Autić Parkić SDD

Document Name: Autić Parkić Application Design Document

Version: 1.1

Author: Bogdan Jovanović

Date: June 22, 2023

Table of Contents

1. Introduction

- **Purpose of the Document**: This document provides an in-depth design overview of the Autić Parkić Application.
- **Scope**: This software will manage vehicles and rides, allowing administrators to manage vehicle data and users to create and manage rides.
- **Links**: Homepage, Dashboard, Swagger, GitHub Repo.

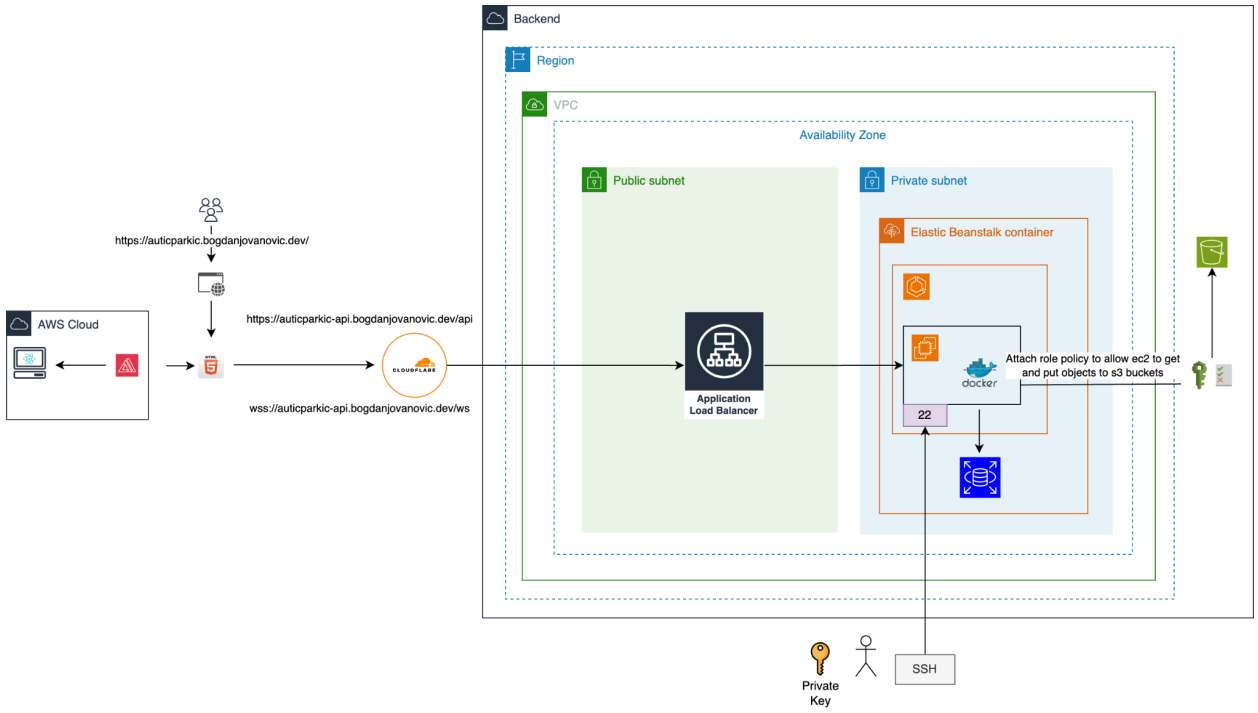
2. System Architecture

- **Real-time Communication**: The application must ensure real-time sync with multiple client instances, displaying consistent elapsed time for rides in *RUNNING* status across all clients. This demands a real-time communication mechanism between the server and all connected clients to maintain a consistent state.

- **Application Workflow**: Users can register a vehicle, which has a distinct name in the database and an associated image stored on *AWS S3*. Once a vehicle is registered, rides can be initiated. It's important to note that a vehicle can only be engaged in a single ride at a given time. Rides transition through several distinct statuses: *CREATED*, *RUNNING*, *PAUSED*, *STOPPED*, and *FINISHED*. Initially, a ride is in the *CREATED* state. Once *RUNNING*, it can either be *PAUSED* or *STOPPED*. When a ride is *PAUSED*, it can be resumed by returning to the *RUNNING* state. Conversely, a *STOPPED* ride offers two choices: it can either restart or advance to its *FINISHED* state. Opting to restart concludes the current ride and launches a new one using the same vehicle. Once designated as *FINISHED*, a ride's status is fixed and unalterable. A key function is the real-time tabulation and distribution of the elapsed ride time once it's *RUNNING*. This time calculation disregards any *PAUSED* intervals. To facilitate this, the database stores arrays of started_at and paused_at timestamps. The computed elapsed time is then shared in real-time to all connected clients using the *STOMP WebSocket* protocol. Considering the application's need to relay elapsed time updates every second, ongoing rides are temporarily cached in system memory (*RAM*) using `Collections.synchronizedList`. This strategy helps avoid recurrent database queries. Given the cap of 20 active rides at any instance, there's no compelling reason to employ comprehensive distributed caching systems like Redis or memcached.
- ## Considered Alternatives to REST
    - **HTTP Polling (Long Polling)**: Clients would regularly send HTTP requests to the server to check for updates. However, this approach has several downsides, including increased server load, network latency, and the possibility of clients receiving updates at slightly different times.
    - **WebSockets**: A WebSocket provides a full-duplex communication channel over a single, long-lived connection. This allows the server to push updates to clients in real-time, ensuring all clients receive updates simultaneously.
    - **Server-Sent Events (SSE)**: SSE is a one-way communication channel from the server to the client over a single HTTP connection. It allows the server to push updates to the client in real-time, but does not allow the client to send data to the server over the same connection.
- After evaluating the alternatives, it was decided to use WebSockets (to be precise publish-subscribe messaging through STOMP as a sub-protocol over WebSocket) for the following reasons:
    - **Full-Duplex Communication**: WebSockets provide a full-duplex communication channel, allowing both the server and the clients to send data over the same connection. This is particularly useful for our application as it requires bi-directional communication.
    - **Real-time updates**: WebSockets allow the server to push updates to clients in real-time, ensuring all clients receive updates simultaneously and maintain a consistent state.
    - **Efficiency**: WebSockets are more efficient than HTTP polling as they require fewer network resources and reduce the server load.

    - **Standardized Protocol**: WebSocket is a standardized protocol supported by all modern web browsers and many programming languages and frameworks.
    - **Firewall friendly (standard)**: Designed to work over HTTP using ports 80 and 443 and allowing re-use of existing firewall rules.
- **Cons**:
    - L7 L/B challenging (timeouts)
    - Proxying is tricky
    - Stateful protocol, hard to scale horizontal

- **High-Level Architecture Diagram**:



3. Tools and Technologies

➔ **Backend Technologies**:

- Quick overview of languages considered for backend implementation:
- **Golang**: Compiled language, that offers fast performance and efficient concurrent programming through goroutines. Its rich standard library excels in I/O operations and web servers, and its syntax emphasizes readability and simplicity. Additionally, Go's compiled binaries work across platforms. On the downside, its ecosystem is not as mature as Java's. Its garbage collection can introduce latency, and the initial lack of generics, while being addressed, has been a limitation for some applications.
- **Rust**: Boasts memory safety through its unique ownership system, negating the need for a garbage collector. As a low-level language, its performance is on par with C and C++. It also features modern, safe concurrency tools and a burgeoning ecosystem. However, grasping its ownership system can be challenging, and its ecosystem, while growing, isn't as mature as those of some established languages like Java.
- **Java**: Established language, boasts a vast ecosystem, strong performance with JIT compilers, and JVM optimization. Its "Write Once, Run Anywhere" approach and rich concurrency tools, alongside extensive enterprise features, make it ideal for large-scale apps. The potential of Java 21's virtual threads could further enhance performance. Yet, Java often uses more memory than Go or Rust and

can be verbose, resulting in longer codebases. Web Frameworks: Spring Boot, Quarkus, Micronaut.

- **TypeScript**: Extension of JavaScript, introduces static typing, enabling early error detection. It blends modern ES features with static typing and has a burgeoning backend ecosystem due to Node.js. However, its reliance on Node.js may cause performance drops in CPU-intensive tasks compared to compiled languages like Go or Rust. Additionally, unlike pure JavaScript, TypeScript requires a compilation step.
- **Final Decision**: Considering the above evaluations, the choice was made to use **Java with the Spring Boot framework** for backend development. The primary reasons are Java's mature ecosystem, robust performance, extensive library support, and advanced features that Spring Boot brings to the table, optimizing both development speed and production readiness.

➔ **Frontend Technologies**:

- When it comes to choosing a frontend technology the choice is very hard since there are tons of JavaScript frameworks and they are shipped daily. Since the application requires some kind of state management between different UIs here are the considered frontend frameworks:
- **Vue.js**: Favored for its easy integration into projects, comprehensive documentation, and reactive two-way data binding. It provides a gentler learning curve for those familiar with basic web technologies and promotes reusable, modular components. However, compared to React, Vue has a smaller community, can sometimes be overly flexible leading to confusion, and has less market penetration among major companies, potentially affecting its perceived suitability for large-scale applications.
- **React.js**: Backed by Facebook, boasts a large community, leading to an extensive ecosystem and high performance due to its virtual DOM. It emphasizes modular, reusable components and offers advanced features for large applications. However, it presents a steeper learning curve for beginners, undergoes rapid updates, and occasionally requires more boilerplate code than Vue.
- For the frontend development of a complex UI, React.js with TypeScript was chosen due to React's robust state management, huge ecosystem, and modular structure.

➔ **Database Technologies**:

- **PostgreSQL**: Robust relational database, is favored for its advanced locking mechanisms, which facilitate intricate concurrency tasks and data integrity. Its ACID compliance ensures reliable transactions, while its extensibility supports diverse data types. However, like many RDBMS systems, it may exhibit scaling challenges when facing extremely high workloads compared to some NoSQL alternatives. The decision to opt for PostgreSQL was also bolstered by its vast community and strong support for complex queries.

➔ **Continuous Integration / Continuous Deployment (CI/CD)**:

- GitHub Actions was selected for CI/CD due to its seamless integration with GitHub repositories, simplifying the development pipeline. Its matrix builds enable testing across multiple platforms and configurations concurrently, enhancing efficiency. While it provides free minutes for public repositories, extensive builds might incur costs. The choice was further influenced by its rich marketplace of actions, enabling customizable workflows, and straightforward YAML configuration.

➔ **Hosting**:

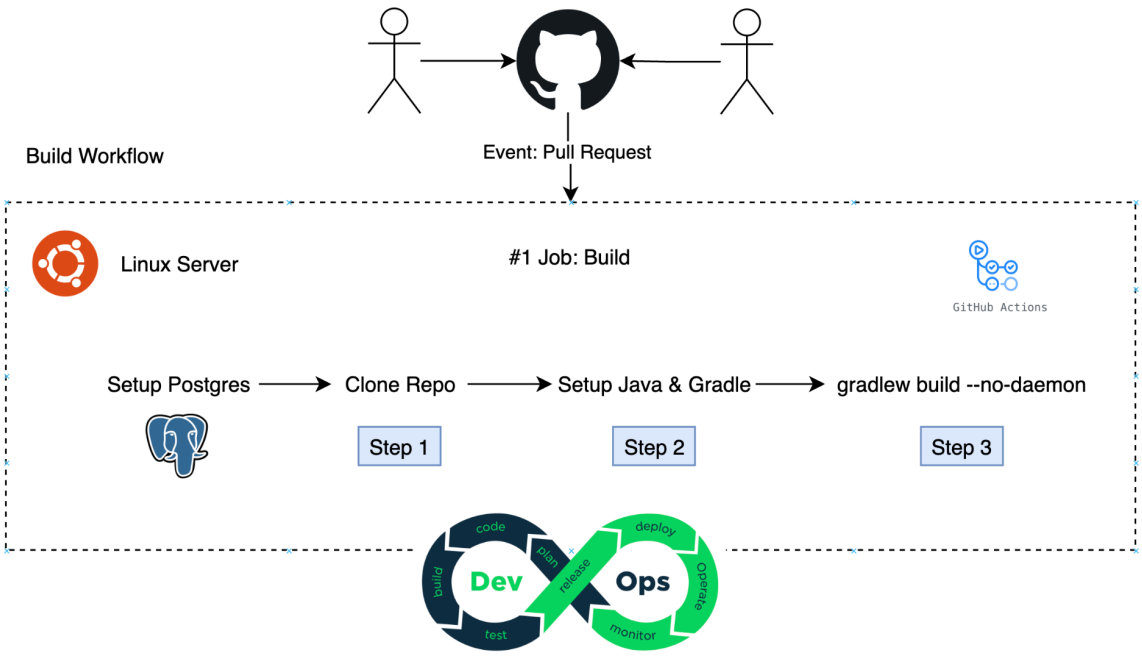
- **Backend**: Given the application's reliance on WebSockets, AWS Elastic Beanstalk was the preferred choice due to its streamlined configuration capabilities. Ensuring WebSocket protocol support can be complex, but with AWS Beanstalk combined with the Application Load Balancer, it becomes straightforward, enabling real-time bidirectional communication without intricate setup. This decision ensured seamless deployment and scalability while accommodating the application's specific requirements. Within the application, AWS S3 buckets have been integrated to handle file uploads. Leveraging S3 ensures scalable, secure, and efficient storage and retrieval of files, enhancing the overall functionality and reliability of the application.
- **Frontend**: AWS Amplify was chosen for deploying the React application due to its seamless integration with frontend workflows, enhanced global performance via Amazon CloudFront's CDN, and automated CI/CD capabilities ensuring rapid and secure deployments.

➔ **Domain Management & DNS Configuration**:

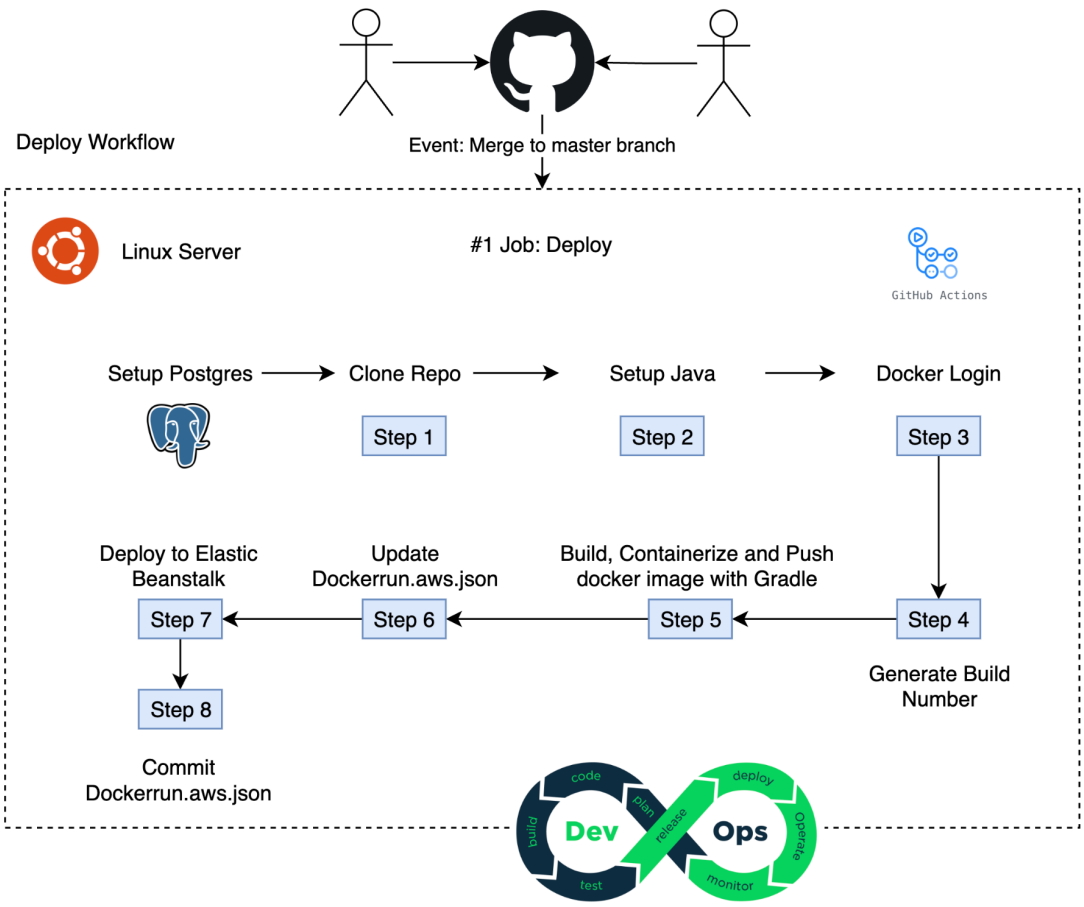
- For domain name registration, Cloudflare is utilized, pointing directly to AWS Application Load Balancer. This choice was influenced by Cloudflare's advanced DNS features, ensuring reliable domain resolution and enhanced performance. AWS ALB cannot be accessed directly, meaning that it will only accept connections from Cloudflare IPs. The request is proxied through Cloudflare servers before it reaches the ALB.
- **Secure Connections (HSTS)**: For ensuring secure data transfer, the application employs SSL/TLS encryption. The certificate authority (CA) provided by Amazon will be utilized to validate and establish these encrypted connections. With the latest browsers HTTP requests will be automatically redirected to HTTPS due to built-in HSTS support, however, if for example another client is used such as `curl` to send a request, then HSTS will be ignored and HTTP requests won't be redirected to HTTPS.

4. CI/CD Deployment Diagrams

- **Continuous Integration**: Once the pull request is opened automatically *`backend-ci.yml`* GitHub Action will be triggered to build and test the application.

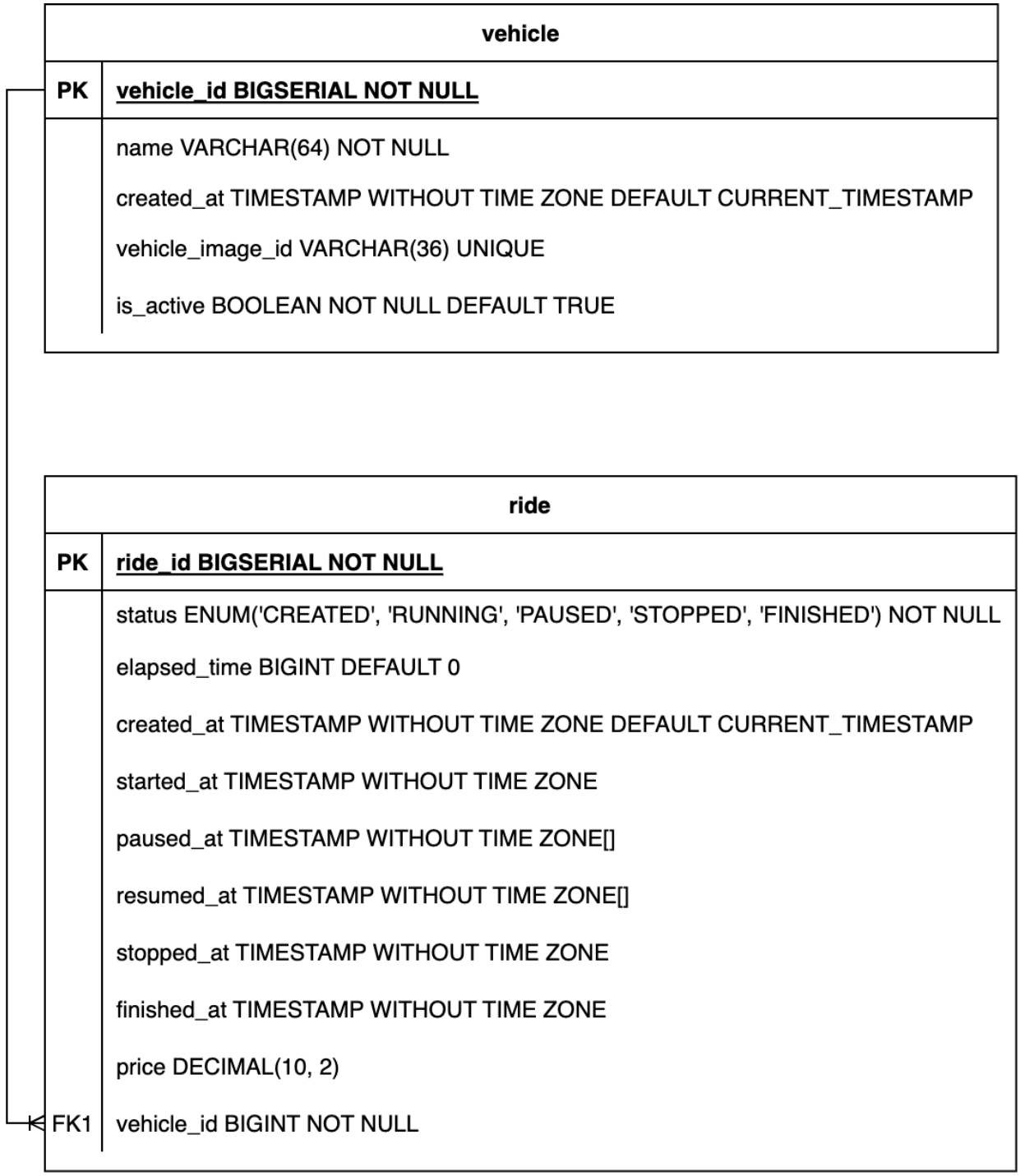


- **Continuous Delivery**: Once the pull request is successfully built, tested, and approved, on merge to master branch backend-cd.yml GitHub Action will be triggered to build, containerize, and push the docker image to Docker Hub. The new image name will be written in Dockerrun.aws.json (but not committed) and deployed to AWS Elastic Beanstalk. Once the deployment is finished AWS will commit to the repository docker image with the latest version/tag.



5. Detailed Design

- **Database Design**:



- **Concurrency Control**:
    - To maintain data integrity and handle concurrent requests, the application will leverage explicit locking mechanisms provided by PostgreSQL. Specifically, tables will be locked at a row level to prevent race conditions and ensure consistent reads/writes. This ensures that if multiple clients try to modify the

same data simultaneously, the system will handle the requests in a serialized manner, avoiding potential conflicts and data inconsistencies.

- **UI Design**: (To be detailed with wireframes/screenshots)
  - **Vehicle Control Dashboard**: Display for adding/modifying vehicles.
  - **Ride Control Interface**: Display for starting, pausing, stopping, and ending rides.
- **API Design**:
  - **Ride API**:
    - **Create Ride**:
      - **Endpoint**: /rides.create
      - **Method**: MessageMapping
      - **Payload**: VehicleIdPayload containing the vehicle ID
      - **Response**: List of RideDTO
      - **Description**: Creates a ride with a specific vehicle id.
    - **Start Ride**:
      - **Endpoint**: /rides.start
      - **Method**: MessageMapping
      - **Payload**: RideIdPayload containing the ride ID
      - **Description**: Starts a ride with a specific id.
    - **Pause Ride**:
      - **Endpoint**: /rides.pause
      - **Method**: MessageMapping
      - **Payload**: RideIdPayload containing the ride ID
      - **Response**: List of RideDTO
      - **Description**: Pauses a ride with a specific id.
    - **Stop Ride**:
      - **Endpoint**: /rides.stop
      - **Method**: MessageMapping
      - **Payload**: RideIdPayload containing the ride ID
      - **Response**: List of RideDTO
      - **Description**: Stops a ride with a specific id.
    - **Restart Ride**:
      - **Endpoint**: /rides.restart
      - **Method**: MessageMapping
      - **Payload**: RideIdPayload containing the ride ID
      - **Response**: List of RideDTO
      - **Description**: Restarts a ride with a specific id.
    - **Finish Ride**:
      - **Endpoint**: /rides.finish
      - **Method**: MessageMapping
      - **Payload**: RideIdPayload containing the ride ID
      - **Response**: List of RideDTO
      - **Description**: Marks a ride as finished with a specific id.
    - **Stream Unfinished Rides**:
      - **Endpoint**: /rides.streamUnfinishedRidesData

        - **Method**: MessageMapping
        - **Response**: List of RideDTO
        - **Description**: Streams data for unfinished rides.
- **Vehicle API**:
    - **Get Vehicle By ID**:
        - **Endpoint**: GET /{vehicleId}
        - **Response**: Vehicle
        - **Description**: Retrieves a specific vehicle by its ID.
    - **Create Vehicle**:
        - **Endpoint**: POST /
        - **Request Body**: VehicleRequestDTO
        - **Response**: ID of the created vehicle.
        - **Description**: Creates a vehicle if it doesn't already exist.
    - **Update Vehicle**:
        - **Endpoint**: PUT /{vehicleId}
        - **Request Body**: VehicleRequestDTO
        - **Description**: Updates a specific vehicle by its ID.
    - **Get Available Vehicles**:
        - **Endpoint**: GET /available
        - **Response**: List of Vehicle
        - **Description**: Retrieves all available vehicles.
    - **Get All Vehicles**:
        - **Endpoint**: GET /
        - **Response**: List of Vehicle
        - **Description**: Retrieves all vehicles.
    - **Upload Vehicle Image**:
        - **Endpoint**: POST /image/{vehicleId}/upload
        - **Request Body**: Multipart image file.
        - **Description**: Uploads an image for a specific vehicle by its ID.
    - **Get Vehicle Image**:
        - **Endpoint**: GET /image/{vehicleId}/download
        - **Response**: Byte array of the vehicle image.
        - **Description**: Downloads the image of a specific vehicle by its ID.
- **Error Handling**: For error management, Spring Boot's `@ControllerAdvice` is employed to ensure appropriate exception handling and response dispatching. This aids in delivering precise responses like bad requests and other relevant error messages to the user.

6. Revision History

- **Version 1.0**: Initial version detailing the design of the Autić Parkić Application.

- **Version 1.1**: Updated to include real-time communication requirements for consistent ride status across multiple client instances.