# **Messaging Module**

## **1. Overview**

The **Messaging Module** provides a robust, internal communication system for users within the marketplace. This module enables direct, one-on-one conversations that are always contextualized by a specific product. This design ensures that all discussions, such as inquiries, price negotiations, or offers, remain relevant and easily accessible.

The module is built with **Clean Architecture** as its guiding principle, ensuring a clear separation of concerns across its layers: Entities, Repositories, Serializers, and Services. This approach makes the module highly **maintainable, reusable, and testable**.

## **2. Architecture Layers**

### **2.1 Entities (core/entities/)**

Entities are the core business models and are framework-agnostic. They are pure data representations used across the module.

* ConversationEntity: Represents a unique chat channel.
  + **Fields**: id, product, participants (list of users), created\_at.
* MessageEntity: Represents a single message within a conversation.
  + **Fields**: id, conversation, sender, content, timestamp, is\_read.

### **2.2 Repositories (core/repositories/)**

The repository layer abstracts all database operations, insulating the business logic from the persistence layer.

* ConversationRepository: Handles database interactions for conversations.
  + **Methods**: create\_conversation(), get\_conversation\_by\_product\_and\_users(), get\_conversations\_for\_user().
* MessageRepository: Manages database operations for messages.
  + **Methods**: create\_message(), get\_messages\_for\_conversation(), mark\_messages\_as\_read().

### **2.3 Services (core/services/)**

The service layer contains the core business logic. It orchestrates interactions between repositories and serializers.

* MessageService:
  + start\_or\_get\_conversation(): Checks for an existing conversation between two users for a given product. If one doesn't exist, a new one is created.
  + send\_message(): Validates the sender and conversation, then persists the new message via the repository.
  + mark\_as\_read(): Updates the read status of all messages for a specific user within a conversation.

### **2.4 Serializers (api/serializers/)**

Serializers are responsible for data transformation, converting internal entities into a format suitable for API consumption, such as JSON.

* ConversationSerializer: Formats conversation data for the API response, including details about participants, the associated product, and a snippet of the last message.
* MessageSerializer: Formats message data, including the sender, content, timestamp, and read status.

## **3. Operational Workflow**

### **Conversation Initiation**

1. A user attempts to start a conversation on a product.
2. The MessageService is called to check for an existing conversation between the two users for that specific product.
3. The ConversationRepository performs a lookup.
4. If a conversation exists, it is returned. If not, a new Conversation record is created, and the new ConversationEntity is returned.

### **Message Exchange**

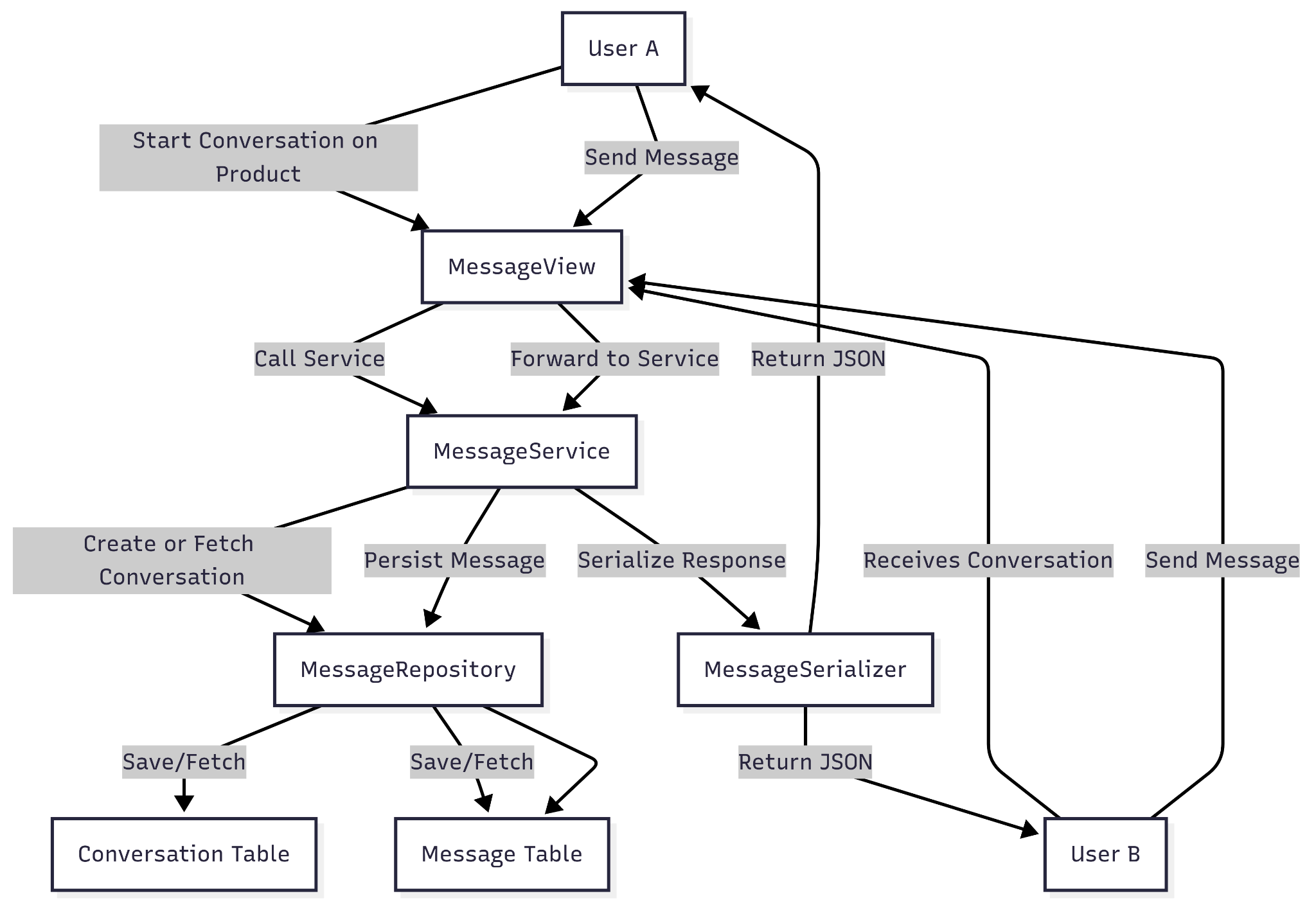
1. A user sends a message to an existing conversation via an API endpoint.
2. The MessageService receives the request, validates the sender and conversation.
3. The MessageRepository persists the new message to the database.
4. The MessageService uses the MessageSerializer to format the message and returns the JSON response to both participants.

### **Read Receipts**

1. When a user opens a conversation, a request is sent to an API endpoint.
2. The MessageService is invoked to update the read status.
3. The MessageRepository marks all messages in that conversation as read for the viewing user.
4. The MessageSerializer includes the is\_read flag in all subsequent API responses for messages.

## **4. Diagram**

The following flowchart illustrates the complete data flow within the module, from a user action to the database.



## **5. Integration and Extensibility**

### **Integration Points**

* **Authentication**: All API endpoints for this module require user authentication to ensure only authorized users can access conversations.
* **Product Module**: The Messaging Module is deeply integrated with the Product Module, as every conversation is tied to a specific product, and the product ID serves as a foreign key.
* **Notification System**: This module can be easily extended to integrate with an external notification system to send push, SMS, or email notifications for new messages.

### **Extensibility**

The module's clean design allows for easy addition of new features without a major refactor:

* **Real-time Communication**: A WebSocket layer can be integrated to provide a live chat experience.
* **Media Attachments**: The MessageEntity and MessageRepository can be extended to support file uploads (images, PDFs, etc.).
* **Reactions**: The MessageEntity can be extended to support likes or other emoji reactions.
* **Group Conversations**: The current two-user conversation model can be extended to handle group chats by modifying the participants field in the ConversationEntity.