy systems. Ultimately, the choice between XML and JSON hinges on the specific requirements and preferences of individual projects, with both formats offering distinct advantages in different contexts.

XML vs JSON



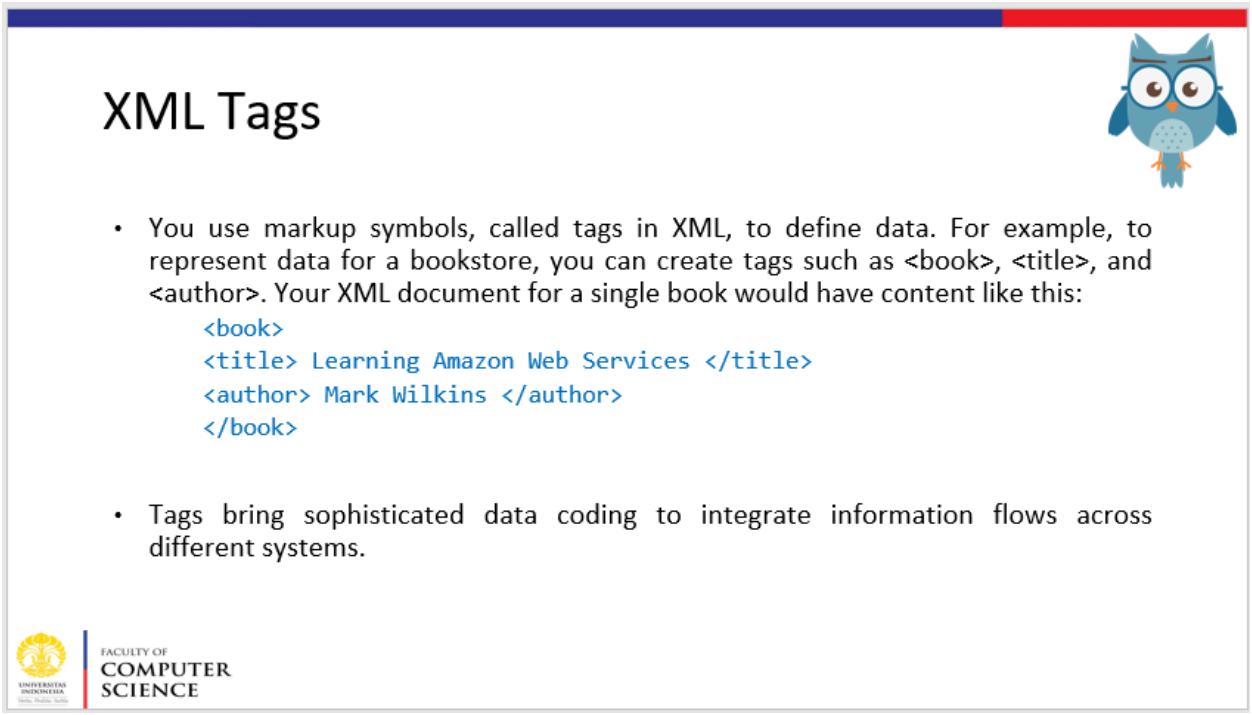
The preference for JSON over XML in REST APIs stems from several key factors. Firstly, JSON's simplicity and compact format make it easier to write, read, and understand compared to XML. JSON's straightforward representation of objects aligns well with the data structures commonly used in programming languages, enhancing developer productivity and code clarity.

Secondly, JSON tends to offer better parsing performance than XML, resulting in faster data retrieval and improved efficiency, particularly in web and mobile applications where speed is paramount.

Moreover, JSON's compatibility with web programming languages and its natural alignment with web data types make it an ideal choice for web applications. Its widespread adoption in the web development community has led to robust community support and a plethora of libraries and tools for JSON parsing and generation, streamlining the development process and enhancing developer productivity.

Overall, JSON's simplicity, efficiency, compatibility with web technologies, and extensive community support make it the preferred choice for many developers when designing REST APIs.

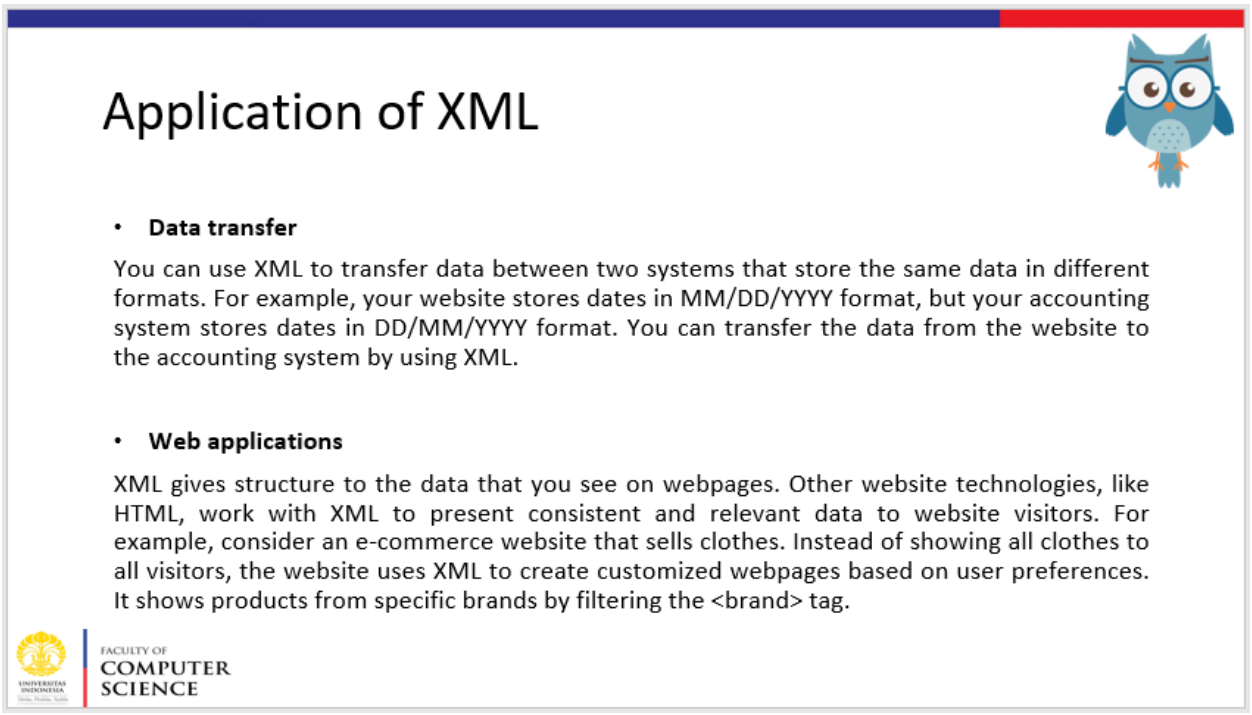
XML Tags



XML tags are essential elements used to define data structures in XML documents. These markup symbols encapsulate content and provide structure to the data. For instance, in representing bookstore information, tags like <book>, <title>, and <author> are utilized.

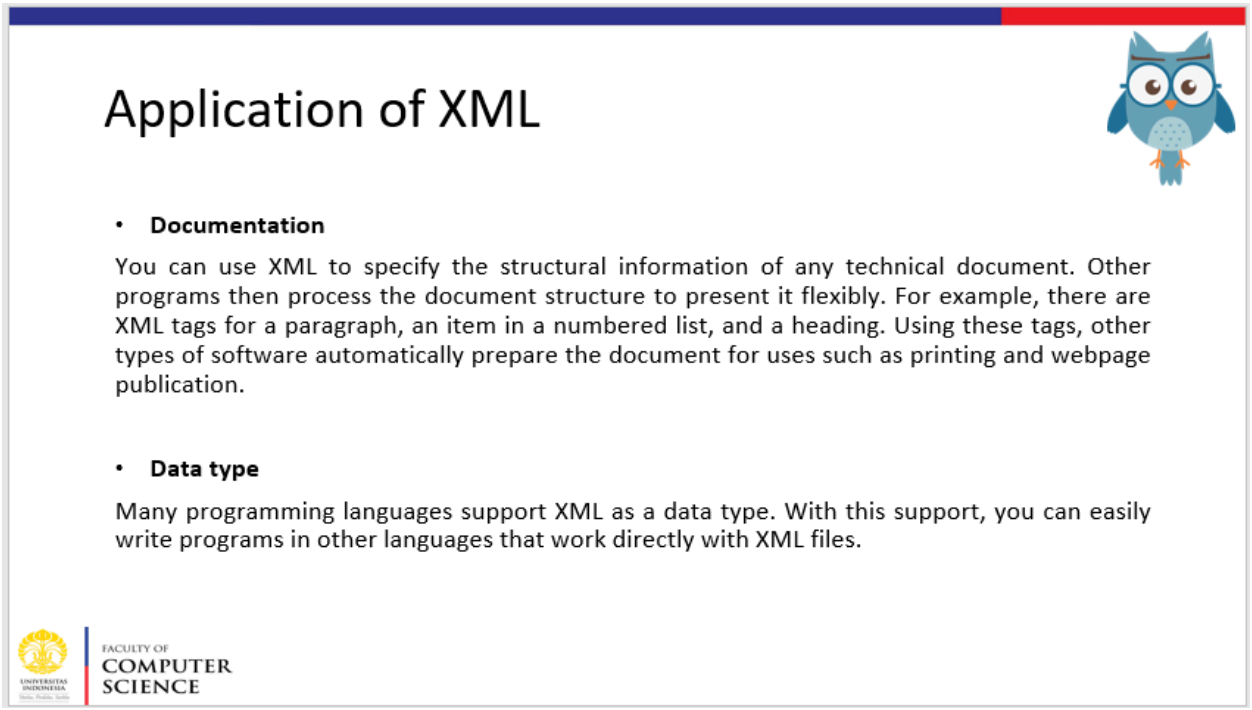
Each tag delineates specific information within the document, enhancing readability and organization. XML tags facilitate the creation of well-structured documents that can be easily understood by both humans and machines. They enable the integration of diverse information flows across different systems, allowing for seamless data exchange and interoperability. Through the use of tags, XML provides a standardized and flexible approach to representing complex data structures, making it a versatile choice for various applications, including data interchange, configuration files, and web services.

Application of XML



XML (Extensible Markup Language) finds diverse applications across various domains, owing to its flexibility and versatility. One of its primary applications lies in data transfer between systems that store data in different formats. For instance, XML facilitates seamless data exchange between systems with varying date formats, such as MM/DD/YYYY and DD/MM/YYYY, ensuring compatibility and consistency in data transmission.

In web applications, XML plays a pivotal role in structuring data for presentation on webpages. Collaborating with technologies like HTML, XML enables the creation of dynamic and personalized web content. For example, in an e-commerce website, XML assists in customizing webpages based on user preferences, filtering products based on specific attributes such as brand through XML tags like <brand>.



Additionally, XML serves as a valuable tool for documentation, allowing users to specify the structural information of technical documents. XML tags define elements such as paragraphs, numbered lists, and headings, facilitating flexible presentation across various platforms. This structured approach enhances document management, preparation for printing, and publication on webpages.

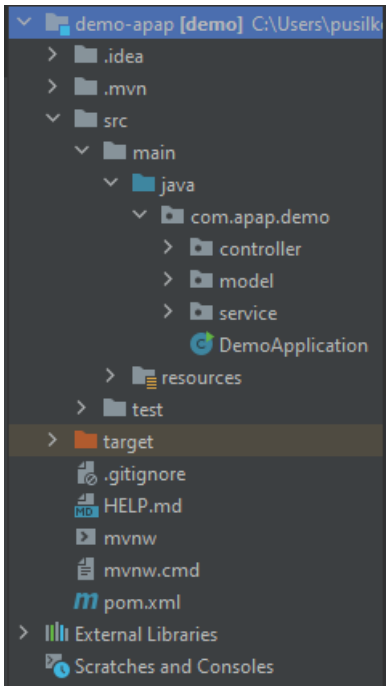
Moreover, XML functions as a data type in many programming languages, simplifying the development of programs that directly interact with XML files. This support enables seamless integration of XML into software applications, facilitating data processing, manipulation, and exchange across different systems and platforms.

Overall, XML's versatility and adaptability make it indispensable in diverse fields, including data interchange, web development, documentation, and software programming. Its standardized format and robust features contribute to efficient data management, interoperability, and flexibility in modern computing environments.

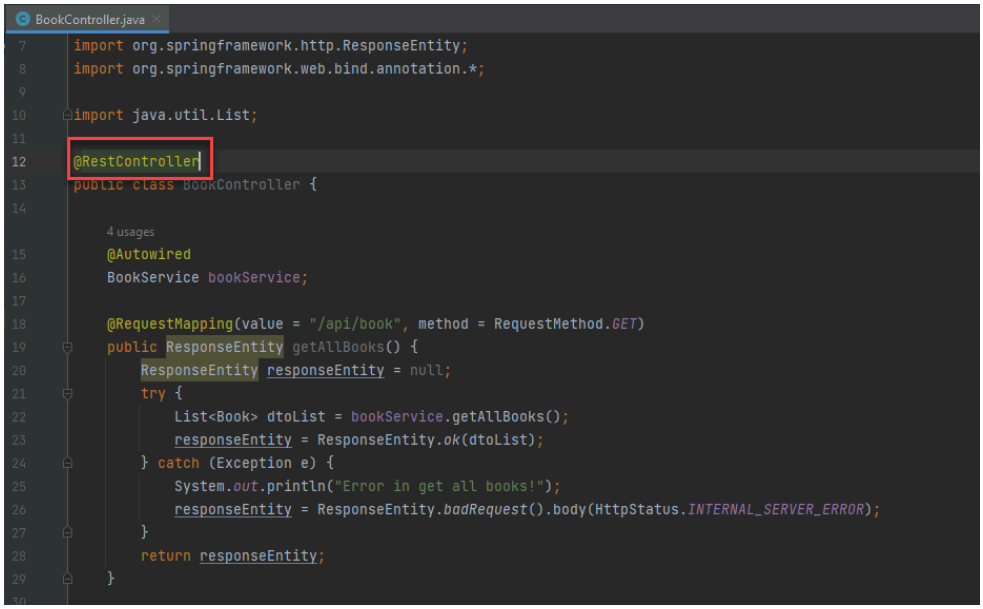
REST in Java (Spring Boot)

Source code : https://github.com/muhammad-khadafi/basic-rest

1. Create controller, model & service



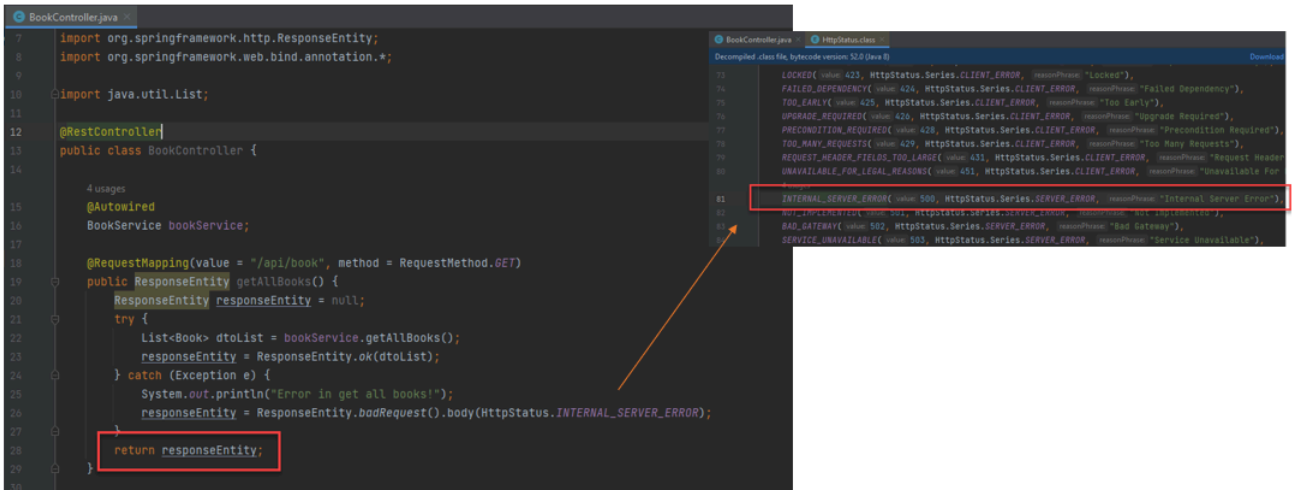
2. Common Implementation for service & model, We'll focus only for controller
3. Add `@RestController`



4. Define Endpoint & HTTP Method

```java
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
public class BookController {

    4 usages
    @Autowired
    BookService bookService;

    @RequestMapping(value = "/api/book", method = RequestMethod.GET)
    public ResponseEntity getAllBooks() {
        ResponseEntity responseEntity = null;
        try {
            List<Book> dtoList = bookService.getAllBooks();
            responseEntity = ResponseEntity.ok(dtoList);
        } catch (Exception e) {
            System.out.println("Error in get all books!");
            responseEntity = ResponseEntity.badRequest().body(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return responseEntity;
    }
```

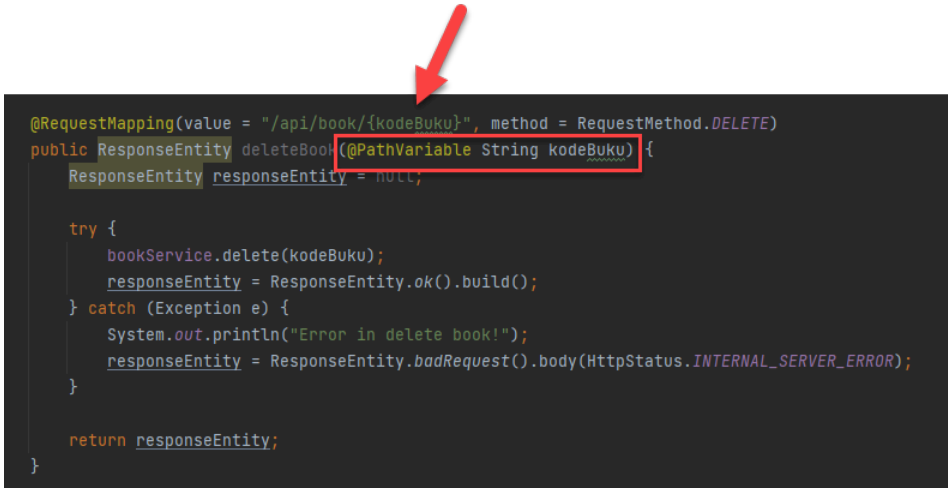
5. Use response entity to return the response with HTTP status code



6. Add request body for supplying the object parameter in POST and PUT method

```java
@RequestMapping(value = "/api/book", method = RequestMethod.POST)
public ResponseEntity addBook(@RequestBody Book book) {
    ResponseEntity responseEntity = null;

    try {
        bookService.add(book);
        responseEntity = ResponseEntity.ok().build();
    } catch (Exception e) {
        System.out.println("Error in add book!");
        responseEntity = ResponseEntity.badRequest().body(HttpStatus.INTERNAL_SERVER_ERROR);
    }

    return responseEntity;
}
```

7. Add path variable for supplying the id parameter in DELETE method



```
@RequestMapping(value = "/api/book/{kodeBuku}", method = RequestMethod.DELETE)
public ResponseEntity deleteBook(@PathVariable String kodeBuku) {
    ResponseEntity responseEntity = null;

    try {
        bookService.delete(kodeBuku);
        responseEntity = ResponseEntity.ok().build();
    } catch (Exception e) {
        System.out.println("Error in delete book!");
        responseEntity = ResponseEntity.badRequest().body(HttpStatus.INTERNAL_SERVER_ERROR);
    }

    return responseEntity;
}
```

REST in Rust (Actix Web)

Source code : https://github.com/muhammad-khadafi/rust-rest

1. Add dependencies

Cargo.toml

```toml
[package]
name = "sample_rest"
version = "0.1.0"
edition = "2021"

# Dependencies
[dependencies]
actix-web = "4.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

```

2. Create main.rs for basic config

```rust
use actix_web::{web, App, HttpServer};
mod handlers;
mod models;
mod state;

use crate::state::AppState;

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let app_data = web::Data::new(AppState::new());

    HttpServer::new(move || {
        App::new()
            .app_data(app_data.clone())
            .configure(handlers::config_handlers)
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

3. Create struct in model

src > models.rs

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Clone)]
pub struct Book {
    pub id: u32,
    pub title: String,
    pub author: String,
    pub year: u32,
}
```

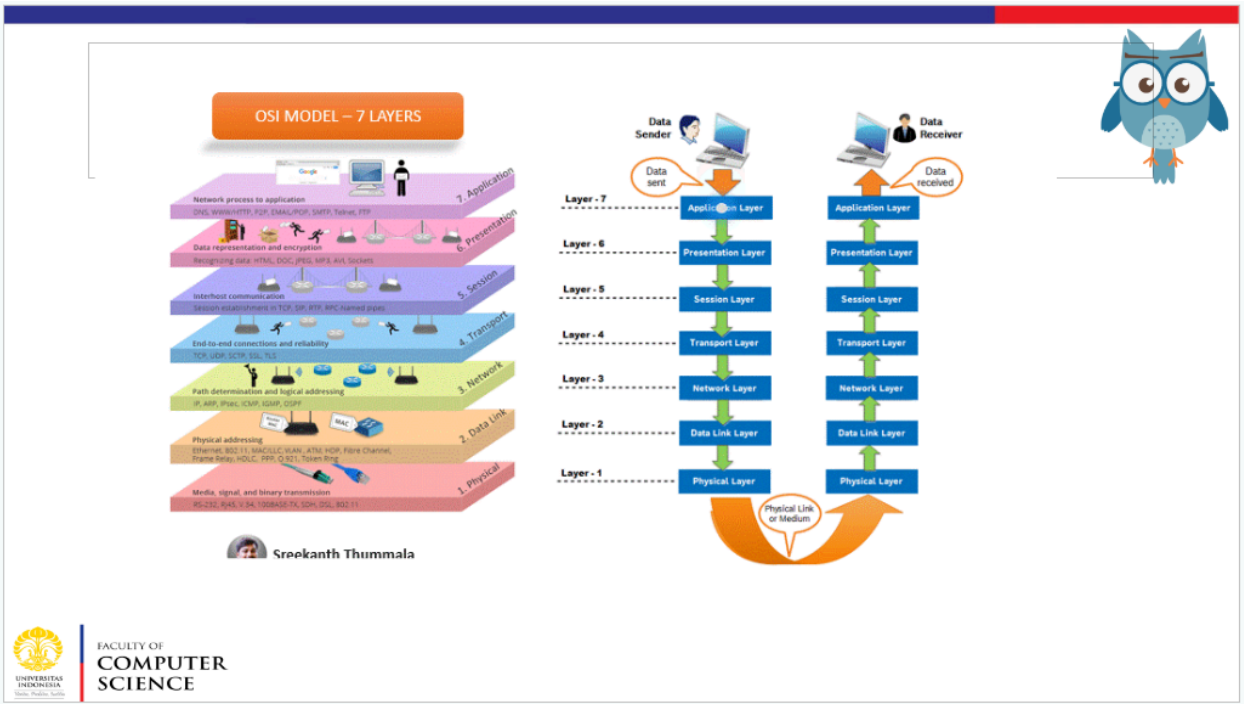
4. Routing REST API endpoint in controller/handler

src > handlers.rs

```rust
use actix_web::{web, HttpResponse, Responder};
use crate::models::Book;
use crate::state::AppState;
use serde_json::json;

pub fn config_handlers(cfg: &mut web::ServiceConfig) {
    cfg.route("/books", web::post().to(create_book))
       .route("/books/{id}", web::get().to(get_book))
       .route("/books/{id}", web::put().to(update_book))
       .route("/books/{id}", web::delete().to(delete_book));
}

async fn create_book(data: web::Data<AppState>, book: web::Json<Book>) -> impl Responder {
    data.create_book(book.into_inner());
    HttpResponse::Created().finish()
}

async fn get_book(data: web::Data<AppState>, book_id: web::Path<u32>) -> impl Responder {
    if let Some(book) = data.get_book(*book_id) {
        HttpResponse::Ok().json(book)
    } else {
        HttpResponse::NotFound().finish()
    }
}
```

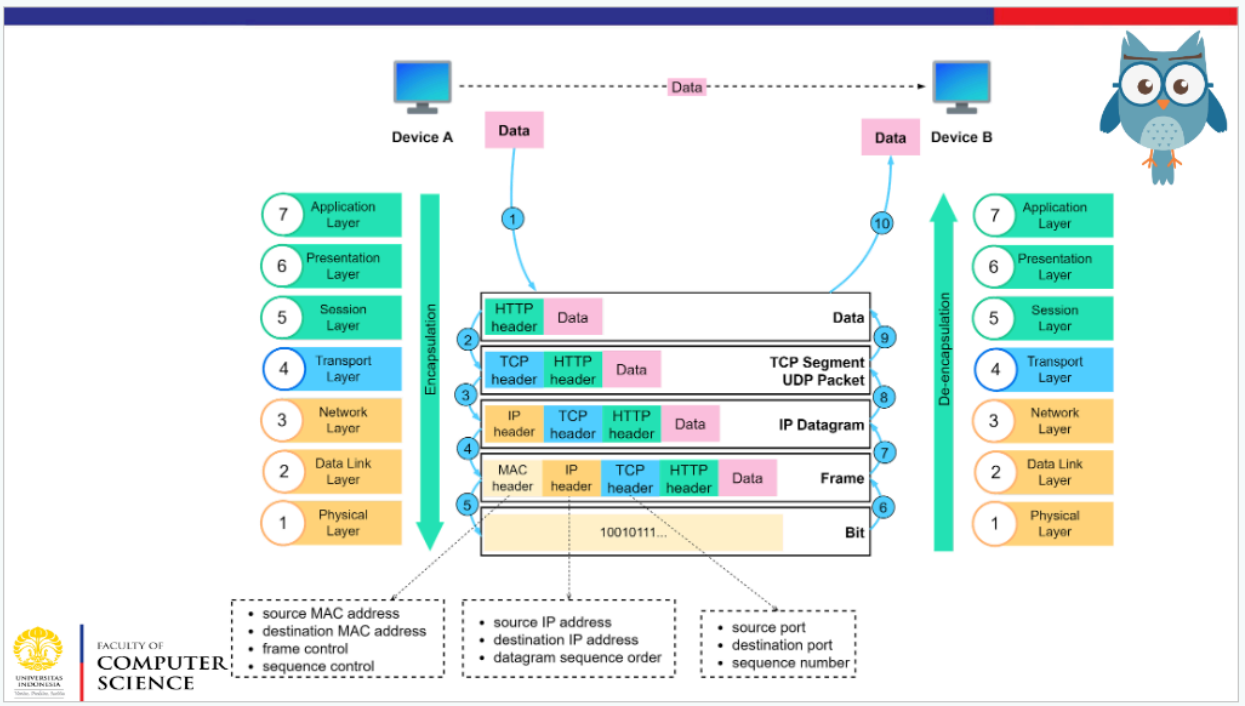
```rust
async fn update_book(data: web::Data<AppState>, book_id: web::Path<u32>, book: web::Json<Book>) -> impl Responder {
    if let Some(book) = data.update_book(*book_id, book.into_inner()) {
        HttpResponse::Ok().json(book)
    } else {
        HttpResponse::NotFound().finish()
    }
}

async fn delete_book(data: web::Data<AppState>, book_id: web::Path<u32>) -> impl Responder {
    if data.delete_book(*book_id) {
        HttpResponse::Ok().json(json!({"message": "Book deleted"}))
    } else {
        HttpResponse::NotFound().finish()
    }
}
```

A Brief Introduction to Computer Network



The OSI (Open Systems Interconnection) model is a seven-layer framework that standardizes the functions of telecommunication and computing systems to facilitate interoperability between different technologies. The layers, from lowest to highest, are Physical, Data Link, Network, Transport, Session, Presentation, and Application. Each layer has a specific role, from transmitting raw data over physical media (Physical layer) to ensuring reliable data transfer (Transport layer), and up to providing network services to end-user applications (Application layer).

The model ensures seamless data flow by allowing each layer to interact only with its adjacent layers. For instance, the Data Link layer handles error detection for the Physical layer, while the Network layer manages routing through logical addressing. Higher up, the Presentation layer translates data formats, and the Application layer interfaces directly with user applications like web browsers and email clients. Understanding the OSI model is essential for network design and troubleshooting, as it provides a clear structure for how data should be transmitted and received across different systems.



In the OSI model, data flows from the sender to the receiver through seven layers, each adding its own specific function to the process.

1. Application Layer: The sender's application (like a web browser or email client) creates the data to be sent.
2. Presentation Layer: This layer translates the data into a format suitable for transmission, encrypts it if necessary, and compresses it to reduce size.
3. Session Layer: It establishes a session between the sender and receiver, managing the connection and ensuring data exchanges are properly synchronized.
4. Transport Layer: This layer segments the data into smaller units, adds error-checking information, and ensures reliable delivery through protocols like TCP or UDP.
5. Network Layer: The segmented data is packaged into packets with source and destination IP addresses. This layer determines the best route for the data to reach the receiver.
6. Data Link Layer: Packets are framed with MAC addresses for node-to-node delivery. It handles error detection and correction from the physical transmission.
7. Physical Layer: The data is converted into electrical, optical, or radio signals and transmitted over the physical medium (like cables or wireless).

At the receiver's end, the data flows up through these layers in reverse order, from the Physical layer to the Application layer:

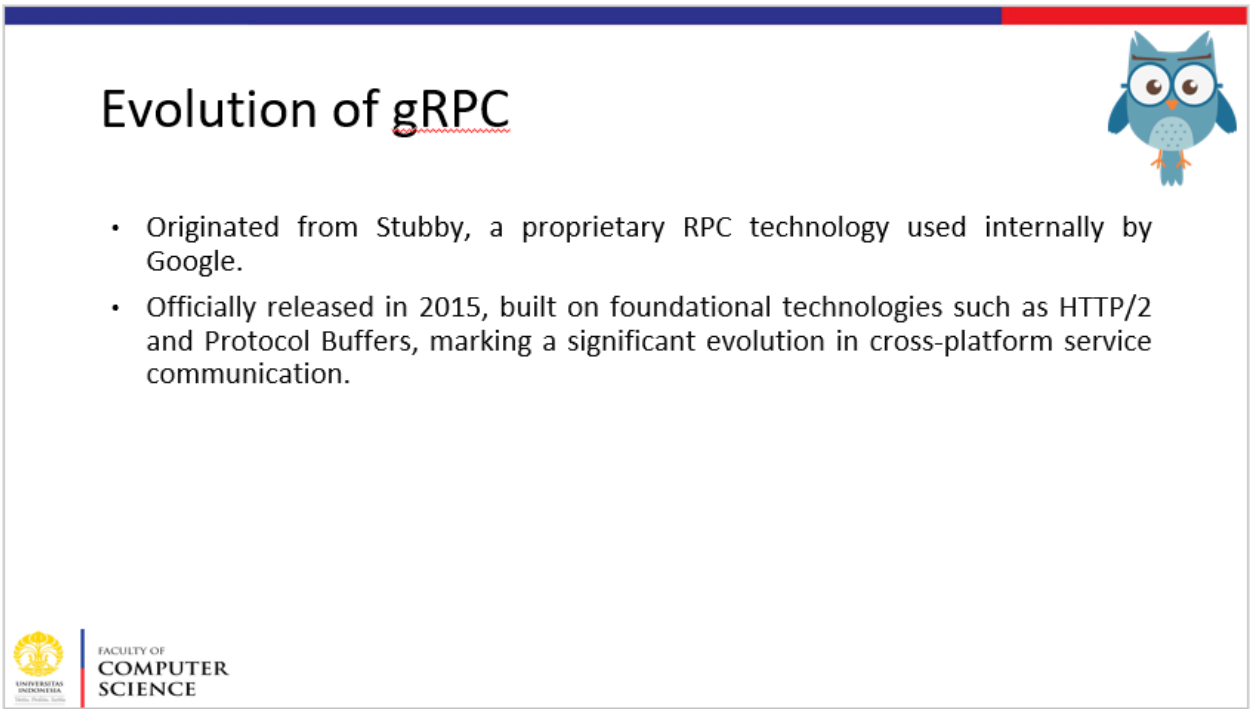
1. Physical Layer: Receives the raw signals and converts them back into bits.
2. Data Link Layer: Reassembles the frames, checks for errors, and forwards the data to the Network layer.
3. Network Layer: Extracts the packets, verifies IP addresses, and routes the data to the correct Transport layer.
4. Transport Layer: Reassembles the segments, performs error checking, and ensures complete data transfer to the Session layer.
5. Session Layer: Manages and maintains the session, ensuring the data is synchronized correctly.
6. Presentation Layer: Decompresses and decrypts the data, translating it back into a readable format for the application.
7. Application Layer: Delivers the data to the receiving application (like a web browser or email client), completing the communication process.

gRPC

gRPC, an acronym for Google Remote Procedure Call, stands as a high-performance open-source framework crafted by Google. It streamlines communication between client and server applications, enabling them to invoke methods on each other similar to local function calls. This efficiency makes it particularly suitable for scenarios involving server-to-server communications, as well as interactions between browser or mobile applications and backend systems. gRPC's effectiveness extends to enhancing connectivity within intricate architectures such as microservices, where efficient communication is paramount.

By providing a standardized and efficient means of communication, gRPC contributes to the development of scalable and robust distributed systems. Its support for multiple programming languages and platforms further enhances its versatility and adoption across diverse environments. Overall, gRPC plays a crucial role in facilitating seamless and high-performance communication in modern software architectures, empowering developers to build efficient and scalable distributed applications.

Evolution of gRPC

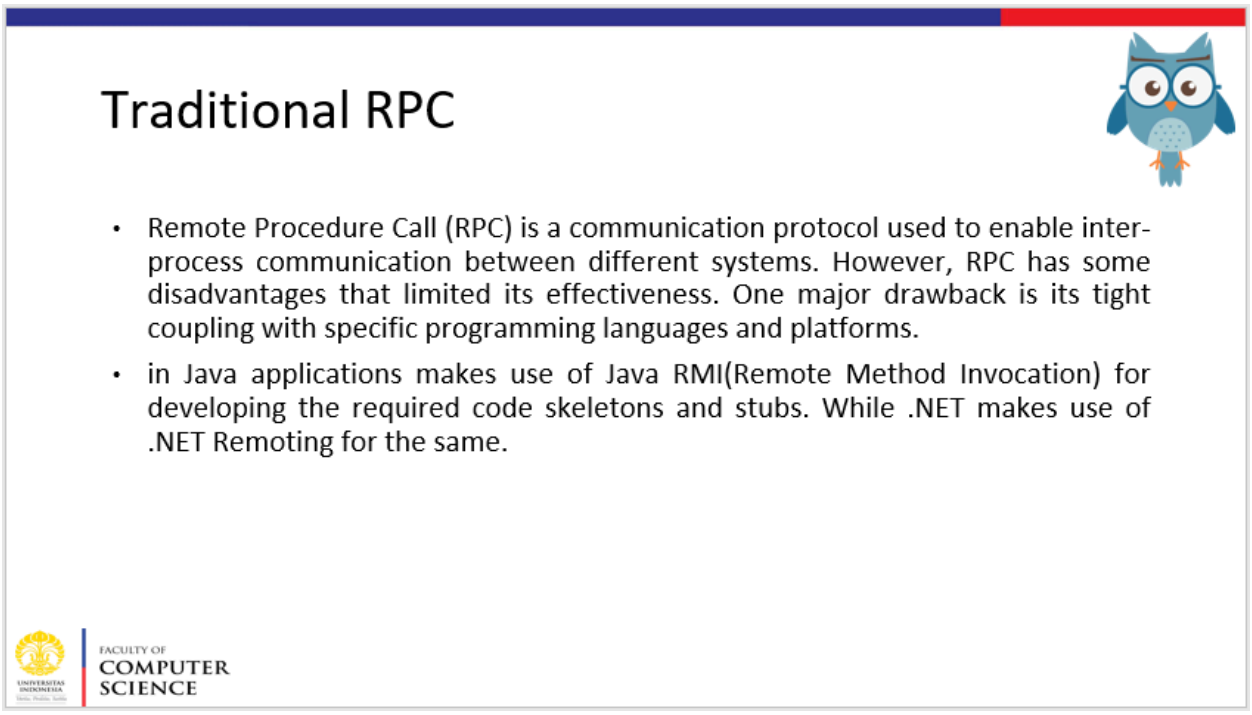


The evolution of gRPC traces back to its origins in Stubby, a proprietary RPC (Remote Procedure Call) technology developed internally by Google. Stubby laid the groundwork for efficient and scalable service communication within Google's infrastructure. In 2015, gRPC was officially released, representing a significant advancement in cross-platform service communication. Built on foundational technologies like HTTP/2 and Protocol Buffers, gRPC revolutionized the way applications communicate with each other over the network.

By leveraging HTTP/2, gRPC introduces features such as multiplexing, header compression, and server push, enhancing performance and reducing latency in communication between client and server applications. Protocol Buffers, on the other hand, provide a language-neutral, platform-neutral mechanism for serializing structured data, enabling efficient data exchange between different systems.

The release of gRPC marked a milestone in the evolution of RPC frameworks, offering developers a robust and efficient solution for building distributed systems. Its adoption continues to grow rapidly, powering communication in diverse environments ranging from microservices architectures to cloud-native applications. Overall, gRPC's evolution reflects a commitment to advancing the state of the art in service communication and facilitating the development of scalable and interoperable software systems.

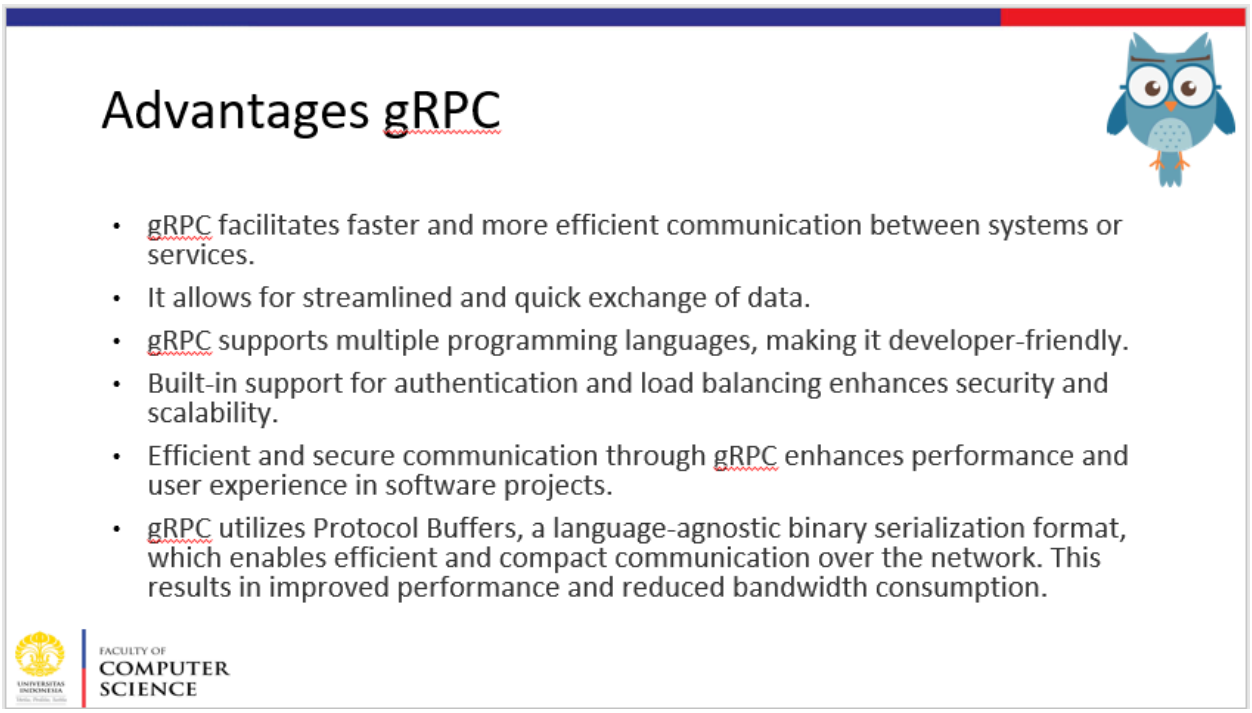
Traditional RPC



Traditional gRPC, or Remote Procedure Call (RPC), serves as a vital communication protocol facilitating inter-process communication across different systems. However, despite its utility, traditional RPC implementations possess inherent limitations that hinder their effectiveness. One significant drawback lies in their tight coupling with specific programming languages and platforms. For instance, Java applications typically rely on Java RMI (Remote Method Invocation) for generating necessary code skeletons and stubs, while .NET applications utilize .NET Remoting for similar functionality. This language-specific coupling restricts interoperability and hampers the scalability of RPC-based systems, as they are inherently bound to particular programming environments.

Furthermore, traditional RPC frameworks often lack the flexibility and extensibility required for modern distributed systems. They may struggle to accommodate evolving requirements and may not seamlessly integrate with emerging technologies or architectures, such as microservices or cloud-native environments. As a result, developers may encounter challenges when attempting to scale and adapt RPC-based solutions to meet the demands of contemporary software development paradigms. In response to these limitations, the evolution of gRPC represents a significant advancement, offering a more flexible, efficient, and language-agnostic approach to RPC-based communication.

Advantages gRPC



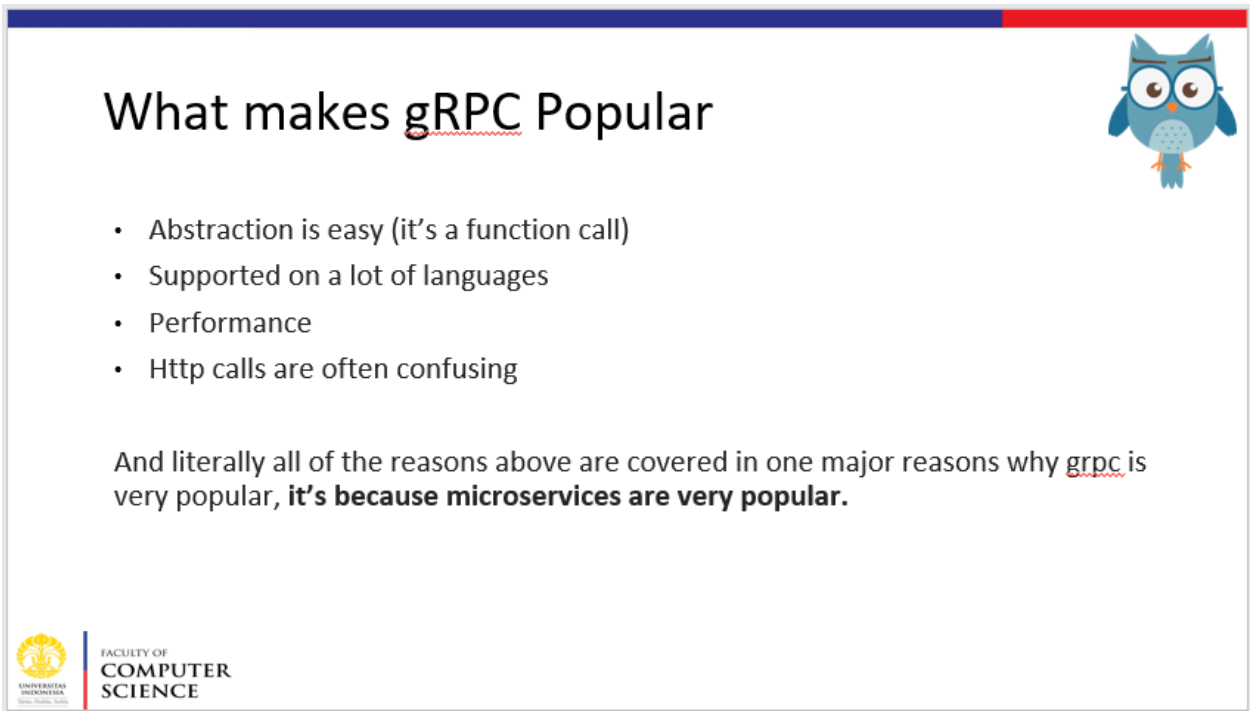
gRPC offers several advantages that contribute to enhanced communication and efficiency in software projects. Firstly, it facilitates faster and more efficient communication between systems or services, allowing for streamlined and quick exchange of data. This results in improved performance and responsiveness, leading to a better user experience.

One of gRPC's notable advantages is its support for multiple programming languages, making it accessible and developer-friendly across different environments. This versatility enables teams to leverage their preferred programming languages and frameworks while benefiting from gRPC's capabilities.

Moreover, gRPC comes with built-in support for authentication and load balancing, enhancing security and scalability in distributed systems. This simplifies the implementation of robust security measures and ensures reliable service delivery in high-demand environments.

Additionally, gRPC utilizes Protocol Buffers, a language-agnostic binary serialization format. This enables efficient and compact communication over the network, reducing bandwidth consumption and optimizing resource utilization. As a result, gRPC not only improves performance but also helps minimize infrastructure costs, making it a compelling choice for modern software projects. Overall, gRPC's combination of speed, efficiency, versatility, and security makes it a valuable tool for building scalable and high-performance distributed systems.

What makes gRPC Popular



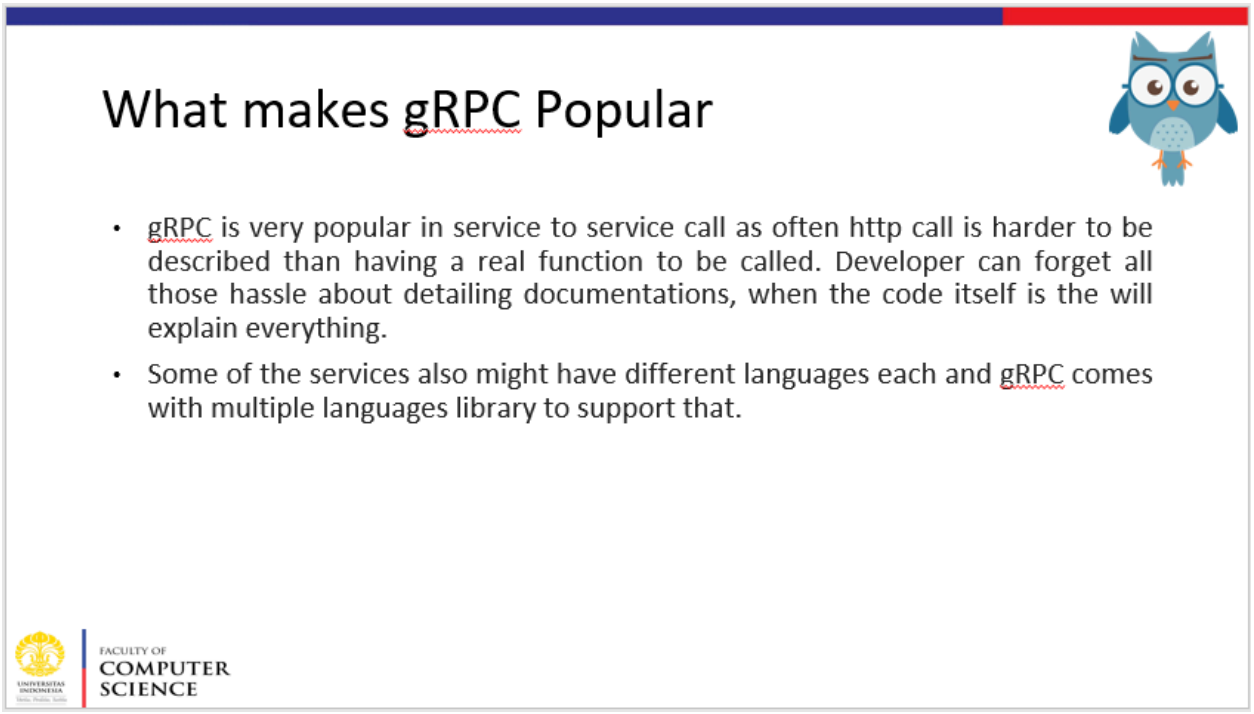
gRPC has garnered popularity for several reasons, which collectively contribute to its widespread adoption in modern software development. Firstly, gRPC offers an abstraction that simplifies communication between services by treating it akin to a function call. This high-level abstraction streamlines the development process and reduces the complexity of implementing inter-service communication.

Another key factor driving gRPC's popularity is its extensive language support. With support for multiple programming languages, developers can seamlessly integrate gRPC into their projects, regardless of their preferred language or framework.

Moreover, gRPC boasts superior performance compared to traditional HTTP calls, thanks to its efficient binary serialization format and support for HTTP/2. This enhanced performance translates to faster response times and improved scalability, particularly in distributed systems.

Furthermore, the inherent complexity of HTTP calls often poses challenges for developers, whereas gRPC provides a more straightforward and intuitive approach to communication between services.

Ultimately, the rising popularity of microservices architecture aligns closely with the adoption of gRPC. Microservices promote modular and scalable application development, and gRPC's features align perfectly with the requirements of microservices-based architectures.

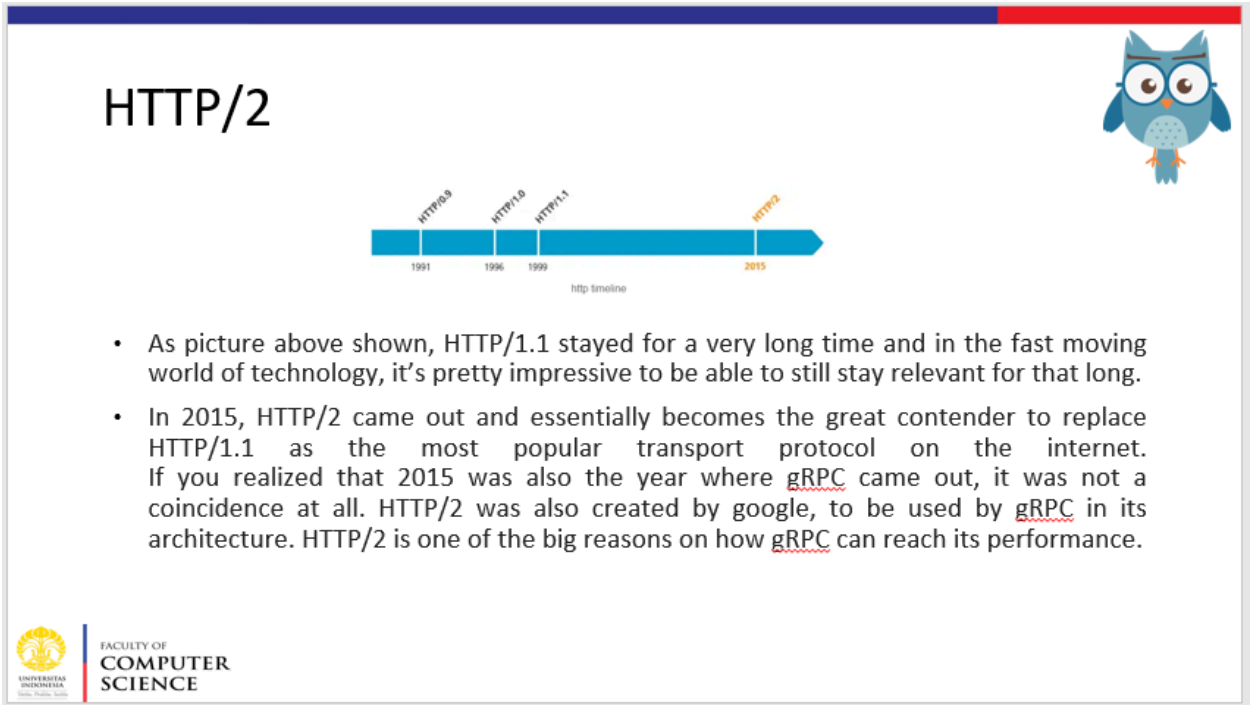


gRPC has gained significant popularity, particularly in service-to-service communication, due to several key factors. Firstly, gRPC simplifies communication between services by abstracting it as a function call. This abstraction eliminates the need for extensive documentation and enables developers to focus on writing code that clearly describes the intended functionality. With gRPC, developers can seamlessly invoke remote procedures without the complexities often associated with HTTP calls, enhancing productivity and reducing development time.

Moreover, gRPC's support for multiple programming languages further contributes to its popularity. In a diverse ecosystem where services may be implemented in different languages, gRPC's language-specific libraries ensure interoperability and ease of integration. This allows organizations to leverage their existing technology stack and empowers teams to choose the most suitable programming language for each service, without sacrificing communication efficiency.

gRPC Achitecture

HTTP/2

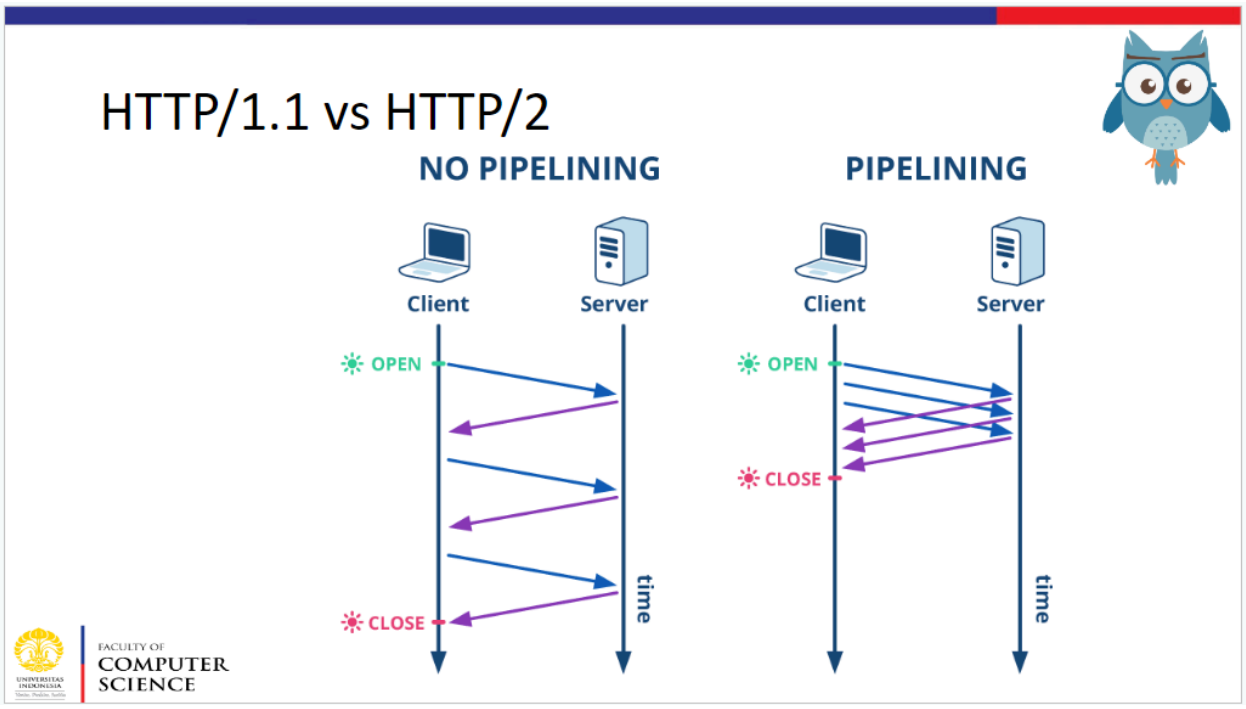


The gRPC architecture relies heavily on the underlying HTTP/2 protocol, which plays a crucial role in enabling efficient communication between client and server applications. HTTP/1.1 had long been the dominant transport protocol on the internet, showcasing impressive longevity in the rapidly evolving technological landscape. However, in 2015, HTTP/2 emerged as a significant advancement, poised to replace HTTP/1.1 as the new standard for internet communication.

Coinciding with the release of HTTP/2, gRPC also made its debut in 2015. This timing was no coincidence; in fact, HTTP/2 was specifically designed by Google to complement gRPC's architecture. By leveraging HTTP/2, gRPC is able to achieve superior performance and efficiency in service communication.

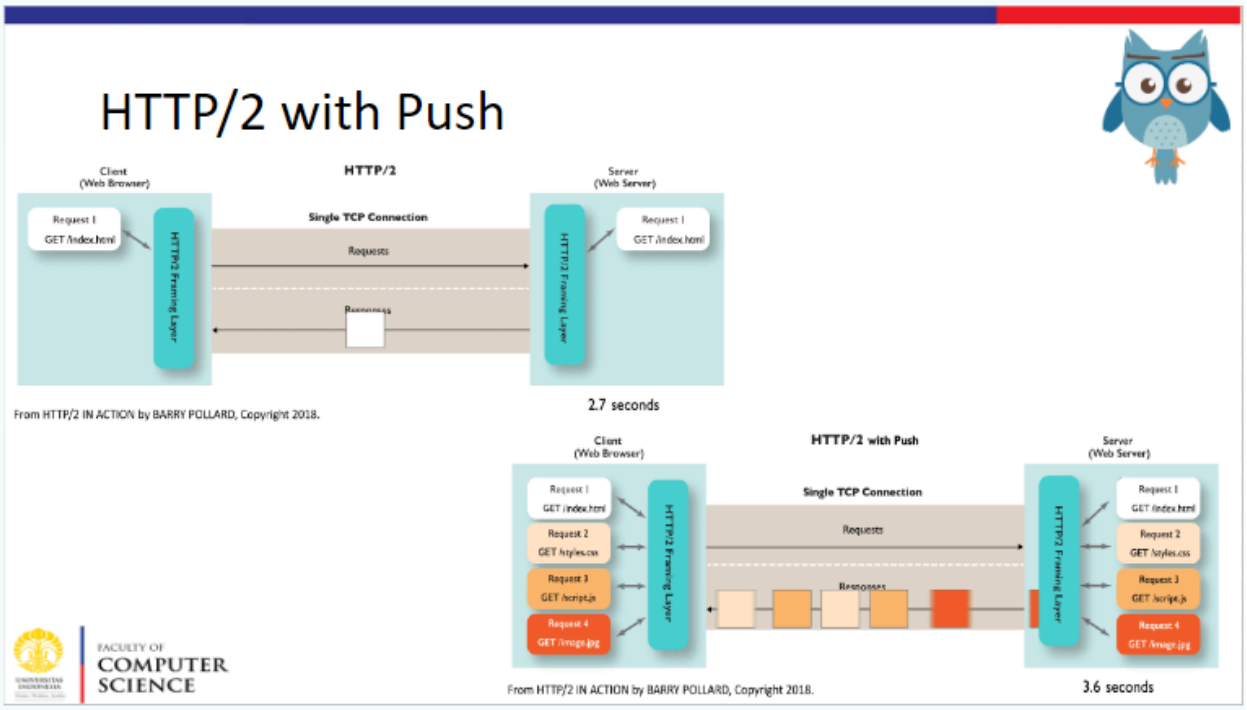
HTTP/2 introduces several key features, including multiplexing, header compression, and server push, which collectively contribute to faster and more scalable communication between client and server applications. These advancements align perfectly with the goals of gRPC, enabling it to deliver high-performance, low-latency communication in modern distributed systems architectures.

Overall, the integration of HTTP/2 into the gRPC architecture represents a pivotal advancement, underscoring the importance of modern transport protocols in facilitating efficient and scalable communication in distributed systems. As organizations continue to embrace microservices architectures and the demand for high-performance service communication grows, the synergy between gRPC and HTTP/2 is expected to play a central role in shaping the future of distributed application development.



HTTP/1.1 and HTTP/2 represent significant advancements in how web protocols handle data transmission, especially in the context of gRPC. In HTTP/1.1, requests and responses are processed sequentially over individual connections, leading to potential delays and inefficiencies due to head-of-line blocking. This means each request must wait for the previous one to complete, impacting overall performance.

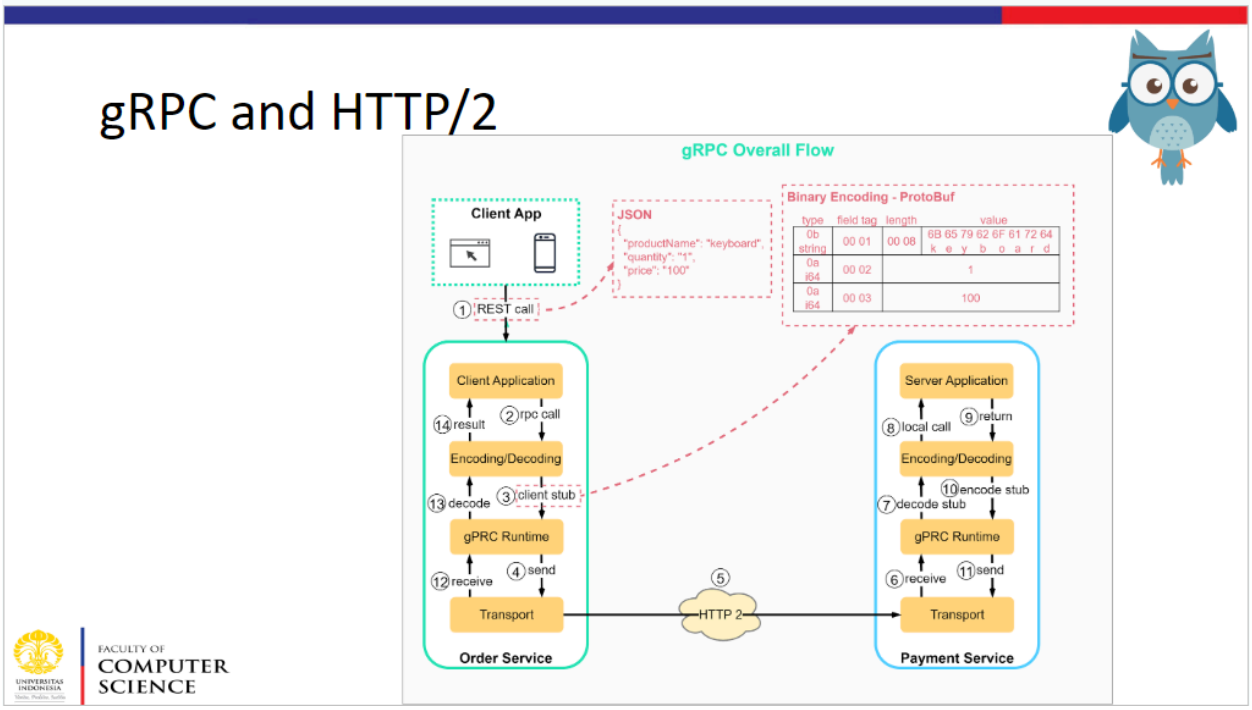
HTTP/2, however, introduces pipelining through multiplexing. This allows multiple streams of data (requests and responses) to be sent and received concurrently over a single TCP connection. Each stream operates independently, eliminating the bottleneck of sequential processing. This improvement enhances gRPC's ability to handle multiple RPC calls simultaneously, improving efficiency, reducing latency, and optimizing resource utilization. Additionally, HTTP/2's binary framing layer enhances data encoding and reduces overhead compared to HTTP/1.1's text-based format, further boosting performance in gRPC applications. Overall, HTTP/2's pipelining capabilities make it a superior choice for modern, high-performance RPC frameworks like gRPC.



HTTP/2 introduces several features that enhance web communication, one of the most notable being server push. In the context of gRPC, an advanced Remote Procedure Call framework that leverages HTTP/2, server push can significantly improve efficiency and performance in certain scenarios.

Server push in HTTP/2 allows the server to proactively send resources to the client before the client explicitly requests them. This can be particularly useful in gRPC for preemptively pushing related data or configuration updates that the client is likely to need. For instance, when a client makes an initial request, the server can push additional related data streams that the client will require soon after. This reduces the round-trip time and latency, as the client does not have to request each resource individually.

In gRPC, server push can streamline operations by ensuring that essential data is available to the client as soon as it's needed, without the need for multiple request-response cycles. For example, in a streaming scenario, where the client and server maintain a long-lived connection and continuously exchange data, server push can ensure that data dependencies are sent ahead of time, optimizing the communication flow and reducing waiting periods.



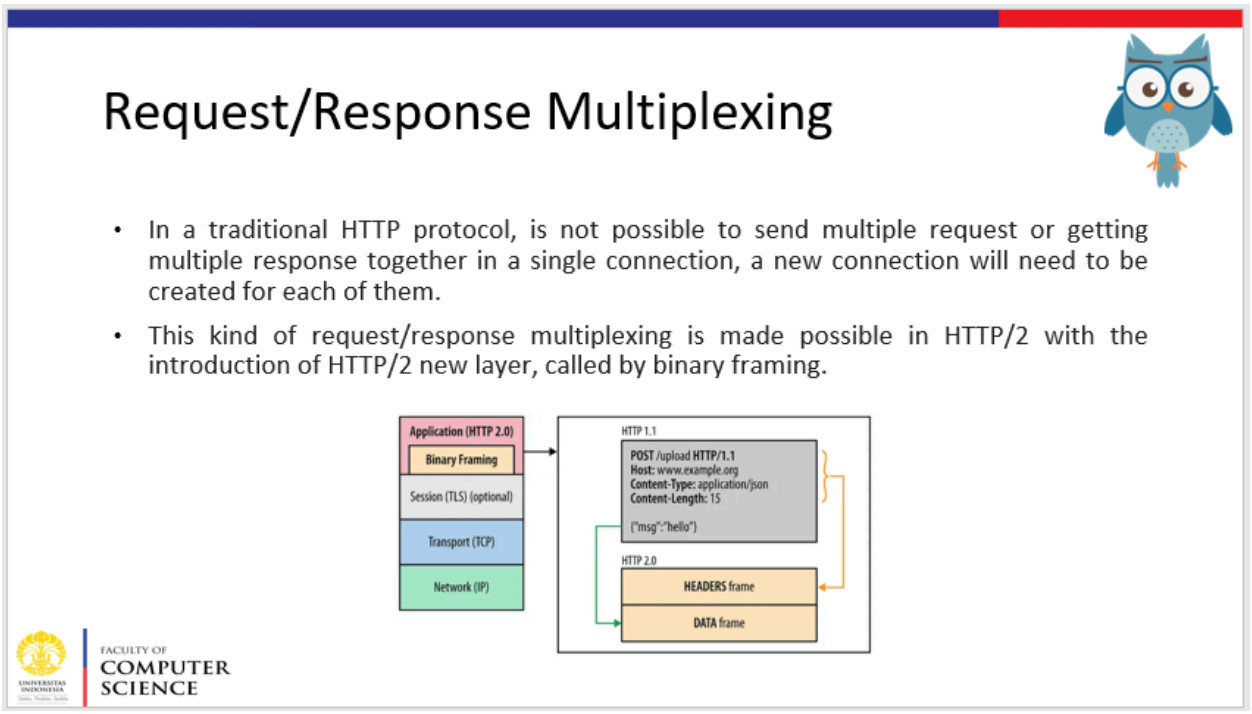
When a client application initiates the request, it makes a REST call to the Order Service. The Order Service processes this request and encodes the order details into a protocol buffer format. This encoded data is used to generate a client stub, representing the gRPC call. The gRPC runtime in the Order Service sending the request through the transport layer.

The request travels over the HTTP/2 network to the Payment Service's transport layer, which receives it and passes it to the gRPC runtime on the server side. The gRPC runtime decodes the protocol buffer message into a format the Payment Service application can process. The Payment Service then processes the request, such as confirming a payment, and generates a response.

The response from the Payment Service is encoded back into a protocol buffer format by the gRPC runtime. This encoded response is sent through the Payment Service's transport layer, traveling back over the HTTP/2 network to the Order Service's transport layer. The Order Service's transport layer receives the response and hands it off to the gRPC runtime.

The gRPC runtime in the Order Service decodes the response message and sends the result back to the client application. This completes the communication flow.

Request/Reponse Multiplexing

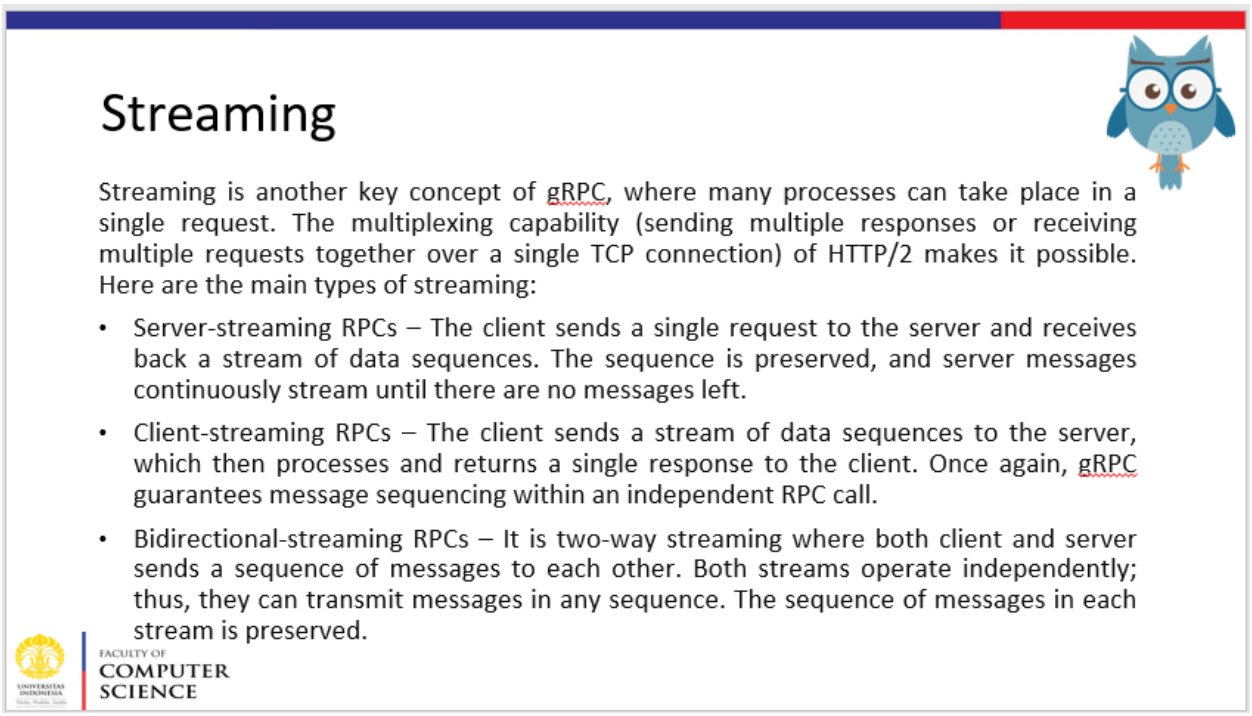


In traditional HTTP protocols, the inability to send multiple requests or receive multiple responses within a single connection presents a significant limitation. Each request or response necessitates the creation of a new connection, resulting in increased latency and resource consumption. However, HTTP/2 addresses this issue by introducing a new layer called binary framing, enabling request/response multiplexing.

With request/response multiplexing, HTTP/2 allows for the concurrent transmission of multiple requests and responses over a single connection. This means that clients can initiate multiple requests to the server simultaneously, and the server can respond with multiple responses in any order, all within the same connection. This capability dramatically improves the efficiency and performance of communication between client and server applications.

By leveraging request/response multiplexing, gRPC can make full use of the capabilities provided by HTTP/2 to achieve high-performance, low-latency communication. Clients can issue multiple RPC calls concurrently without waiting for previous calls to complete, while servers can process requests in parallel and respond in an efficient manner. This asynchronous communication model enhances scalability and responsiveness, making gRPC well-suited for building modern distributed systems where efficient communication is paramount. Overall, request/response multiplexing represents a fundamental feature of the gRPC architecture, enabling efficient and scalable communication in distributed applications.

Streaming



Streaming is a fundamental concept within the gRPC architecture, enabled by the multiplexing capabilities of HTTP/2. It allows for the efficient transmission of data sequences between client and server, facilitating various types of streaming RPCs.

The first type, server-streaming RPCs, involves the client sending a single request to the server and receiving back a continuous stream of data sequences. This allows for the transmission of large volumes of data in a sequential manner, with the server sending messages continuously until the stream is exhausted.

Conversely, client-streaming RPCs enable the client to send a stream of data sequences to the server, which processes the data and returns a single response. This type of RPC is beneficial for scenarios where the client needs to transmit a large amount of data to the server for processing.

Bidirectional-streaming RPCs represent a two-way streaming mechanism where both client and server can send and receive messages independently. This bidirectional communication allows for real-time interaction between client and server, with messages being transmitted in any sequence while preserving the order within each stream.

Overall, streaming in gRPC enhances the flexibility and efficiency of communication between client and server, enabling the development of scalable and responsive distributed systems. By

leveraging the streaming capabilities of gRPC, developers can build robust applications capable of handling complex data flows and real-time interactions.

Header Compression

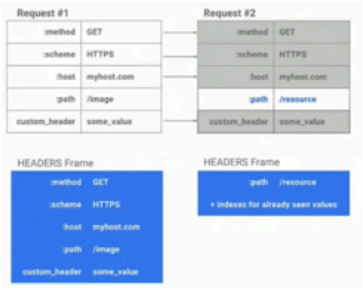
Header Compression

- Sometimes we might have encountered many cases where our http header is even bigger than our payload. And HTTP/2 fully agree with that. HTTP/2 has a very interesting strategy called HPack to handle that.
- HTTP/2 will map the header in both client and server side. From that, HTTP/2 will be able to know if the header contains the same value and will only send header value that is different with previous header.

FACULTY OF COMPUTER SCIENCE

Header Compression

- As seen in picture below, request 2 will only send path since other values are the exact same. And yes, this does cuts a lot of the payload and in turns pumping http/2 performance even more.



FACULTY OF COMPUTER SCIENCE

Header compression is a crucial feature of the gRPC architecture, enabled by the HTTP/2 protocol's innovative strategy known as HPack. In traditional HTTP communication, headers can often be larger than the payload itself, leading to increased overhead and reduced performance. HTTP/2 addresses this issue by employing efficient header compression techniques.

With HPack, HTTP/2 maps headers on both the client and server sides, allowing it to identify and eliminate redundant header information. When a request or response contains headers that are identical to those sent previously, HTTP/2 only transmits the differences, significantly reducing the amount of data transmitted over the network.

For example, if a client sends multiple requests with similar headers, HTTP/2 will only transmit the parts of the headers that differ between requests. This approach minimizes the size of the transmitted data and optimizes network utilization, ultimately improving performance and responsiveness.

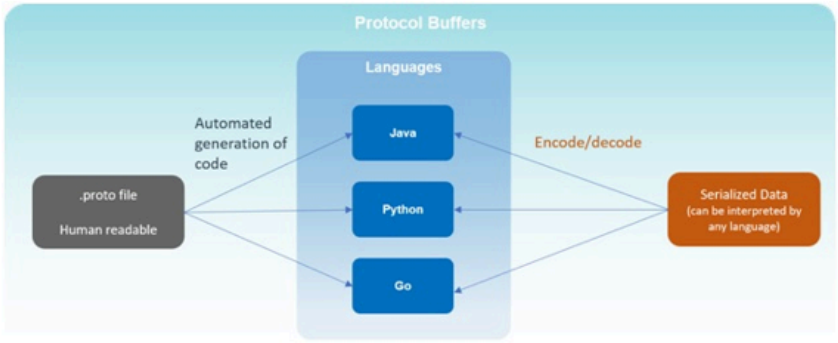
By leveraging header compression, gRPC can achieve higher efficiency and lower latency in communication between client and server applications. This optimization is particularly beneficial in scenarios where headers play a significant role, such as API requests with authentication tokens or metadata. Overall, header compression enhances the scalability and performance of gRPC-based systems, making it a critical component of the architecture.

Protocol Buffers

- Protocol buffers, or Protobuf, is Google's serialization/deserialization protocol that enables the easy definition of services and auto-generation of client libraries. gRPC uses this protocol as their Interface Definition Language (IDL) and serialization toolset. Its current version is proto3, which has the latest features and is easier to use.
- gRPC services and messages between clients and servers are defined in proto files. The Protobuf compiler, protoc, generates client and server code that loads the .proto file into the memory at runtime and uses the in-memory schema to serialize/deserialize the binary message. After code generation, each message is exchanged between the client and remote service.

FACULTY OF COMPUTER SCIENCE

Protocol Buffers



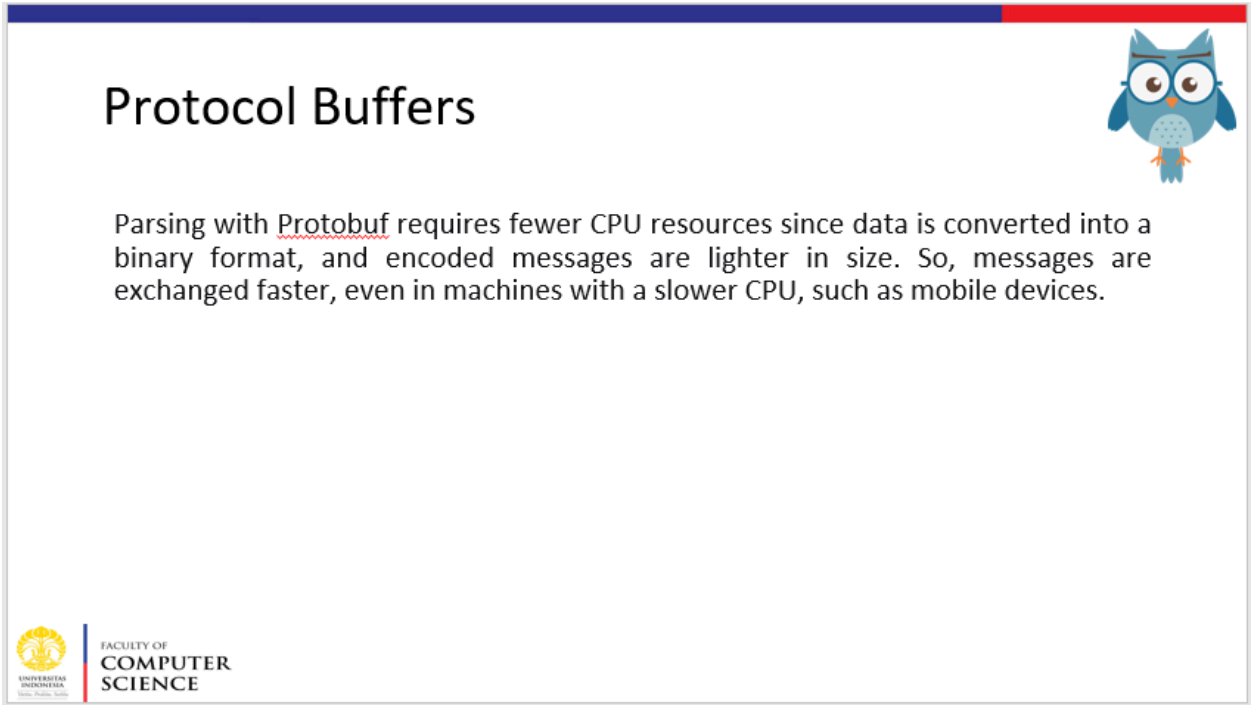
FACULTY OF COMPUTER SCIENCE

gRPC leverages Protocol Buffers (Protobuf) as its Interface Definition Language (IDL) and serialization/deserialization protocol. Protobuf, developed by Google, offers a streamlined

approach to defining services and message structures, allowing for the automatic generation of client libraries. The current version, proto3, introduces advanced features and enhances usability.

In gRPC, services and message formats are specified in .proto files using Protobuf syntax. These files serve as blueprints for defining the communication interface between clients and servers. Upon compilation using the Protobuf compiler (protoc), client and server code is generated based on the .proto file. At runtime, the generated code loads the schema defined in the .proto file into memory, enabling efficient serialization and deserialization of binary messages exchanged between client and server.

By utilizing Protobuf, gRPC benefits from efficient and compact message serialization, reducing bandwidth consumption and improving network performance. Additionally, Protobuf's IDL capabilities facilitate clear and concise service definitions, enhancing code maintainability and interoperability. Overall, Protobuf plays a crucial role in the gRPC architecture, enabling seamless communication and efficient data exchange between distributed systems.



Parsing with Protocol Buffers (Protobuf) in gRPC architecture demands fewer CPU resources due to its binary format. Encoded messages are lighter, leading to faster exchanges, even on devices with slower CPUs like mobile devices. This efficiency ensures optimal performance and responsiveness in communication between client and server applications. With Protobuf, gRPC achieves streamlined parsing and serialization, enabling efficient data transmission while minimizing resource consumption. As a result, gRPC-based systems can deliver

high-performance communication across diverse environments, without compromising speed or reliability.

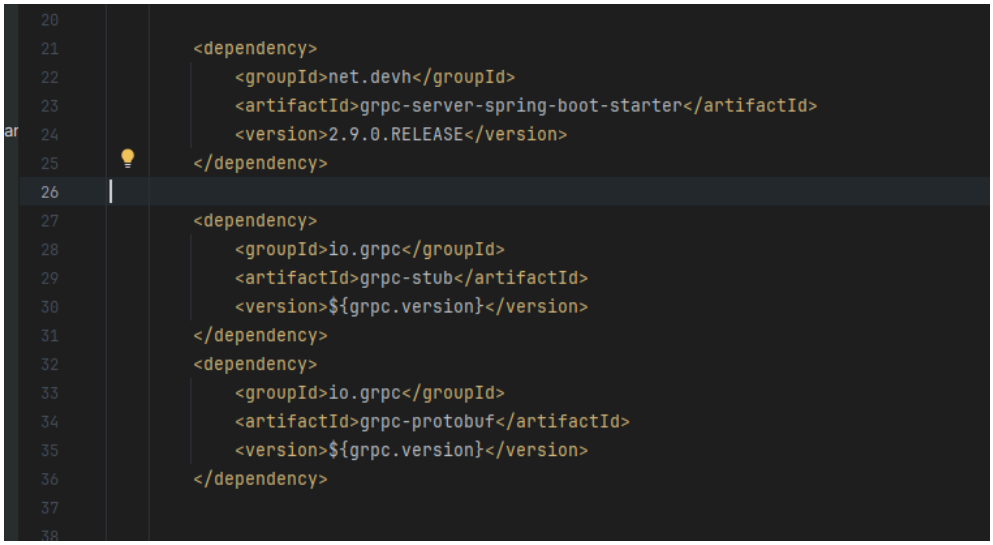
gRPC in Java

Source code:

https://github.com/muhammad-khadafi/java-grpc-server

https://github.com/muhammad-khadafi/java-grpc-client

1. Add gRPC dependencies



2. Add gRPC plugins

pom.xml (Spring-Boot-GRPC-Client-Example)

```
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>org.xolstice.maven.plugins</groupId>
            <artifactId>protobuf-maven-plugin</artifactId>
            <version>0.5.1</version>
            <configuration>
                <protocArtifact>
                    com.google.protobuf:protoc:3.6.1:exe:${os.detected.classifier}
                </protocArtifact>
                <pluginId>grpc-java</pluginId>
                <pluginArtifact>
                    io.grpc:protoc-gen-grpc-java:1.22.1:exe:${os.detected.classifier}
                </pluginArtifact>
                <protoSourceRoot>
                    ${basedir}/src/main/proto/
                </protoSourceRoot>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
```

3. Create protobuf file

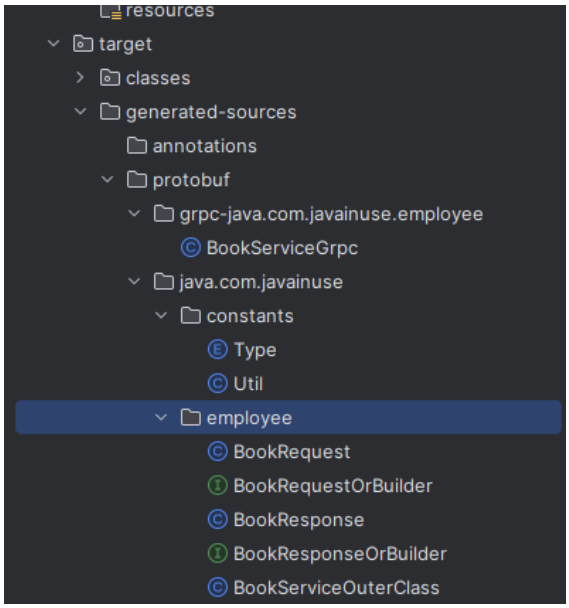
book-service.proto

```
syntax = "proto3";

import "constants/util.proto";

option java_multiple_files = true;
option java_package = "com.javainuse.employee";

message BookRequest{
  string book_id = 1;
}
message BookResponse {
  string book_id = 1;
  string name = 2;
  constants.Type type = 3;
}

service BookService {
  rpc getBook(BookRequest) returns (BookResponse) {};
}
```

book-service.proto util.proto

```
syntax = "proto3";

package constants;

option java_multiple_files = true;
option java_package = "com.javainuse.constants";

enum Type {
  FANTASY = 0;
  AUTOBIOGRAPHY = 1;
  HISTORY = 2;
}
```

4. Generate RPC stubs from protobuf



5. Create implementation of request, Create implementation of request in gRPC Client and get the response from gRPC server

```java
@Service
public class BookService {

    public void getBook() {

        ManagedChannel channel = ManagedChannelBuilder.forAddress( name: "localhost",  port: 8090).usePlaintext().build()

        BookServiceGrpc.BookServiceBlockingStub stub = BookServiceGrpc.newBlockingStub(channel);

        BookResponse bookResponse = stub.getBook(BookRequest.newBuilder().setBookId("1").build());

        System.out.println(bookResponse);

        channel.shutdown();
    }
}
```

6. Create implementation of service, Create implementation of service in gRPC Server and build the response (based on the Protobuf file)

```java
@GrpcService                                                                    ⚠ 2  ✓ 1
public class BookService extends BookServiceGrpc.BookServiceImplBase {

    /**
     * Unary operation to get the book based on book id
     *
     * @param request
     * @param responseObserver
     */
    1 usage
    @Override
    public void getBook(BookRequest request, StreamObserver<BookResponse> responseObserver) {

        // We have mocked the employee data.
        // In real world this should be fetched from database or from some other source
        BookResponse empResp = BookResponse.newBuilder().setBookId(request.getBookId()).setName("Ayo Belajar GRPC")
                .setType(Type.AUTOBIOGRAPHY).build();

        // set the response object
        responseObserver.onNext(empResp);

        // mark process is completed
        responseObserver.onCompleted();
    }
}
```

gRPC in Rust

Source code : https://github.com/muhammad-khadafi/rust-grpc-demo

1. Add dependencies

```
[dependencies]
tonic = "0.7"
prost = "0.10"
tokio = { version = "1.0", features = ["macros", "rt-multi-thread"] }
```

2. Create protobuf file

proto > hello.proto > ...

```
syntax = "proto3";
package hello;

service Greetings {
    rpc SayHello (ParamRequest) returns (MessageResponse);
}

message ParamRequest {
    string name = 1;
}

message MessageResponse {
    string message = 1;
}
```

3. Add protobuf to build

build.rs

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    tonic_build::compile_protos("proto/hello.proto")?;
    Ok(())
}
```

4. Create implementation of request, Create implementation of request in gRPC Client and get the response from gRPC server

src > client_hello.rs

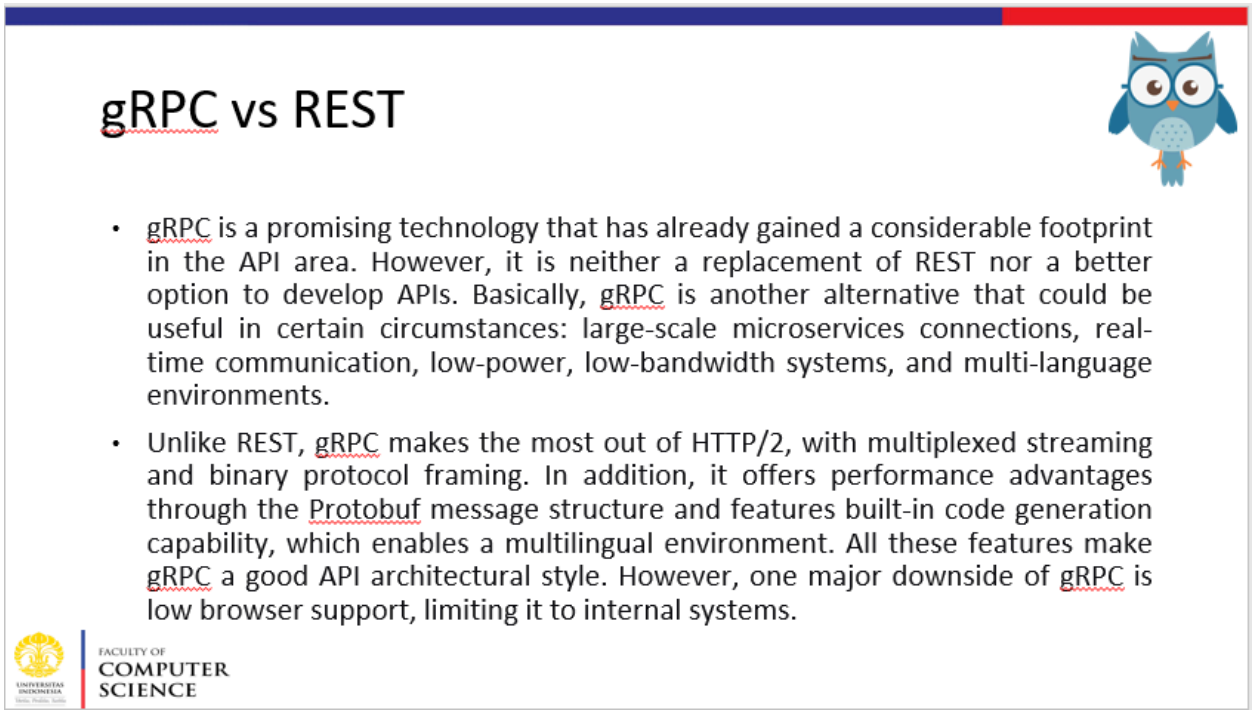
```rust
use hello::greetings_client::GreetingsClient;
use hello::ParamRequest;

pub mod hello {
    tonic::include_proto!("hello");
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut client = GreetingsClient::connect("http://[::1]:50052").await?;

    let request = tonic::Request::new(
        ParamRequest {
            name: "Budi".to_string()
        }
    );

    let response = client.say_hello(request).await?;

    println!("RESPONSE={:?}", response);

    Ok(())
}
```

5. Create implementation of service, create implementation of service in gRPC Server and build the response (based on the Protobuf file)

src > server_hello.rs

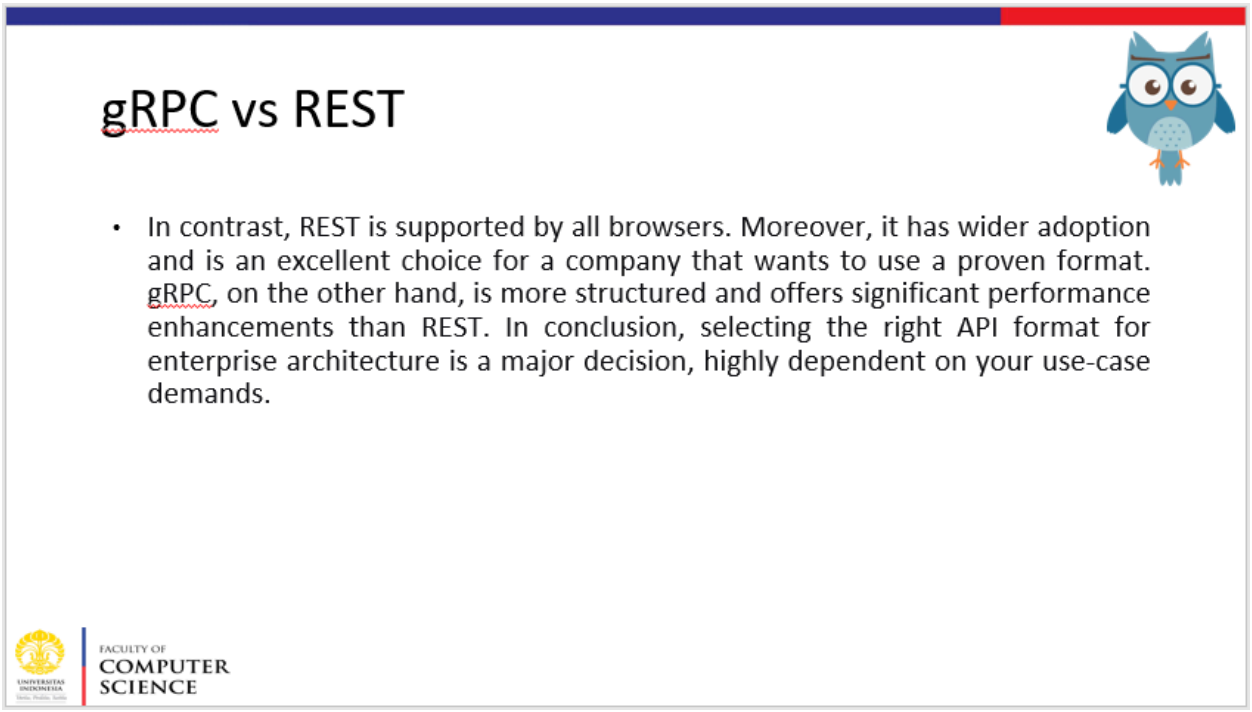
```rust
use tonic::{transport::Server, Request, Response, Status};

use hello::greetings_server::{Greetings, GreetingsServer};
use hello::{MessageResponse, ParamRequest};

pub mod hello {
    tonic::include_proto!("hello");
}

#[derive(Debug, Default)]
pub struct GreetingsService {}

#[tonic::async_trait]
impl Greetings for GreetingsService {
    async fn say_hello(
        &self,
        request: Request<ParamRequest>,
    ) -> Result<Response<MessageResponse>, Status> {
        println!("Got a request: {:?}", request);

        let req = request.into_inner();

        let reply = MessageResponse {
            message: format!("Hello {}.", req.name).into(),
        };

        Ok(Response::new(reply))
    }
}
```

gRPC vs REST



gRPC and REST represent two distinct approaches to building APIs, each with its own strengths and weaknesses. While gRPC has gained traction for its advanced features and performance benefits, it is not a direct replacement for REST but rather an alternative suited for specific use cases.

gRPC excels in scenarios such as large-scale microservices connections, real-time communication, and environments with low-power or low-bandwidth constraints. Leveraging HTTP/2 multiplexed streaming and binary protocol framing, gRPC offers superior performance compared to REST. Additionally, its use of Protobuf for message structure and built-in code generation capabilities enables seamless multilingual development.



However, one drawback of gRPC is its limited browser support, which restricts its applicability to internal systems. In contrast, REST enjoys wide browser support and broader adoption, making it a suitable choice for companies seeking a proven and widely accepted API format.

Ultimately, the decision between gRPC and REST hinges on the specific requirements and constraints of the enterprise architecture. While gRPC offers enhanced performance and structure, REST remains a reliable option with broader compatibility. Understanding the unique demands of the use case is essential in determining the most suitable API format for a given scenario.

This is a summary comparison of gRPC and REST

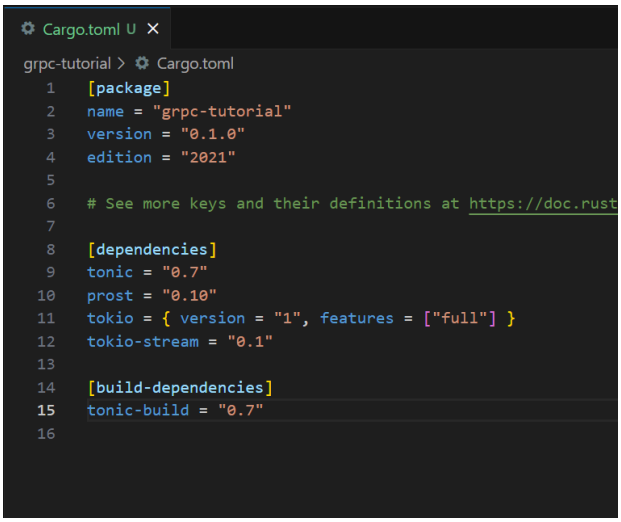
	gRPC	REST
Data format	Uses Protobuf to encode data in binary form	Uses plain-text data formats (JSON and XML)
Data validation	Automatically validates every message against the API contract	Requires an extra validation step on JSON data
Communication pattern	Supports unary communication, as well as server streaming, client streaming, and bidirectional streaming	Follows a unary request/response cycle
Design pattern	Defines callable functions on the server, which the client can invoke as if they were local	Uses HTTP methods to grant access to resources through dedicated endpoints
Code generation	Supports code generation in many programming languages	No native support for code generation
Primary use case	Microservice architectures	Public APIs or other APIs where ease of use is a priority

Tutorial - Rust gRPC

1. Create a new rust project using cargo, use the command::

   ```
   cargo new grpc-tutorial
   ```

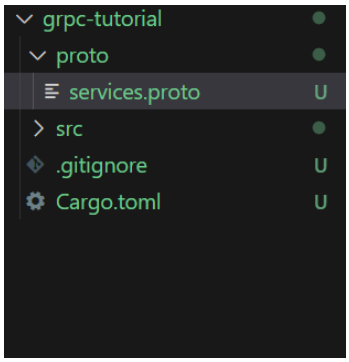
2. Add 4 dependencies and 1 build dependency to Cargo.toml for the needs of the gRPC client and server. The 4 dependencies are tonic, prost, tokio and tokio stream.



- Tonic is a gRPC framework written using Rust. Tonic is built on top of Hyper (HTTP library) and Tower (middleware library), leveraging Rust's modern async/await features to create efficient and expressive RPC handling.
- Prost is an encoder and decoder library for Protobuf written in Rust. This converts .proto files (Protobuf definitions) into Rust code that can be used for data serialization and deserialization.
- Tokio is a comprehensive asynchronous runtime for Rust, enabling non-blocking, event-driven development of high-scale applications. Tokio leverages the Rust futures and tasks ecosystem to run multiple tasks simultaneously with low overhead.
- Tokio-Stream is part of the Tokio ecosystem that provides abstractions for working with streams in an asynchronous context. This library allows the development of code that can handle continuously arriving data streams (such as web sockets, or data streams from databases).

- Tonic build is a tool used to generate the Rust code required to process RPC (Remote Procedure Call) calls using gRPC (Google Remote Procedure Call) in Rust. This is especially useful for building gRPC servers and clients in Rust.

3. Create a proto folder and **services.proto** file in that folder

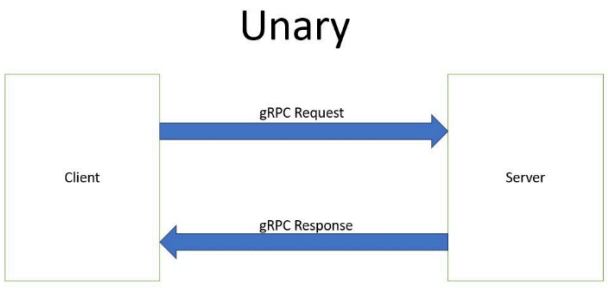


*****

There are several data transmission mechanisms that can be carried out by gRPC : unary, client streaming, server streaming and bi-directional streaming..

- Unary, Unary gRPC is a type of communication protocol that allows client-server interactions where the client sends a single request to the server and gets a single response back. This means that the client sends one piece of data to the server and waits for a response.

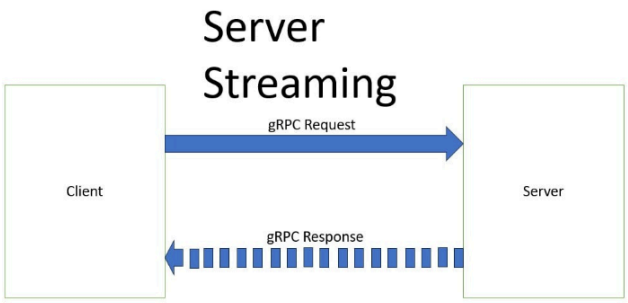
  For example, if you want to fetch a single item from a database, authenticate a user, or perform a calculation and obtain the result, unary gRPC can be a good choice.



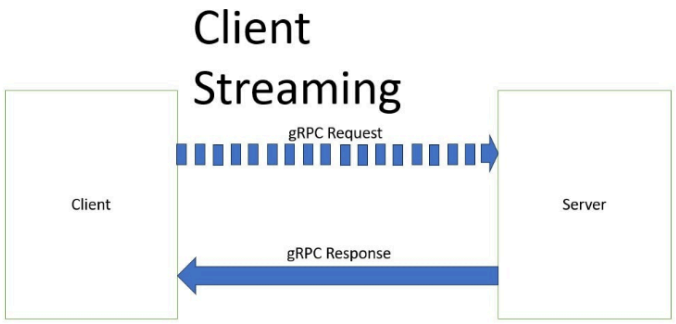
- Server Streaming, Server streaming gRPC is a communication protocol that allows the server to send a stream of responses to the client. This pattern is useful in scenarios where the server needs to push a large amount of data or a continuous stream of updates

to the client.

For example, it can be used for real-time updates like stock market prices, weather updates, news feeds, or even sending a large file in chunks.

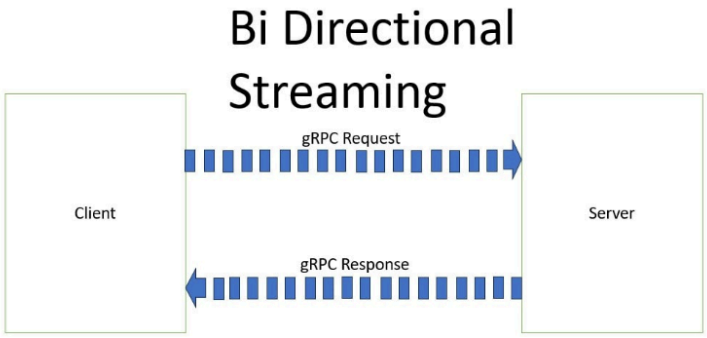


- Client streaming, Client streaming gRPC is a communication pattern in which the client sends multiple messages to the server using a single gRPC connection. It allows the client to initiate a stream of data and send chunks of information to the server, which processes the data and responds accordingly. One common use case for Client streaming gRPC is when the client needs to send a large amount of data in small chunks to the server. It can be useful in scenarios where the client is continuously generating or collecting data that needs to be processed or analyzed on the server side.



- Bidirectional streaming gRPC is a communication pattern in which both the client and server can send multiple messages to each other in a continuous stream. It allows real-time, two-way communication between the client and server. This pattern is commonly used in scenarios where there is a need for ongoing, interactive communication. For example, in a chat application, bidirectional streaming gRPC can be used to send and receive messages in real-time between the client and server. It can also

be used for real-time analytics, where data is continuously sent from server to client and vice versa.



In this tutorial, we will simulate unary data transmission mechanism, server streaming, and bi-directional streaming using gRPC. For this purpose, we will create 3 services in the protobuf file:

- **PaymentService**: The client will make a single request for payment submission, and the server will provide a single response for that payment (**unary**).
- **TransactionService**: The client will make a single request to retrieve transaction history data, and since transaction history data is considered as large data, the server will split the response into multiple transmissions (**server streaming**).
- **ChatService**: The client and server will engage in interactive communication to conduct a conversation (**bi-directional streaming**).

*****

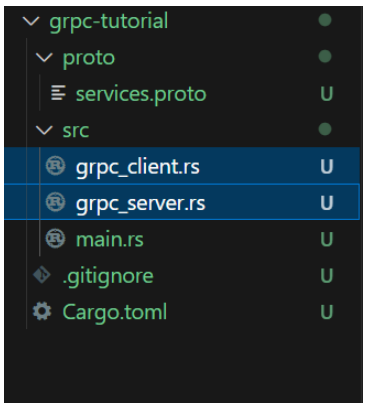
4. Create 3 services in the protobuf file (**services.proto**)

```
grpc-tutorial > proto > services.proto > ...
1   syntax = "proto3";
2
3   package services;
4
5   // Service for processing payments
6   service PaymentService {
7     rpc ProcessPayment(PaymentRequest) returns (PaymentResponse);
8   }
9
10  // Service for checking transaction history
11  service TransactionService {
12    rpc GetTransactionHistory(TransactionRequest) returns (stream TransactionResponse);
13  }
14
15  // Service for chatting with a customer service bot
16  service ChatService {
17    rpc Chat(stream ChatMessage) returns (stream ChatMessage);
18  }
19
```

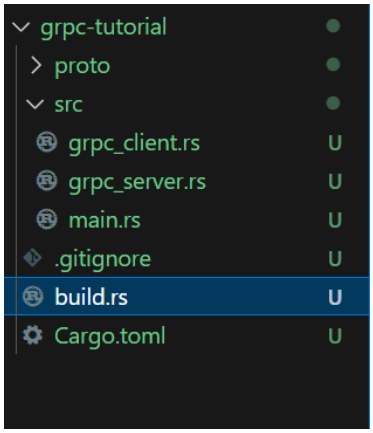
5. Still in the same file, define each request and response parameter for the services

```
grpc-tutorial > proto > services.proto > ...
19
20  message PaymentRequest {
21    string user_id = 1;
22    double amount = 2;
23  }
24
25  message PaymentResponse {
26    bool success = 1;
27  }
28
29  message TransactionRequest {
30    string user_id = 1;
31  }
32
33  message TransactionResponse {
34    string transaction_id = 1;
35    string status = 2;
36    double amount = 3;
37    string timestamp = 4;
38  }
39
40  message ChatMessage {
41    string user_id = 1;
42    string message = 2;
43  }
44
```

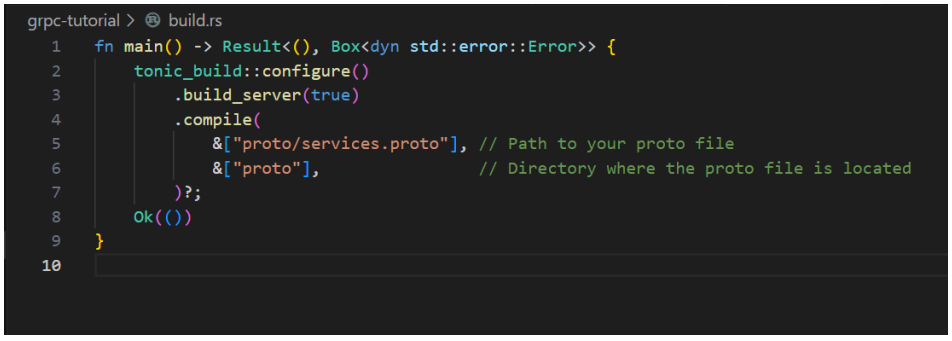
6. Create **grpc_server.rs** and **grpc_client.rs** files in **/src** to implement the 3 services above.



7. Create **build.rs** file to put the protobuf file build configuration



8. Write a protobuf build configuration in the **build.rs** file



9. **Commit your changes to Git with the message: “Create base skeleton of Rust gRPC.”**

Payment Service Implementation (Unary)

10. We will start by implementing the payment service in **grpc_server.rs** first
11. Import the tonic package

```
use tonic::{transport::Server, Request, Response, Status};
```

12. Defines a Rust module named services which is made public, allowing it to be accessible from outside the crate where it is defined. Within this module, the `tonic::include_proto!("services")` macro is used. This macro automatically generates Rust code from gRPC protocol definitions contained in a .proto file specified by the argument "services". Essentially, this setup integrates gRPC service definitions with Rust code, enabling the implementation of the defined services and their methods using tonic. This approach simplifies the development of networked applications by auto-generating necessary boilerplate code for communication over gRPC.

```
pub mod services {
    tonic::include_proto!("services");
}
```

13. Use the services needed for the payment service

```
use services::{payment_service_server::{PaymentService, PaymentServiceServer}, PaymentRequest, PaymentResponse};
```

14. Define the Struct

```
#[derive(Default)]
pub struct MyPaymentService {}
```

15. Implement the service trait. You must ensure that MyPaymentService conforms to the PaymentService trait generated by tonic from your .proto definitions. This involves implementing each RPC method declared in the proto file.

```rust
#[tonic::async_trait]
impl PaymentService for MyPaymentService {
    async fn process_payment(
        &self,
        request: Request<PaymentRequest>,
    ) -> Result<Response<PaymentResponse>, Status> {
        println!("Received payment request: {:?}", request);

        // Process the request and return a response
        // This example immediately returns a successful result for demonstration purposes
        Ok(Response::new(PaymentResponse { success: true }))
    }
}
```

16. Still in **grpc_server.rs**, Implement PaymentService in main function, define the port and start the server

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let addr = "[::1]:50051".parse()?;
    let payment_service = MyPaymentService::default();

    Server::builder()
        .add_service(PaymentServiceServer::new(payment_service))
        .serve(addr)
        .await?;

    Ok(())
}
```

17. Code on the server is complete, we move on to the ***grpc_client.rs*** file

18. Set up the proto definitions

```rust
pub mod services {
    tonic::include_proto!("services");
}
```

19. Import necessary modules

```rust
use services::{payment_service_client::PaymentServiceClient, PaymentRequest};
```

20. Create the main Function and Initialize the Client

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut client = PaymentServiceClient::connect("http://[::1]:50051").await?;

}
```

21. Create and send the request

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut client = PaymentServiceClient::connect("http://[::1]:50051").await?;
    let request = tonic::Request::new(PaymentRequest {
        user_id: "user_123".to_string(),
        amount: 100.0,
    });

    let response = client.process_payment(request).await?;
    println!("RESPONSE={:?}", response.into_inner());


    Ok(())
}
```

- Construct a payment request using the PaymentRequest struct. This struct should match the proto definition and include all necessary fields
- Send the payment request to the server using the process_payment method of the client. Handle the response asynchronously
- The process_payment method sends the PaymentRequest to the server and awaits a PaymentResponse. The response is unwrapped and printed to the console. The into_inner() method is used to extract the actual response from the tonic's wrapper.
- The function signature Result<(), Box<dyn std::error::Error>> indicates that the function will return a result that, on error, boxes any error implementing the Error trait, providing flexibility in error handling.

22. Create binary profile for **grpc_server.rs** and **grpc_client.rs** in ***Cargo.toml***

```toml
grpc-tutorial > Cargo.toml
[package]
name = "grpc-tutorial"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/

[[bin]]
name = "grpc-server"
path = "src/grpc_server.rs"

[[bin]]
name = "grpc-client"
path = "src/grpc_client.rs"

[dependencies]
tonic = "0.7"
```

23. Run gRPC server for test the payment service with command :

    `cargo run --bin grpc-server`

24. Run gRPC client for test the payment service with command :

    `cargo run --bin grpc-client`

25. **Commit your changes to Git with the message: “Payment Service Implementation”**

Transaction Service Implementation (Server Streaming)

26. Next, we will implement Transaction Service, we will start from the grpc_server.rs file first
27. Use necessary library

```
use tonic::{transport::Server, Request, Response, Status};
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;
use tokio::sync::mpsc::{Receiver, Sender};
```

28. Use necessary services

```
use services::{payment_service_server::{PaymentService, PaymentServiceServer}, PaymentRequest, PaymentResponse,
    transaction_service_server::{TransactionService, TransactionServiceServer}, TransactionRequest, TransactionResponse};
```

29. Define a struct that will serve as the service implementation

```
#[derive(Default)]
pub struct MyTransactionService {}
```

Here, MyTransactionService is a simple Rust struct with a Default derive. The Default trait provides a default instance of this struct, which is useful when initializing the service without requiring any special configuration.

30. Implement the TransactionService trait for MyTransactionService. This trait will be generated from the proto file, which should define the gRPC service and the methods it supports. In this case, the method is get_transaction_history.

```rust
#[tonic::async_trait]
impl TransactionService for MyTransactionService {
    type GetTransactionHistoryStream = ReceiverStream<Result<TransactionResponse, Status>>;

    async fn get_transaction_history(
        &self,
        request: Request<TransactionRequest>,
    ) -> Result<Response<Self::GetTransactionHistoryStream>, Status> {
        println!("Received transaction history request: {:?}", request);
        let (tx, rx): (Sender<Result<TransactionResponse, Status>>, Receiver<Result<TransactionResponse, Status>>) = mpsc::channel(4);

        tokio::spawn(async move {
            for i in 0..30 {  // Simulate sending 30 transaction records
                if tx.send(Ok(TransactionResponse {
                    transaction_id: format!("trans_{}", i),
                    status: "Completed".to_string(),
                    amount: 100.0,
                    timestamp: "2022-01-01T12:00:00Z".to_string(),
                })).await.is_err() {
                    break;
                }
                if i % 10 == 9 {
                    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
                }
            }
        });

        Ok(Response::new(ReceiverStream::new(rx)))
    }
}
```

- **Method Signature:** The get_transaction_history method is an asynchronous function that takes a `Request<TransactionRequest>` and returns a Result containing either a Response of a ReceiverStream or a Status error.
- **Request Handling:** The method begins by logging the incoming request. This is crucial for debugging and ensuring that the service receives requests correctly.
- **Channel Setup:** It sets up a channel using `tokio::sync::mpsc` with a buffer size of 4. This channel is used to send transaction responses from a producer task (inside the same function) to the consumer/client (gRPC framework that sends these responses to the client).
- **Stream Generation:**
    - A new task is spawned to simulate the generation of transaction records. This is done to mimic a real-world scenario where transaction records might be fetched from a database or another service.
    - The loop within the task iterates 30 times to simulate sending 30 transaction records. Each record is sent through the channel, and every tenth record, a sleep is induced to simulate processing delay with server streaming mechanism.

- **Stream Handling:** The ReceiverStream is a utility from `tokio_stream::wrappers` that converts a tokio receiver into a stream that tonic can work with. This stream is then wrapped in a Response and returned.

31. Still in **grpc_server.rs**, Implement TransactionService in main function.

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let addr = "[::1]:50051".parse()?;
    let payment_service = MyPaymentService::default();
    let transaction_service = MyTransactionService::default();

    Server::builder()
        .add_service(PaymentServiceServer::new(payment_service))
        .add_service(TransactionServiceServer::new(transaction_service))
        .serve(addr)
        .await?;

    Ok(())
}
```

32. Code on the server is complete, we move on to the ***grpc_client.rs*** file

33. Import necessary modules

```rust
use services::{payment_service_client::PaymentServiceClient, PaymentRequest,
    transaction_service_client::TransactionServiceClient, TransactionRequest};
```

34. Initialize the client, initialize the request and execute `get_transaction_history`

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut client = PaymentServiceClient::connect("http://[::1]:50051").await?;
    let request = tonic::Request::new(PaymentRequest {
        user_id: "user_123".to_string(),
        amount: 100.0,
    });

    let response = client.process_payment(request).await?;
    println!("RESPONSE={:?}", response.into_inner());

    let mut transaction_client = TransactionServiceClient::connect("http://[::1]:50051").await?;
    let request = tonic::Request::new(TransactionRequest {
        user_id: "user_123".to_string(),
    });

    let mut stream = transaction_client.get_transaction_history(request).await?.into_inner();
    while let Some(transaction) = stream.message().await? {
        println!("Transaction: {:?}", transaction);
    }

    Ok(())
}
```

- **Create Client Connection:** `TransactionServiceClient::connect("http://[::1]:50051").await?;` creates a client for the TransactionService and attempts to connect to the

service running on localhost ([::1]) at port 50051. The await keyword is used to asynchronously wait for the connection to be established, and the ? operator propagates any errors that might occur during the connection process.

- **Create and Send Request:**
  `tonic::Request::new(TransactionRequest { user_id: "user_123".to_string(), });` constructs a new request for transaction history. This request includes a TransactionRequest object containing the user ID "user_123".
- **Receive and Process Stream:**
  `transaction_client.get_transaction_history(request).await? .into_inner();` sends the transaction history request to the server and awaits the streaming response. The `into_inner()` method extracts the actual stream from the response wrapper.
  `while let Some(transaction) = stream.message().await? { println!("Transaction: {:?}", transaction); }` iteratively retrieves messages from the stream. Each message represents a transaction record. It uses await to asynchronously wait for each message, and if a message is received, it prints the transaction details. The loop continues until there are no more messages in the stream (i.e., the stream is exhausted).

35. Run gRPC server for test the transaction service with command : `cargo run --bin grpc-server`
36. Run gRPC client for test the transaction service with command : `cargo run --bin grpc-client`
37. **Commit your changes to Git with the message: “Transaction Service Implementation”**

Chat Service Implementation (Bi-Directional Streaming)

38. Next, we will implement the Chat Service, we will start from the **grpc_server.rs** file first
39. No new libraries need to be added to **grpc_server.rs**

```
use tonic::{transport::Server, Request, Response, Status};
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;
use tokio::sync::mpsc::{Receiver, Sender};
```

40. Add necessary modules

```
use services::{payment_service_server::{PaymentService, PaymentServiceServer}, PaymentRequest, PaymentResponse,
    transaction_service_server::{TransactionService, TransactionServiceServer}, TransactionRequest, TransactionResponse,
    chat_service_server::{ChatService, ChatServiceServer}, ChatMessage};
```

41. Define a struct that will serve as the service implementation

```
#[derive(Default)]
pub struct MyChatService {}
```

42. Service trait implementation

```
#[tonic::async_trait]
impl ChatService for MyChatService {
    type ChatStream = ReceiverStream<Result<ChatMessage, Status>>;

    async fn chat(
        &self,
        request: Request<tonic::Streaming<ChatMessage>>,
    ) -> Result<Response<Self::ChatStream>, Status> {
        let mut stream = request.into_inner();
        let (tx, rx) = mpsc::channel(10);

        tokio::spawn(async move {
            while let Some(message) = stream.message().await.unwrap_or_else(|_| None) {
                println!("Received message: {:?}", message);
                let reply = ChatMessage {
                    user_id: message.user_id.clone(),
                    message: format!("Terima kasih telah melakukan chat kepada CS virtual,
                    Pesan anda akan dibalas pada jam kerja. pesan anda : {}", message.message),
                };

                tx.send(Ok(reply)).await.unwrap_or_else(|_| {});
            }
        });

        Ok(Response::new(ReceiverStream::new(rx)))
    }
}
```

Streaming Request and Response

- Request Handling: The method signature async fn chat indicates it's an asynchronous function, essential for handling potentially long-lived communication in a chat session. The input request wraps a streaming object,

`tonic::Streaming<ChatMessage>`, allowing the server to asynchronously receive messages sent by the client over time.

- Extracting the Stream: `let mut stream = request.into_inner();` converts the request into its inner streaming component, enabling the server to iterate over incoming messages as they arrive.

Channel Setup for Responses

- Creating a Channel: `let (tx, rx) = mpsc::channel(10);` sets up a multi-producer, single-consumer (MPSC) channel with a buffer size of 10. This channel is used to send responses back to the client. The choice of 10 as the buffer size is arbitrary but should be optimized based on expected traffic and performance testing.

Processing Incoming Messages

- Asynchronous Processing: The `tokio::spawn` function is used to handle incoming messages in a new asynchronous task, which allows the main task to return immediately without waiting for all chat messages to be processed.
- Message Handling Loop:
  `while let Some(message) = stream.message().await.unwrap_or_else(|_| None)` continuously tries to receive new messages from the client, handling each message as it arrives. The use of `unwrap_or_else(|_| None)` ensures that any errors in message reception do not crash the service; instead, they stop the processing loop gracefully.

Message Response

- Crafting Replies: For each received message, a reply is constructed using the received data. This implementation simply echoes back a predefined message appended to the original message, which serves as an acknowledgment or placeholder for more complex logic.
- Sending the Response: `tx.send(Ok(reply)).await.unwrap_or_else(|_| {});` sends the crafted message back through the channel. The use of await here ensures that the send operation does not block the current task, waiting instead for the channel to be ready to receive the message.

Streaming Response to Client

- Converting Receiver to Stream:

`Ok(Response::new(ReceiverStream::new(rx)))` converts the receiving end of the channel (rx) into a ReceiverStream that can be returned as part of a gRPC stream response. This stream is then wrapped in a Response object and returned to the client, allowing it to receive the responses asynchronously as they are sent.

43. Still in **grpc_server.rs**, Implement ChatServices in main function.

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let addr = "[::1]:50051".parse()?;
    let payment_service = MyPaymentService::default();
    let transaction_service = MyTransactionService::default();
    let chat_service = MyChatService::default();

    Server::builder()
        .add_service(PaymentServiceServer::new(payment_service))
        .add_service(TransactionServiceServer::new(transaction_service))
        .add_service(ChatServiceServer::new(chat_service))
        .serve(addr)
        .await?;

    Ok(())
}
```

44. Code on the server is complete, we move on to the **grpc_client.rs** file

45. Use necessary package from tokio and tonic

```
use tonic::transport::Channel;
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;
use tokio::sync::mpsc::{Sender, Receiver};
use tokio::io::{self, AsyncBufReadExt};
```

46. Add necessary services

```
use services::{payment_service_client::PaymentServiceClient, PaymentRequest,
    transaction_service_client::TransactionServiceClient, TransactionRequest,
    chat_service_client::ChatServiceClient, ChatMessage};
```

47. Creating Channel and Client:

```
let channel = Channel::from_static("http://[::1]:50051").connect().await?;
let mut client = ChatServiceClient::new(channel);
```

- `let channel = Channel::from_static("http://[::1]:50051").connect().await?;:` This line establishes a gRPC channel connecting to the server running on localhost at port 50051. Students will need to replace this with the appropriate server address if it differs.

- `let mut client = ChatServiceClient::new(channel);`: Initializes a new gRPC client for the ChatService using the previously created channel.

48. Setting Up Communication Channels:

```
let (tx, rx): (Sender<ChatMessage>, Receiver<ChatMessage>) = mpsc::channel(32);
```

Creates a bounded multi-producer, single-consumer (MPSC) channel to communicate between the main thread and the background task. This channel allows the main thread to send messages to the background task for processing.

49. Background Task for Sending Messages

```
tokio::spawn(async move {
    let stdin = io::stdin();
    let mut reader = io::BufReader::new(stdin).lines();

    while let Ok(Some(line)) = reader.next_line().await {
        if line.trim().is_empty() {
            continue;
        }
        let message = ChatMessage {
            user_id: "user_123".to_string(),
            message: line,
        };

        if tx.send(message).await.is_err() {
            eprintln!("Failed to send message to server.");
            break;
        }
    }
});
```

- Spawning Background Task:

  `tokio::spawn(async move { ... }`: Spawns a background task using tokio::spawn. This task continuously reads input from the console and sends it to the server.

- Reading User Input:

  `let stdin = io::stdin(); let mut reader = io::BufReader::new(stdin).lines();`: Initializes a buffered reader to read input from the console.

  `while let Ok(Some(line)) = reader.next_line().await { ... }`: Loops to continuously read input from the console.

- Sending Messages to Server:

  `let message = ChatMessage { user_id: "user_123".to_string(), message: line, };`: Constructs a ChatMessage from the input line.

  `if tx.send(message).await.is_err() { ... }`: Sends the constructed message to the server via the channel. If an error occurs during sending, it prints an error message and breaks the loop.

50. Chat with the server

```
let request = tonic::Request::new(ReceiverStream::new(rx));
let mut response_stream = client.chat(request).await?.into_inner();

while let Some(response) = response_stream.message().await? {
    println!("Server says: {:?}", response);
}
```

- Creating Request with Message Stream:

  `let request = tonic::Request::new(ReceiverStream::new(rx));:` Constructs a gRPC request with a streaming message containing messages received from the background task.

- Initiating Chat with Server:

  `let mut response_stream = client.chat(request).await?.into_inner();:` Initiates a chat with the server by calling the chat method of the ChatService client. This method returns a stream of responses from the server.

- Processing Server Responses:

  `while let Some(response) = response_stream.message().await? { ... }:` Loops to continuously receive and print responses from the server.

51. Run gRPC server for test the chat service with command : `cargo run --bin grpc-server`
52. Run gRPC client for test the chat service with command : `cargo run --bin grpc-client`
53. Type a message in the grpc client console and you will immediately get a chat response from the server.

```
tidak bisa login
Server says: ChatMessage { user_id: "user_123", message: "Terima kasih telah melakukan chat kepada CS virtual, Pesan anda akan dibalas pada j
am kerja. pesan anda : tidak bisa login" }
lupa password
Server says: ChatMessage { user_id: "user_123", message: "Terima kasih telah melakukan chat kepada CS virtual, Pesan anda akan dibalas pada j
am kerja. pesan anda : lupa password" }
```

54. **Commit your changes to Git with the message: “Chat Service Implementation”**

Complete snippet of code in ***grpc_server.rs*** :

grpc-tutorial > src > grpc_server.rs

```rust
use tonic::{transport::Server, Request, Response, Status};
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;
use tokio::sync::mpsc::{Receiver, Sender};

pub mod services {
    tonic::include_proto!("services");
}

use services::{payment_service_server::{PaymentService, PaymentServiceServer}, PaymentRequest, PaymentResponse,
    transaction_service_server::{TransactionService, TransactionServiceServer}, TransactionRequest, TransactionResponse,
    chat_service_server::{ChatService, ChatServiceServer}, ChatMessage};

#[derive(Default)]
pub struct MyPaymentService {}

#[tonic::async_trait]
impl PaymentService for MyPaymentService {
    async fn process_payment(
        &self,
        request: Request<PaymentRequest>,
    ) -> Result<Response<PaymentResponse>, Status> {
        println!("Received payment request: {:?}", request);

        // Process the request and return a response
        // This example immediately returns a successful result for demonstration purposes
        Ok(Response::new(PaymentResponse { success: true }))
    }
}

#[derive(Default)]
pub struct MyTransactionService {}

#[tonic::async_trait]
impl TransactionService for MyTransactionService {
    type GetTransactionHistoryStream = ReceiverStream<Result<TransactionResponse, Status>>;

    async fn get_transaction_history(
        &self,
        request: Request<TransactionRequest>,
    ) -> Result<Response<Self::GetTransactionHistoryStream>, Status> {
        println!("Received transaction history request: {:?}", request);
        let (tx, rx): (Sender<Result<TransactionResponse, Status>>, Receiver<Result<TransactionResponse, Status>>) = mpsc::channel(4);

        tokio::spawn(async move {
            for i in 0..30 {  // Simulate sending 30 transaction records
                if tx.send(Ok(TransactionResponse {
                    transaction_id: format!("trans_{}", i),
                    status: "Completed".to_string(),
                    amount: 100.0,
                    timestamp: "2022-01-01T12:00:00Z".to_string(),
                })).await.is_err() {
                    break;
                }
                if i % 10 == 9 {
                    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
                }
            }
        });

        Ok(Response::new(ReceiverStream::new(rx)))
    }
}

```

```rust

#[derive(Default)]
pub struct MyChatService {}

#[tonic::async_trait]
impl ChatService for MyChatService {
    type ChatStream = ReceiverStream<Result<ChatMessage, Status>>;

    async fn chat(
        &self,
        request: Request<tonic::Streaming<ChatMessage>>,
    ) -> Result<Response<Self::ChatStream>, Status> {
        let mut stream = request.into_inner();
        let (tx, rx) = mpsc::channel(10);

        tokio::spawn(async move {
            while let Some(message) = stream.message().await.unwrap_or_else(|_| None) {
                println!("Received message: {:?}", message);
                let reply = ChatMessage {
                    user_id: message.user_id.clone(),
                    message: format!("Terima kasih telah melakukan chat kepada CS virtual, Pesan anda akan dibalas pada jam kerja. pesan 
                };

                tx.send(Ok(reply)).await.unwrap_or_else(|_| {});
            }
        });

        Ok(Response::new(ReceiverStream::new(rx)))
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let addr = "[::1]:50051".parse()?;
    let payment_service = MyPaymentService::default();
    let transaction_service = MyTransactionService::default();
    let chat_service = MyChatService::default();

    Server::builder()
        .add_service(PaymentServiceServer::new(payment_service))
        .add_service(TransactionServiceServer::new(transaction_service))
        .add_service(ChatServiceServer::new(chat_service))
        .serve(addr)
        .await?;

    Ok(())
}

```

Complete snippet of code in ***grpc_client.rs*** :

grpc-tutorial > src > grpc_client.rs

```rust
use tonic::transport::Channel;
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;
use tokio::sync::mpsc::{Sender, Receiver};
use tokio::io::{self, AsyncBufReadExt};


pub mod services {
    tonic::include_proto!("services");
}

use services::{payment_service_client::PaymentServiceClient, PaymentRequest,
    transaction_service_client::TransactionServiceClient, TransactionRequest,
    chat_service_client::ChatServiceClient, ChatMessage};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut client = PaymentServiceClient::connect("http://[::1]:50051").await?;
    let request = tonic::Request::new(PaymentRequest {
        user_id: "user_123".to_string(),
        amount: 100.0,
    });

    let response = client.process_payment(request).await?;
    println!("RESPONSE={:?}", response.into_inner());

    let mut transaction_client = TransactionServiceClient::connect("http://[::1]:50051").await?;
    let request = tonic::Request::new(TransactionRequest {
        user_id: "user_123".to_string(),
    });

    let mut stream = transaction_client.get_transaction_history(request).await?.into_inner();
    while let Some(transaction) = stream.message().await? {
        println!("Transaction: {:?}", transaction);
    }

    let channel = Channel::from_static("http://[::1]:50051").connect().await?;
    let mut client = ChatServiceClient::new(channel);
    let (tx, rx): (Sender<ChatMessage>, Receiver<ChatMessage>) = mpsc::channel(32);

    tokio::spawn(async move {
        let stdin = io::stdin();
        let mut reader = io::BufReader::new(stdin).lines();

        while let Ok(Some(line)) = reader.next_line().await {
            if line.trim().is_empty() {
                continue;
            }
            let message = ChatMessage {
                user_id: "user_123".to_string(),
                message: line,
            };

            if tx.send(message).await.is_err() {
                eprintln!("Failed to send message to server.");
                break;
            }
        }
    });

    let request = tonic::Request::new(ReceiverStream::new(rx));
    let mut response_stream = client.chat(request).await?.into_inner();

    while let Some(response) = response_stream.message().await? {
        println!("Server says: {:?}", response);
    }

    Ok(())
}
```

Reflection

1. What are the key differences between unary, server streaming, and bi-directional streaming RPC (Remote Procedure Call) methods, and in what scenarios would each be most suitable?
2. What are the potential security considerations involved in implementing a gRPC service in Rust, particularly regarding authentication, authorization, and data encryption?
3. What are the potential challenges or issues that may arise when handling bidirectional streaming in Rust gRPC, especially in scenarios like chat applications?
4. What are the advantages and disadvantages of using the `tokio_stream::wrappers::ReceiverStream` for streaming responses in Rust gRPC services?
5. In what ways could the Rust gRPC code be structured to facilitate code reuse and modularity, promoting maintainability and extensibility over time?
6. In the **MyPaymentService** implementation, what additional steps might be necessary to handle more complex payment processing logic?
7. What impact does the adoption of gRPC as a communication protocol have on the overall architecture and design of distributed systems, particularly in terms of interoperability with other technologies and platforms?
8. What are the advantages and disadvantages of using HTTP/2, the underlying protocol for gRPC, compared to HTTP/1.1 or HTTP/1.1 with WebSocket for REST APIs?
9. How does the request-response model of REST APIs contrast with the bidirectional streaming capabilities of gRPC in terms of real-time communication and responsiveness?
10. What are the implications of the schema-based approach of gRPC, using Protocol Buffers, compared to the more flexible, schema-less nature of JSON in REST API payloads?

Please write your reflection on the README.md file.

Grading Scheme

Scale

All components will be scored on scale 0-4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting.

Components

- 40% - Commits (Tutorial)
- 20% - Reflections
- 40% - Group Project progress (grading scheme made by teaching assistant).

For **Group Project progress**, it is claimable by attending weekly demonstrations to the teaching assistant team. To claim a score for a topic, students need to prove they have implemented a certain week's material as individual work in the Group Project.

Rubrics

	Score 4	Score 3	Score 2	Score 1
Commits (Tutorial)	100% of the commits are correct according to the tutorial.	>= 75% of the commits are correct.	>=50% of the commits are correct.	<50% of the commits are correct
Reflections	The description is sound, and the analysis is comprehensive.	The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.