

PROJECT REPORT

Liuyin Shi, Samkith K Jain, Varun Rajput

Department of Computing & Software, McMaster University

I. Introduction

The architectural design of our project, a gamified running application aimed at promoting health and fitness, serves as the backbone for a dynamic and user-centric experience. This report offers an overview of our architectural choices, detailing the justification for our bounded contexts, service boundaries, hexagonal architecture, APIs, testing, potential service patterns and improvements.

In Section II, we present the result of our event storming in the form of bounded contexts. We justify the choice of our bounded contexts, elaborating on how each context encapsulates distinct functionalities. We discuss the significance of these boundaries in terms of system modularity and maintainability. In Section III, we use hexagonal architecture to provide an overview of the relationships between different services and components. In Section IV, we provide detailed explanations for the key APIs in our system, classifying them as asynchronous or synchronous based on their role and significance. We discuss the reasoning behind these classifications, emphasizing the importance of responsive and non-blocking communication. Section V explores the potential use of service patterns in our architecture, including circuit breakers and load balancers. We justify their implementation, demonstrating how they enhance system reliability, fault tolerance, and scalability. In Section VI, we delve into various miscellaneous implementation details, emphasizing the aspects we targeted to enhance the maintainability of our application's infrastructure. Then, in Section VII, we wrap up by summarizing the architecture and highlighting potential areas for improvement that were beyond the scope of this project due to time constraints.

II. Bounded Context

The result of our event storming is given below:

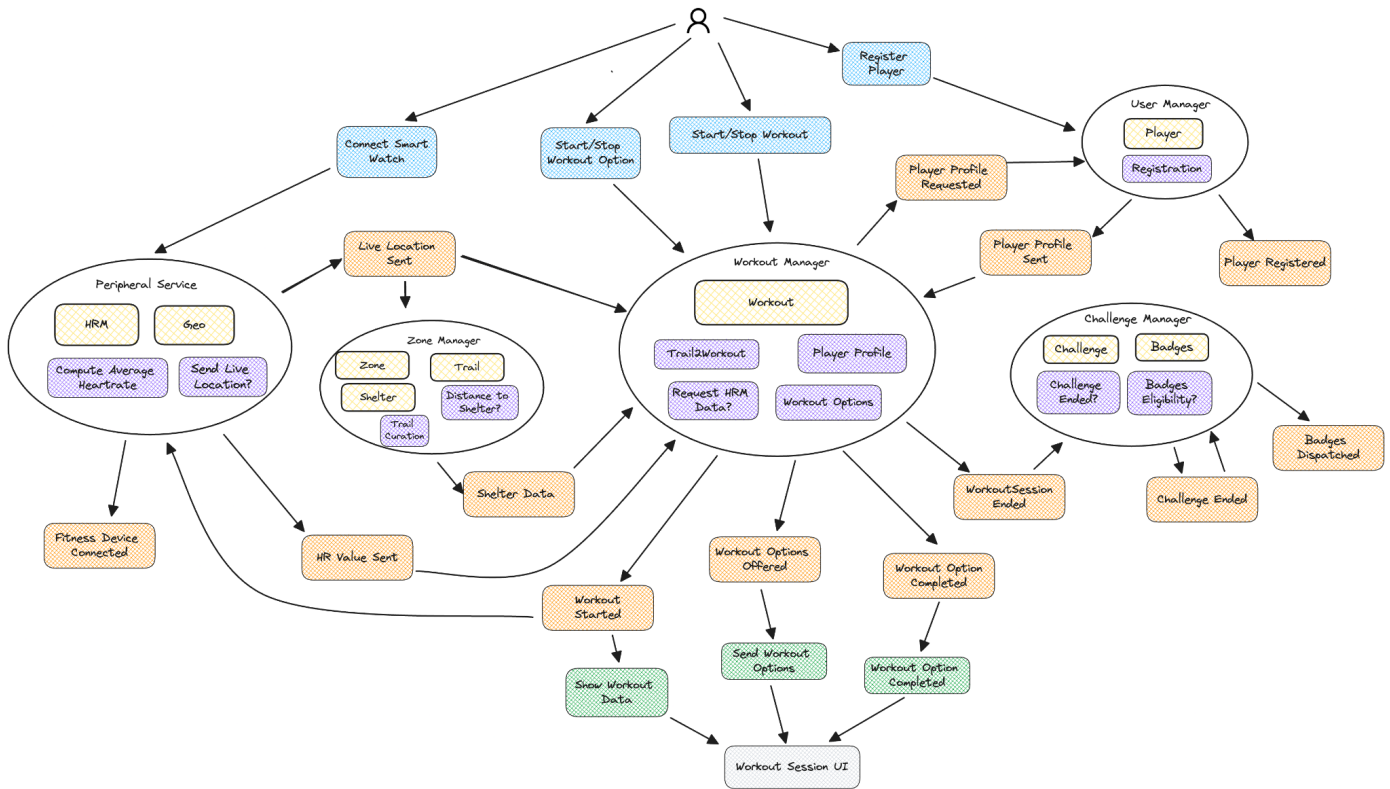


Fig. 1: Bounded Contexts

Justification for Bounded Contexts

- 1) *User Manager*: This context encapsulates user-related functionalities, to allow distinguishing between player and potential administrative entities. It's separated due to the clear distinction in user roles, data storage, and access privileges.
- 2) *Zone Manager*: This context handles the administration of zones, trails and shelters. It also determines the closest shelter locations based on live location data. It's distinct due to its unique role in managing the geographical zones and shelter-related information on the trails.
- 3) *Peripheral Service*: This context focuses on gathering and managing peripheral data like live location and heart rate. It's a distinct boundary because it deals with external device integration and data related to the user's physical activity.
- 4) *Workout Manager*: Being the core context, it manages workouts by calculating and offering different options such as sheltering, fighting back and escaping during a workout session. It's a separate context because it combines user preference, age, and average heart rate to provide the options on the fly.
- 5) *Challenge Manager*: This context acts as the achievement system of our application, managing the various challenges and assigning badges. It's distinct due to its role in assessing workout statistics and rewarding badges.

III. Hexagonal Architecture

The hexagonal architecture for our system is given below:

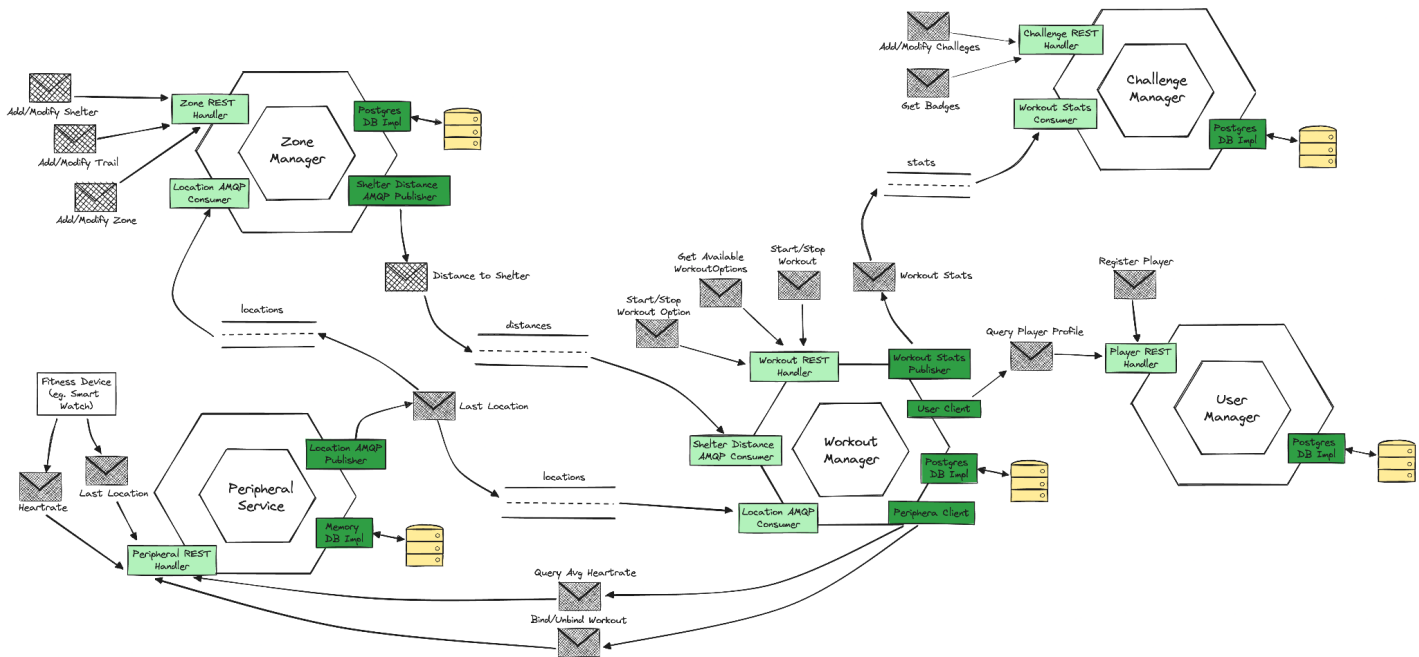


Fig. 2: Hexagonal Architecture

1. User Manager

The user manager is an integral component of our application, primarily designed to manage player-related operations. It encapsulates functionalities like listing all players, registering new players, retrieving, updating, and potentially deleting player information. Notably, the service was architecturally designed to extend its capabilities to include different user roles such as administrators.

Primary Adapters

- *Player REST Handler*: It caters to the requests of adding, retrieving, deleting and modifying a player, and potentially can be expanded for admin users.

Secondary Adapters

- *User Repository*: It acts as the data access layer, managing interactions with the database for operations such as creating, updating, fetching, and deleting users.

2. Zone Manager

The primary function of this service is to manage zones, as well as the curation of trails and shelters in the respective zone. A key feature of this service is its ability to compute the closest trail to a user based on their live location. Additionally, during workouts, the service is capable of calculating the distance from the user's current, near real-time location to the nearest shelter. This functionality is crucial for features such as guiding users to the nearest safe zones during their workouts and enhancing the user experience.

Primary Adapters

- *Zone REST Handler*: It serves as the primary interface for administrative tasks and user interactions. It allows administrators to create, fetch, modify, and delete zones, trails, and shelters. Additionally, it provides functionality for users to find the closest trail based on their current location, to enhance the overall user experience.
- *Location AMQP Consumer*: It is dedicated to receiving live location updates of users from the Peripheral Service. This real-time data is crucial for dynamically calculating distances to shelters during workouts.

Secondary Adapters

- *Zone Repository*: It functions as the storage component for the Zone Manager, holding the zones, trails, and shelters. This repository ensures that all geographic information related to zones, trails and shelters is stored and readily accessible for the various operations.
- *Shelter Distance AMQP Publisher*: It is responsible for sending the calculated distances to the nearest shelter to the Workout Service during workouts. This communication is key to integrating the trail data with the workout sessions, providing a seamless and informed workout experience.

3. Peripheral Service

The core logic of the Peripheral Service is centered on efficiently processing real-time peripheral data including heart rate and location from external devices such as mobile phones, smart watches, heart rate monitors (HRMs) etc. to optimize the workout experience. It aggregates heart rate metrics and the live location of the user to tailor workout plans based on their performance.

Primary Adapters

- *Peripheral REST Handler*: It manages interactions related to HRMs, live location and workout sessions. It handles connecting and disconnecting HRMs, binding and unbinding workouts, and calculating average heart rate data. It enables fitness devices such as smart watches, mobile phones, etc to transmit heart rate data and live location to the Peripheral Service.

Secondary Adapters

- *Peripheral Repository*: It acts as the data storage component, maintaining records of live locations and heart rates collected from external devices. This database is essential for historical data analysis and computing metrics like average heart rates.
- *Location AMQP Publisher*: It is responsible for broadcasting live location data to two queues, one for the Workout Manager facilitating the incorporation of live location data into workout options generation and the other for the Zone Manager, for calculating distance to shelters.

4. Workout Manager

The core logic involves managing workouts, customizing experiences based on real-time data (like location and heart rate), and maintaining detailed workout records for user analysis and engagement. This includes decision-making processes for workout options, influenced by user data and peripheral inputs, ensuring a responsive and personalized workout experience. The service dynamically computes workout options based on various user-specific and workout-related parameters. It first retrieves workout options and details from the repository, followed by calculations involving the user's preference (cardio or strength training), current workout statistics (total enemies fought and escaped), and heart rate metrics(only average heart rate is used). The algorithm then integrates these data points to adjust workout options, determining the ranking of the options. Notably, the algorithm's pluggable nature allows for future enhancements (like plugging in an AI Mega-mind) and refinement of the logic, offering flexibility to adapt to different workout profiles and user preferences.

Primary Adapters

- *Workout REST Handler*: It facilitates interactions with external clients by handling requests related to workout activities. This includes operations such as initiating, stopping, and updating workouts, as well as providing access to various workout-related statistics.
- *Location AMQP Consumer*: It's dedicated to processing real-time location updates from the Peripheral Service, enabling the service to adapt workout options (specifically sheltering) based on the user's current location when hardcore is disabled.
- *Shelter Distance AMQP Consumer*: It focuses on receiving live shelter distance information from the Trail Manager during a workout, allowing the service to incorporate shelter-related decisions into the workout experience.

Secondary Adapters

- *Workout Repository*: It acts as the data access layer, managing interactions with the database for operations such as creating, fetching, updating, and deleting workouts and their options.
- *Peripheral Client*: It ensures communication with Peripheral Service, gathering data like average heart rate, integral for tailoring the workout options.
- *User Client*: It manages user-specific data including retrieving workout preferences and user age, contributing to a personalized workout regime.
- *Workout Stats AMQP Publisher*: It's responsible for publishing detailed workout statistics post-workout which can be used for computing achievements and user performance analysis if needed.

5. Challenge Manager

The Challenge Manager acts as the hub for administering global challenges and assigning badges when these challenges end. The Challenge Manager service operates like achievement systems in games [1], automatically enrolling all users in global challenges with each challenge defined by a specific criterion. Similar to the Observer Pattern [1], it continuously monitors user workout statistics, awarding badges upon meeting the challenge criterion. This approach simplifies challenge management akin to gaming achievements.

Primary Adapters

- *Challenge REST Handler*: It provides administrative operations for managing challenges as well as fetching player badges.
- *Workout Stats AMQP Consumer*: It receives workout stats post-workout, and feeds them to Challenge Manager to determine if the player has won a badge when the challenge ends.

Secondary Adapters

- *Challenge Repository*: It acts as the data access layer, managing interactions with the database for operations such as creating, fetching, updating, and deleting challenges and player badges.

IV. APIs

In this section, we'll cover essential asynchronous and synchronous APIs along with their justifications and briefly discuss the other APIs.

Key Asynchronous APIs

1. Last Locations Fan Out Queue

The choice for this to be asynchronous stems from the high frequency of incoming data from various fitness devices. Real-time calculations are crucial for Zone Manager and Workout Manager. As a REST synchronous call would block the Peripheral Service, opting for an asynchronous queue enables concurrent processing of the live location data. This ensures that the Peripheral Service can continue to handle incoming data without waiting for responses from the Zone Manager and Workout Manager, allowing efficient handling of real-time updates.

2. Shelter Distance Queue

As the application scales, the frequency of requests to determine the closest shelter can increase. To prevent potential blocking of the Zone Manager due to varying processing times of the Workout Manager, an asynchronous call is employed. This setup allows the Zone Manager to continue computing shelter distances without being dependent on the immediate processing times of the Workout Manager, ensuring responsiveness and scalability as the application grows.

3. Workout Stats Queue

Immediate feedback to the user is crucial when the workout ends. Hence, the Workout Manager publishes workout stats and continues with its task. The Challenge Manager consumes these stats and computes the badges asynchronously. Badge computation is decoupled from the user-facing interaction, ensuring that the user receives immediate workout stats while badges can be awarded or queried later without affecting the primary workout flow.

Key Synchronous APIs

1. *Get Average Heart Rate (Peripheral Service)*

As per the requirements, only the average heart is required for the workout option computation and it is an opt-in feature - options are computed only when requested. As a result, we chose this to be a synchronous call. Given that this is a vital data point, ensuring a synchronous call for retrieving this information allows the Workout Manager to have up-to-date average heart rate information. Additionally, it's a quick-read operation that doesn't impede the system. If the stream of heart rates becomes a need for computing workout options in the future, then this can easily become an asynchronous call.

2. *Bind Workout (Peripheral Service)*

This API call is synchronous as it's a crucial step in starting a workout. The process ties the workout to the fitness device ID, indicating that the workout has commenced. Blocking the process here ensures that the workout doesn't start until the necessary components (like live location and heart rate monitor) are securely tied up to the workout, avoiding any inconsistencies or incomplete data records at the onset of the workout.

3. *Unbind Workout (Peripheral Service)*

This API call is synchronous, as it methodically proceeds through a sequence of actions: first, it stops the monitoring of connected components (like live location and heart rate monitor), and then it concludes by allowing the workout to end. The de-registration process is designed to be quick and efficient, enables it to be synchronous and ensures the timely and accurate cessation of data collection for the workout, maintaining consistency in the system.

Other APIs

| players | | | ^ |
|---------|----------------------|-----------------------|---|
| PUT | /api/v1/player | Update Player | ▼ |
| GET | /api/v1/player/{id} | Get Player by ID | ▼ |
| GET | /api/v1/players | List Players | ▼ |
| POST | /api/v1/players | Create a Player | ▼ |
| DELETE | /api/v1/players/{id} | Delete a Player by ID | ▼ |

Fig. 3: User Manager APIs using Swagger

| zone | | | ^ |
|--------|--------------------------------------------------------------|------------------------------------------------|---|
| POST | /api/v1/zone | Create a zone | ▼ |
| PUT | /api/v1/zone/{zone_id} | Update a zone | ▼ |
| DELETE | /api/v1/zone/{zone_id} | Delete a zone | ▼ |
| GET | /api/v1/zone/{zone_id}/trail | Get the closest trail | ▼ |
| POST | /api/v1/zone/{zone_id}/trail | Create a trail | ▼ |
| GET | /api/v1/zone/{zone_id}/trail/{trail_id} | Get location information of a specific trail | ▼ |
| PUT | /api/v1/zone/{zone_id}/trail/{trail_id} | Update a trail | ▼ |
| DELETE | /api/v1/zone/{zone_id}/trail/{trail_id} | Delete a specific trail | ▼ |
| GET | /api/v1/zone/{zone_id}/trail/{trail_id}/shelter | Get the closest shelter information | ▼ |
| POST | /api/v1/zone/{zone_id}/trail/{trail_id}/shelter | Create a shelter | ▼ |
| GET | /api/v1/zone/{zone_id}/trail/{trail_id}/shelter/{shelter_id} | Get location information of a specific shelter | ▼ |
| PUT | /api/v1/zone/{zone_id}/trail/{trail_id}/shelter/{shelter_id} | Update a shelter | ▼ |
| DELETE | /api/v1/zone/{zone_id}/trail/{trail_id}/shelter/{shelter_id} | Delete a shelter | ▼ |

Fig. 4: Zone Manager APIs using Swagger


| peripheral | | | ^ |
|------------|-------------------------------------|---------------------------------------|-----------------------------------------------------------------------------------------|
| PUT | /api/v1/geo/:geo_id | Set live location (geo reading) | ▼ |
| PUT | /api/v1/hrm/:hrm_id | Set HRM device reading | ▼  |
| PUT | /api/v1/peripheral | Unbind peripheral data from a workout | ▼ |
| POST | /api/v1/peripheral | Bind peripheral to a workout | ▼ |
| POST | /api/v1/peripheral/hrm | Connect to HRM device | ▼ |
| PUT | /api/v1/peripheral/hrm/{hrmid} | Disconnect HRM device | ▼ |
| GET | /api/v1/peripheral/hrm/{workout_id} | Get average heart rate | ▼ |

Fig. 5: Peripheral Service APIs using Swagger

| workout | | | ^ |
|---------|-------------------------------------|-----------------------------------|---|
| POST | /api/v1/workout | Start a new workout session | ▼ |
| GET | /api/v1/workout/distance | Get distance covered in a workout | ▼ |
| GET | /api/v1/workout/escapes | Get escapes made in a workout | ▼ |
| GET | /api/v1/workout/fights | Get fights fought in a workout | ▼ |
| GET | /api/v1/workout/shelters | Get shelters taken in a workout | ▼ |
| PUT | /api/v1/workout/{workoutId} | Stop an ongoing workout session | ▼ |
| DELETE | /api/v1/workout/{workoutId} | Delete a workout session | ▼ |
| GET | /api/v1/workout/{workoutId}/options | Get workout session options | ▼ |
| POST | /api/v1/workout/{workoutId}/options | Start a workout option | ▼ |
| PATCH | /api/v1/workout/{workoutId}/options | Stop a workout option | ▼ |

Fig. 6: Workout Manager APIs using Swagger

| badges | | | ^ |
|------------|-------------------------|--------------------------|---|
| GET | /api/v1/badges | List Badges by Player ID | ▼ |
| challenges | | | ^ |
| GET | /api/v1/challenges | List Challenges | ▼ |
| POST | /api/v1/challenges | Create a Challenge | ▼ |
| GET | /api/v1/challenges/{id} | Get Challenge by ID | ▼ |
| PUT | /api/v1/challenges/{id} | Update Challenge | ▼ |
| DELETE | /api/v1/challenges/{id} | Delete a Challenge by ID | ▼ |

Fig. 7: Challenge Manager APIs using Swagger

V. Potential Service Patterns

1. Load Balancer Pattern

As the user base grows, particularly during peak usage times, the demand for services such as the Peripheral Service, Workout Manager, and Zone Manager can vary greatly. Implementing a Load Balancer pattern for these services is crucial for efficiently distributing incoming requests across multiple instances or servers. For the Peripheral Service, this is particularly important due to the constant influx of data like heart rates and live location, which require real-time processing. Similarly, the Zone Manager, tasked with handling live location data and dynamically computing the distance to the closest trails, also experiences varying loads. A load balancer will ensure that these services can handle high volumes of location updates and biometric data without compromising performance. This approach prevents overload on any single server, enhancing overall system responsiveness and maintaining smooth user experiences during high-traffic events such as 'HalloweeK' or 'Marathon Rush'. By balancing the load, the system can sustain high performance and reliability, crucial for a fitness platform where timely processing of workout and location data is essential.

2. Circuit Breaker Pattern

The Workout Manager, upon completing a workout session, sends workout statistics to the Challenge Manager via the Workout Stats Queue. The Challenge Manager assesses these stats for achievements. The AMQP service may go down at times. When the Workout Service attempts to send workout stats to the Workout Stats Queue, the Circuit Breaker monitors the health of this communication link. If the link is broken, then it can cache the workout stats locally and continue operations as usual. It would monitor the communication link to AMQP and once it is back it will send all messages in the cache and fall back to the original way of sending workout stats. The key benefit of doing this is to preserve the workout stats and make sure the challenges are evaluated for each workout.

3. API Gateway Pattern

An API Gateway could be introduced at the interface between the User Manager, Workout Service, Peripheral Service, and the external world, acting as a unified entry point for all external client requests. This gateway could handle authentication, request routing, and protocol translation for various client types (mobile, fitness devices, etc.). By incorporating an API Gateway, you'd centralize the management of incoming and outgoing requests, streamlining the communication between clients and your system. However, this pattern has the downside of being a single-point failure. Implementing a *robust* gateway with redundancy is crucial.

VI. Other Implementation Details

1. HATEOAS Implementation: The Start Workout API uses HATEOAS [2] to return a link to get the workout options.

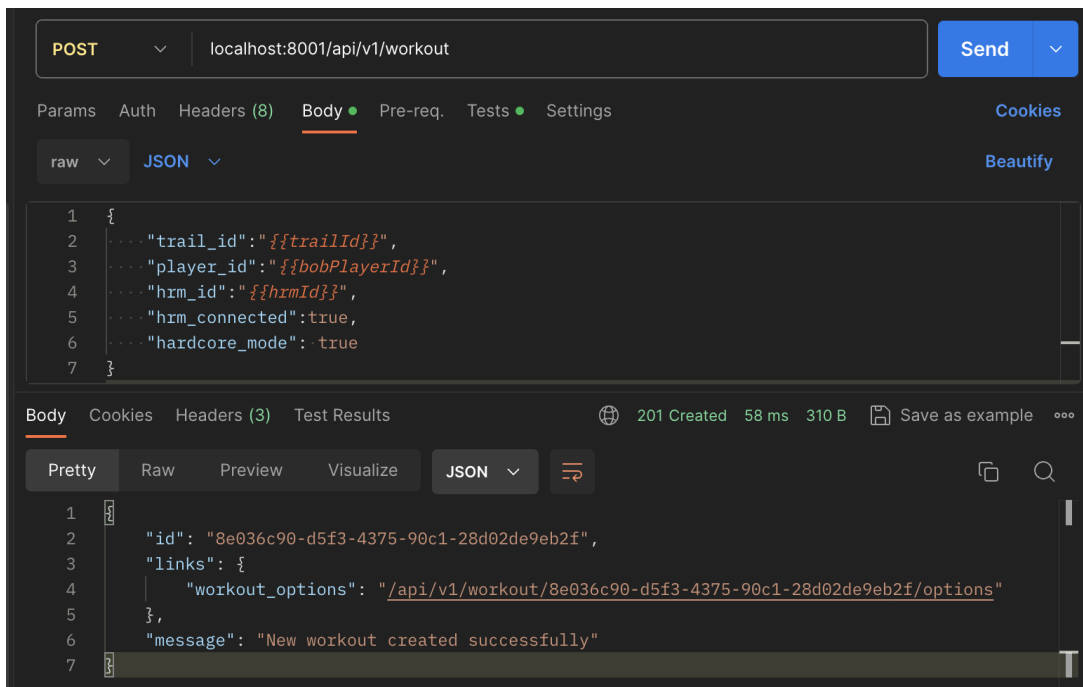


Fig. 7: HATEOAS in Start Workout API

2. Logger Implementation: We employed a sophisticated logger - the Uber Zap Logger - tailored to suit both development and production environments. We tried to follow the best logging practices [2] and used its multiple logging levels like Info, Debug, Error, Panic and Fatal for nuanced logging control throughout our application. Its key features like customizable time formatting and environment-specific configurations, allowed effective monitoring, troubleshooting, and logging of our application behavior across different scenarios.
3. Configuration Management: To follow the basic principles of software configuration management [4], our configuration is designed to centralize and simplify access to various environmental settings for each service. It dynamically loads configurations for different components like PostgreSQL and RabbitMQ from environment variables, with sensible defaults for each parameter. This approach facilitates easy adjustments of configurations without modifying the codebase, making the system highly adaptable to both development and production environments. The configuration covers essential parameters such as hostnames, ports, credentials, and logging levels, and is initialized at startup, ensuring that all parts of the application have consistent access to the latest configuration settings. This design promotes flexibility, maintainability, and scalability of the application's infrastructure.

VII. Conclusion

In conclusion, we delineate our architecture for the ACME Run application, focusing on its modularity, user-centric functionality, as well as scalability. We designed our architecture using bounded contexts to clearly segregate service boundaries and then used hexagonal architecture to describe the relationship between the different services obtained and their intricacies. We justified our architectural choices based on their ability to facilitate a decoupled, highly cohesive, scalable, and maintainable system, complemented by a comprehensive analysis of our key APIs. We also explored the potential use of service patterns like Load Balancers and Circuit Breakers to create a resilient and scalable system. Lastly, we discussed the other implementation details like HATEOAS, logging and configuration management.

VIII. References

1. Nystrom, R. (2014). Observer Pattern. In Game Programming Patterns (p. Genever Benning). Retrieved from <https://gameprogrammingpatterns.com/observer.html>
2. Wikipedia contributors. (2023, November 13). HATEOAS. Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/HATEOAS>
3. S. Gu, G. Rong, H. Zhang and H. Shen, "Logging Practices in Software Engineering: A Systematic Mapping Study," in IEEE Transactions on Software Engineering, vol. 49, no. 2, pp. 902-923, 1 Feb. 2023, doi: 10.1109/TSE.2022.3166924
4. Kirk, R. (2002). Software Configuration Management Principles and Best Practices. In: Oivo, M., Komi-Sirviö, S. (eds) Product Focused Software Process Improvement. PROFES 2002. Lecture Notes in Computer Science, vol 2559. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-36209-6_26