

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

Marc Matija

August 23, 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	3
1.1	An Overview of Spring Boot	3
1.2	An Overview of Actix Web	3
1.3	Other Used Technologies	4
2	Specifications for the web server	5
2.1	REST API	5
2.2	Websocket API	7
2.3	Database Schema	8
2.4	Authentication	9
3	Implementation in Java	10
3.1	Project Structure	10
3.2	Spring Boot	11
3.3	Spring Data JPA	12
4	Implementation in Rust	14
4.1	Project Structure	14
4.2	Actix	14
4.3	Sea-ORM	14
5	Testing Methodology	14
5.1	K6	14
5.2	Load Testing	14
5.3	Performance Testing	14
6	Comparison	14
6.1	Developer Experience	14
6.2	Performance	14
7	Conclusion	14
8	Sources	15

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, maintainability and scalability, as well 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 frameworks, 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.
- 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.

Authentication endpoints:

- **Sign up** POST /api/v1/auth/signup
Registers a new user with the given username and password.
- **Sign in** POST /api/v1/auth/signin
Logs in the user with the given username and password.
- **Sign out** POST /api/v1/auth/signout
Logs out the currently logged in user.

Profile endpoints:

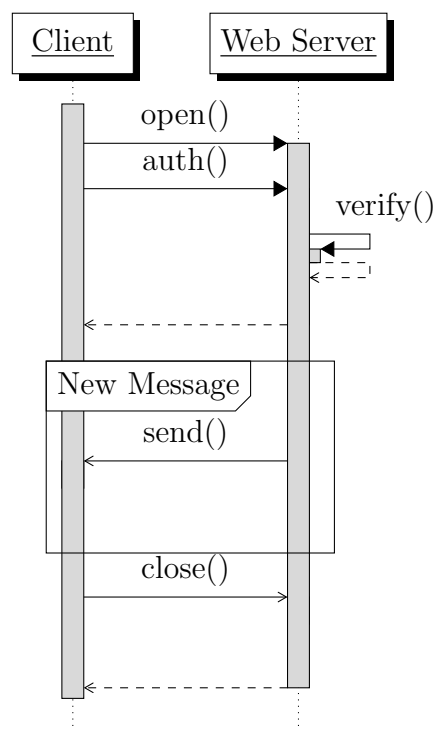
- **Get Profile** GET /api/v1/profile
Retrieves the profile of the currently logged in user.
- **Change Profile** PUT /api/v1/profile
Changes the profile of the currently logged in user.
- **Delete Profile** DELETE /api/v1/profile
Deletes the profile of the currently logged in user.

2.2 Websocket API

In order to allow users to receive messages in real-time, the web server will feature a websocket endpoint that allows users to subscribe to channels and receive updates when new messages are added to the channel. Users will automatically be subscribed to Channels they are a part of. When establishing a connection to a websocket endpoint, they will need to provide a valid JWT token in order to authenticate themselves.

- **Subscribe** `ws://localhost:8080/api/v1/ws`

Subscribes the user to the websocket endpoint and allows them to receive updates when new messages are added to the channels they are a member of.



In a websocket connection request, the headers of the HTTP request cannot be modified, the server will have to first accept the connection, then wait for the client to send a JWT token. The server will then verify the token and if it is valid, the client will be subscribed able to receive all messages from the channels they are a part of. Received messages will take the same format as the messages in the REST API.

Figure 1: Message flow in a websocket connection

2.3 Database Schema

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

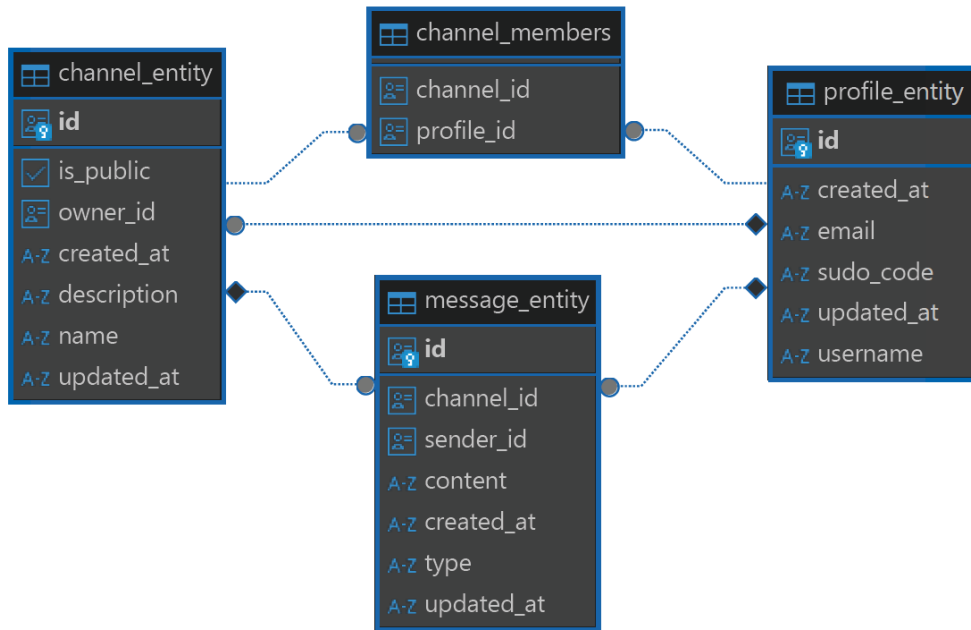


Figure 2: Database schema in DBeaver

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 columns 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 annotations.

```
@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 necessary foreign keys and required Tables using the **@JoinTable** annotation. Further information about the workings of JPA can be found in the Java Implementation section.

2.4 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

```
backendspringboot
├── configuration
├── controller
├── data
│   ├── dto
│   ├── entities
│   ├── Mapper
│   ├── models
│   ├── repositories
└── services
```

Pictured here is the Project structure for the Spring Boot implementation of the API. It is subdivided 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 basically 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 3: 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 initialized 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 divided 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 4: 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 = channelId

Entities

Entities in Springboot describe the objects in the Database and get mapped to a table. where the columns 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 5: 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.

4 Implementation in Rust

4.1 Project Structure

4.2 Actix

4.3 Sea-ORM

5 Testing Methodology

5.1 K6

5.2 Load Testing

5.3 Performance Testing

6 Comparison

6.1 Developer Experience

6.2 Performance

7 Conclusion

8 Sources

- Rust Edu
- The Rust Programming Language
- Rust Standard Library
- Cargo Guide
- Rust by Example
- Rustdoc
- Actix
- DBeaver
- Sea-ORM 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).