An Evaluation of the current State of web server development in Rust and Java

Marc Matija

August 25, 2024

Abstract

This paper aims to evaluate the current state of web server development in Rust and Java. In doing so, it will cover the basics of web server development in both languages, the tools and libraries available, and the advantages and disadvantages of using each language for web server development. Furthermore it will include a comparison of the two languages in terms of performance, ease of use, and other factors that are important for web server development.

Contents

1 Introduction

Web server development is an important aspect of modern software development as they are used to host websites, web applications, and other online services. One of the most popular languages for this is Java and the Spring framework. Java is a mature and widely used programming language that is known for its performance, scalability, and reliability. However, in recent years, Rust has emerged as a promising alternative to Java for web server development as seen at Discord [2], Cloudflare[1], and other companies. Rust is in stark contrast to Java, is a systems programming language that is designed for performance, safety, and concurrency. This paper aims to evaluate the current state of web server development in Rust and Java, and to provide an overview of the tools, libraries, and best practices for web server development in both languages.

1.1 An Overview of Spring Boot

Spring Boot is a popular Java framework for building web applications and microservices used by companies like Deutsche Bahn[3], Udemy [5] and many others[4]. Built on top of the Spring framework, Spring Boot provides a set of tools and libraries that make it easy to build web applications and microservices in Java. It finds wide adoption in the industry due to its ease of use, maintanability and scalability, aswell as the vast amount of libraries and tools available for it.

1.2 An Overview of Actix Web

Actix is a Rust framework for building backend services in rust and the one currently one of the most actively developed framworks, within the Rust ecosystem. It provides provides a very similar feature set to Spring Boot making it a suitable candidate for comparison.

1.3 Other Used Technologies

In addition to the frameworks mentioned above, I will use the following technologies to build the web server:

- RustRover An IDE by Jetbrains that provides support for Rust development.
- IntelliJ IDEA Ultimate An IDE by Jetbrains that provides support for Java development.
- VS Code A Text editor by microsoft used in this project for writing the LaTeX document.
- PostgreSQL A popular open-source relational database management system that is widely used in the industry.
- Docker A platform for building, shipping, and running applications in containers.
- Compose A tool for defining and running multi-container Docker applications.
- JWT A standard for creating JSON-based access tokens that are used for authentication and authorization.
- Swagger A tool for designing, building, and documenting APIs.
- JUnit A popular testing framework for Java.
- Postman A tool for testing APIs.
- K6 A tool for load testing web applications.
- DBeaver A database management tool.
- Talend API Tester A tool for testing APIs.

2 Specifications for the web server

In order to evaluate the current state of web server development in Rust and Java, I will build a simple web server in both languages using the previously mentioned frameworks. The web server will feature a REST api that allows users to create channels, add messages to channels, and retrieve messages from said channels. In addition, the web server will feature a websocket endpoint that allows users to subscribe to channels and receive messages in real-time. Authentication will be handled using JWT tokens and a profile endpoint to authenticate users and retrieve their information.

2.1 REST API

The REST API will feature the following endpoints:
Channel endpoints:

- **Create Channel** POST /api/v1/channels
  Creates a new channel with the given name.
- **Get Channel** GET /api/v1/channels/{id}
  Retrieves the channel with the given id.
- **Get All Channels** GET /api/v1/channels
  Retrieves all channels.
- **Change Channel** PUT /api/v1/channels/{id}
  Changes the name of the channel with the given id.
- **Delete Channel** DELETE /api/v1/channels/{id}
  Deletes the channel with the given id.

Message endpoints:

- **Add Message** `POST /api/v1/channels/{id}/messages`
  Adds a new message to the channel with the given id.
- **Get Message** `GET /api/v1/channels/{id}/messages/{id}`
  Retrieves the message with the given id from the channel with the given id.
- **Get All Messages** `GET /api/v1/channels/{id}/messages`
  Retrieves all messages from the channel with the given id.

- **Change Message** `PUT /api/v1/channels/{id}/messages/{id}`
  Changes the content of the message with the given id from the channel with the given id.
- **Delete Message** `DELETE /api/v1/channels/{id}/messages/{id}`
  Deletes the message with the given id from the channel with the given id.

2.2 Database Schema

The Implementations in both languages will feature the same PostgreSQL database, thus the database schema will be the same for both implementations.

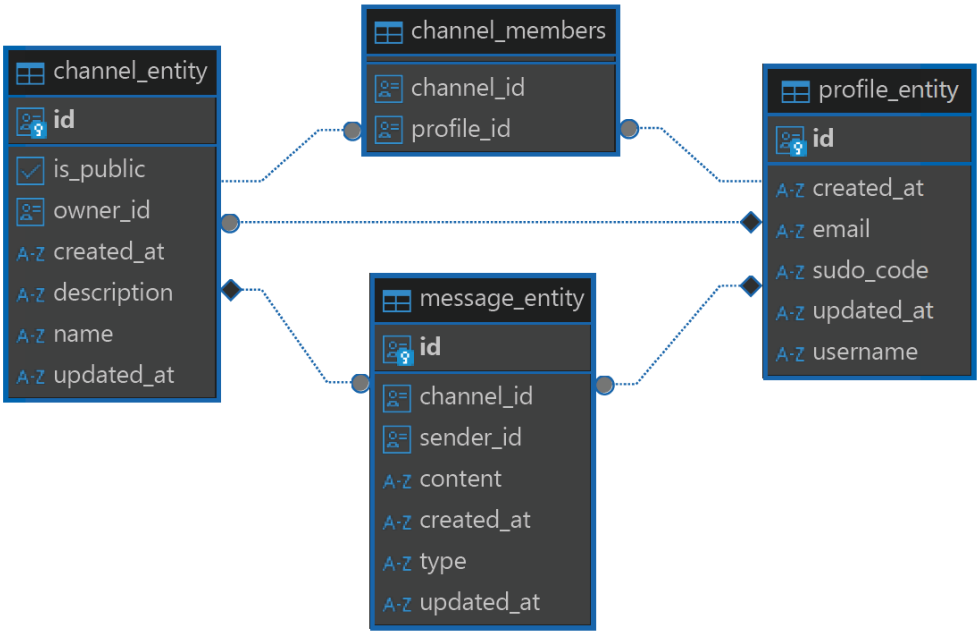


Figure 1: Database schema in DBeaver

It features a table for users, a table for channels, and a table for messages. The users table stores the username, email while the channels table stores the id of whoever created it, making him the **owner** of the channel. The messages table stores the content of the message, the id of the user who sent it, and the id of the channel it was sent to. All Tables also include coloumns for the creation and last update time of the row.

A note about creation

A worthy note about the creation of this database schema is that it was not created by hand. For both Rust and Java I took the help of ORMs (Object Relational Mapper) which automatically map structures from each language into SQL statements. Spring JPA, the ORM used in the Spring-Boot implementation, has the ability to automatically create relations between entities by using anotations.

```
@Entity
@Table(name = "channel_entity")
public class ChannelEntity {
   /* ... */
   @ManyToMany
   @JoinTable(
         name = "channel_members",
         joinColumns = @JoinColumn(name = "channel_id"),
         inverseJoinColumns = @JoinColumn(name = "profile_id")
   )
   private List<ProfileEntity> members;
   /* ... */
}
```

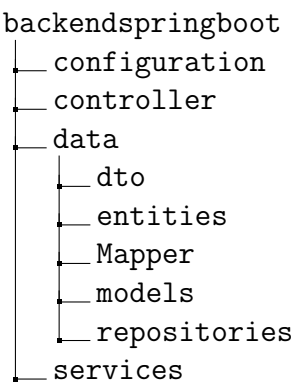
The **@ManyToMany** annotation denotes a traditional Many to Many relationship in a relational database, allowing me to let JPA do the heavy lifting of creating all necesarry foreign keys and required Tables using the **@JoinTable** annotation. Further information about the workings of JPA can be found in the Java Implementation section.

2.3 Authentication

Authentication will be handled using JWT tokens. When a client signs up or signs in, they will receive a JWT token that they can use to authenticate themselves when making requests to the web server. The JWT token will be sent in the Authorization header of the HTTP request as a Bearer token. Both Server implementations will have to verify the token before sending new messages to the client.

3 Implementation in Java

3.1 Project Structure



Pictured here is the Project structure for the Spring Boot implementation of the API. It is subdevided into the **configuration** folder which contains all descriptions for Spring Security, the **controller** folder which contains all rest controllers, which manage the incoming HTTP requests, A **data** folder with subfolders for all relevant data structures found within the API and **services** which perform all actions on the data using the repositories and basicaly glue together the controllers and the data.

3.2 Spring Boot

Rest APIs written in Spring boot use Java Classes with the **@RestController** annotation to handle incoming HTTP requests and route them to specific methods within the annotated controller class.

```
@RestController
@RequestMapping(CHANNEL_BASE_URL)
public class ChannelController {
   private final ChannelService channelService;
   @Autowired
   public ChannelController(ChannelService channelService) {
      this.channelService = channelService;
   }
   @GetMapping("{id}")
   public ResponseEntity<Channel> getChannel(@PathVariable UUID id){
      return ResponseEntity.ok()
         .body(channelService.getChannel(id));
   }
   @PostMapping("")
    public ResponseEntity<Channel> createChannel() {
        var channel = channelService.createChannel(new UUID(0, 0));
        URI location = URI.create(CHANNEL_BASE_URL +
            channel.getId().toString());
        return ResponseEntity.created(location).body(channel);
    }
}
```

Figure 2: A shortened Example of a Controller in Springboot

In this case the **CHANNEL_BASE_URL** is set to **/api/v1/channels/** meaning, all requests starting with this specific string will be routed to this class and it's methods corresponding to how they are annotated. F.e. anything annotated **@GetMapping** will capture a GET request, while **@PostMapping** does the same for any POST requests that arrive at the specified path. These classes don't need to be specifically defined or initiallized by the programmer, because of Java's ability to use Reflection. Spring can just look for the annotation and will automatically register it as a valid Controller.

3.3 Spring Data JPA

As the ORM of my choice, I have decided to use Spring JPA in this project as it gives me the lowest amount of code I have to write to get SQL statements to work and as explained in the Database paragraph it allows me to generate a schema from classes. A Feature, which other ORMs can lack. Data in Spring Data JPA is devided into **Entities** and **Repositories**. Repositories describe a collection of a specific type of Entity and can in their most basic form be written as a one liner. Repositories in Spring JPA allow writing SQL

```
public interface MessageEntityRepository
   extends JpaRepository<MessageEntity, UUID> {
   Collection<MessageEntity> findAllByChannelId(UUID channelId);
}
```

Figure 3: The Message Repository for the Message Entities

Statements purely but naming the method as shown in the example, where *findAllByChannelId* gets broken down into
SELECT FROM message_entity WHERE id c̄hannelId

Entities

Entities in Springboot describe the objects in the Database and get mapped to a table. where the coloumns are described by the fields and properties of each object. Annotations help JPA to further define features and structure of each table. In my case, the Entities required in the Database are Channels, Messages and Profiles.

As is visible however, is that these entities are not directly relayed to the controllers by the services, that is because entites can be database specific and incorporate functionality that might break, when changing databases. That's why for each Object in the database Model and DTO (Data Transfer object) are created to decouple the controller logic from the underlying entities, which allows changing the Database without requiring changes to the controller logic.

Services

Services glue together the controller and their entities. Each controller is internally build as a Singleton class for for more efficient memory management. Each service allows for specific actions on entities and breaks those

down into their respective operations on the repositories. As an example here

```
@Service
public class ChannelService {
   /* ... */
   public Channel updateChannel(Channel channel) {
      var channelEntity = channelEntityRepository
         .findById(channel.getId())
         .orElse(null);
      if (channelEntity == null)
         throw new IllegalArgumentException("Channel not found");

      channelEntity.setDescription(channel.getDescription());
      channelEntity.setIsPublic(channel.getIsPublic());
      channelEntity.setName(channel.getName());
      channelEntity.setUpdatedAt(LocalDateTime.now().toString());

      var saved = channelEntityRepository.save(channelEntity);
      return ChannelMapper.ChannelEntityToChannel(saved);
   }
   /* ... */
}
```

Figure 4: Mapping the update action from one method to multiple repository methods

the Channel object gets new data by first fetching the current entity and it's values from the database and changing what is needed and allowed to change by the API. Afterwards the channel entity gets saved to the database and returned as it's model.

Mapper

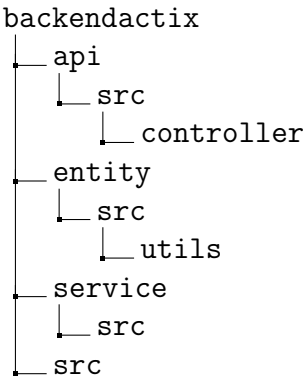
Mapper are very simple, they take in either an Entity or a model and map them to the corresponding other class. They are what keeps everything decoupled and implementation independent.

3.4 Dependencies

A list of all dependencies can be found here.

4 Implementation in Rust

4.1 Project Structure



Similar to Java, the project is broken down into differnt packages, or crates as they are called in Rust. In rust, this is done in order to manage dependencies better. F.E. the Services to not require Actix-Web, which means it does not need to rebuild all of Actix-Web whenever there are changes made to the services. Speeding up build times by a little bit. But just like in the spring boot implementation, it is once again devided up into the controllers, found in the api crate, the data, found in the entity crate, which was partly generated by SeaOrm, more on that later and the services glueing everything together.

4.2 Actix

Just like Spring, Actix features a System of routes by annotations (macros), however, due to the nature of Rust not featuring reflections, each route requires to at some point be manually registered to the server as a service, or by declaring it as a request endpoint and setting it's route and HTTP Request-type manually. In order to achieve this, Actix provides a configuration struct to register said services.

```
// found in api/src/lib.rs
fn init(cfg: &mut web::ServiceConfig) {
    controller::channel_controller::init(cfg);
    controller::message_controller::init(cfg);
}

// found in api/src/controller/channel_controller.rs
pub fn init (cfg: &mut web::ServiceConfig){
    cfg.service(create);
    cfg.service(get_all);
    cfg.service(get);
    cfg.service(put);
    cfg.service(delete);
}
```

Provided here is an example taken from the channel controller which gets all current channels in the database and returns them as a JSON object. The Service is talked to using the implementation methods on the ChannelService struct.

```
#[get("/api/v1/channels/")]
pub async fn get_all(state: web::Data<ApiState>) ->
    Result<HttpResponse, Error> {
    let conn = &state.conn;
    let result = ChannelService::get_all_channels(conn).await;

    if result.is_err() {
        return handle_error_internal(result.unwrap_err());
    }

    let json_result = serde_json::to_string(&result.unwrap());
    Ok(HttpResponse::Ok()
        .content_type(ContentType::json())
        .body(json_result.expect("Failed to serialize")))
}
```

4.3 SeaOrm

The main Framework for modifiyng data in the Database is, atleast for this project, SeaOrm, which is the closest framework I could find to replicate the behaviour of Spring JPA. It generates builds queries using the Sea-Query crate and allows for declaring SQL statements inside of Rust, rather than having to write them by hand. One thing it does not do, is build the Database from the provided Entities, but rather build the entitites from an existing Database, which allowed me to use both the strength of spring JPA to create the Database, then SeaOrm to create the entities for the Rust implementation.

```
pub async fn get_all_channels(db: &DbConn) ->
    Result<Vec<channel_entity::Model>, DbErr>{
    channel_entity::Entity::find().all(db).await
}

pub async fn get_channel(db: &DbConn, id: Uuid) ->
    Result<Option<channel_entity::Model>, DbErr>{
    channel_entity::Entity::find_by_id(id).one(db).await
}
```

Entites

As mentioned before SeaORM allows the user to generate entities form an existing Database and also map their relations to one another directly into Rust structs and Implementations. One features of these generated structs is the provided structure to add code when a new entity is created and when an entity is deleted, this allows easy modification and preset values for f.e. the creation and update date.

```
impl ActiveModelBehavior for ActiveModel {
    fn new() -> Self {
        let current_date: DateTime<Utc> = SystemTime::now().into();
        Self {
            is_public : Set(false),
            id: Set(Uuid::new_v4()),
            owner_id : Set(
            Option::Some(
               uuid!("00000000-0000-0000-0000-000000000000")
            )),
            created_at : Set(current_date.to_rfc3339()),
            description : Set("".to_owned()),
            name : Set("".to_owned()),
            updated_at : Set(current_date.to_rfc3339()),
            ..ActiveModelTrait::default()
        }
    }
}
```

4.4 Dependencies

A list of all dependencies can be found here.

5 Testing Methodology

As the main focus of this paper is to test performance, I will perform load testing to see, how much each implementation differs in performance. In doing so, i used help from the tool k6 developed by Grafana, which allows me to write my load tests in JavaScript and automatically get the results in a specified format. For this project, all results are written to a JSON file for later analysis.

5.1 K6

As explained before for the main testing tool used for both implementations is k6, providing a platform independent system to test apis with. K6 works by leveraging the node.js runtime to create virtual users to send requests at the api specified by the user, or by using .env files.

5.2 Performance Testing

5.3 Load Testing

To facilitate efficient load testing, the k6 test suite is ran multiple times

5.4 Other Tests

Build times

Memory Usage

CPU Usage

6 Comparison

6.1 Developer Experience

6.2 Performance

6.3 Load Endurance

7 Conclusion

8 Sources

- Rust Edu
- The Rust Programming Language
- Rust Standard Library
- Cargo Guide
- Rust by Example
- Rustdoc
- Actix
- DBeaver
- SeaOrm Actix Example

Whole bibliography

[1] Cloudflare. *Open sourcing Pingora: our Rust framework for building programmable network services.* URL: `https://blog.cloudflare.com/pingora-open-source` (visited on 08/14/2024).

[2] Discord. *Why Discord is switching from Go to Rust.* URL: `https://discord.com/blog/why-discord-is-switching-from-go-to-rust` (visited on 08/14/2024).

[3] DB Jobs. *Einstiegsmöglichkeiten für IT Professionals!* URL: `https://db.jobs/de-de/dein-einstieg/akademische-professionals/it` (visited on 08/14/2024).

[4] Stackshare. *Spring Boot.* URL: `https://stackshare.io/spring-boot` (visited on 08/14/2024).

[5] Udemy. *Tge Udemy Techstack.* URL: `https://stackshare.io/udemy/udemy` (visited on 08/14/2024).