Replyke Documentation

Overview

Replyke is designed to simplify the development and implementation of social interaction features by offloading much of the backend work. This allows developers to focus on the frontend experience and reduces complexity. Unlike traditional architectures where all interactions pass through a central server, Replyke communicates directly with the client. This approach reduces complexity for developers but introduces unique challenges in ensuring that data remains valid across various custom use cases—especially when dealing with free-form metadata and content validation.

Replyke addresses many common validation needs out of the box, such as enforcing data ownership, managing user authorization, and implementing logical constraints like limiting users to a single vote per entity/comment. However, for application-specific validation—particularly for custom metadata or complex entity rules—developers need a way to extend Replyke's built-in capabilities.

To solve this, Replyke leverages a webhook-based system. Developers can define custom validation logic by implementing webhooks triggered during the creation or updating of users and entities. This ensures that all relevant data is validated on the developer's server before the operation is finalized in Replyke. The webhook response determines whether the operation proceeds or is rejected, allowing for fine-grained control over data integrity.

By default, no webhooks are configured for new projects, meaning that no further validation takes place beyond Replyke's built-in validation. While this is sufficient for development, implementing webhooks is highly recommended for applications in production that require enhanced security and data validation to maintain the integrity of their data.

Webhooks for Validation

Replyke provides developers with the ability to set up four validation webhooks. These webhooks allow developers to implement custom validation logic for both user and entity operations. A single shared secret from the Replyke dashboard is sufficient for all webhooks. However, it is recommended to periodically rotate the secret for enhanced security.

For reference, the setup for validation webhooks is similar to the notifications webhook integration described in App Notifications Webhook Integration, with the added requirement of sending a signed response.

Validation Webhook Endpoints

1. **User Created** This webhook validates user details before a user is created. The payload includes the following fields:

Field	Description
`projectId`	Always included
`data`	Object containing the user details:
	- `foreignId`: If integrating Replyke with an external user system, a foreign ID will be provided.
	- `role`: Always “visitor” for new users
	- `email`: Optional, user’s email address
	- `name`: Optional, user’s name
	- `username`: Optional, username
	- `avatar`: Optional, URL to the avatar image
	- `bio`: Optional, user biography
	- `location`: Optional, user’s location
	- `birthdate`: Optional, user’s birthdate
	- `metadata`: Optional, free-form metadata
	- `secureMetadata`: Optional, sensitive data

2. **User Updated** This webhook validates user details before a user is updated. The payload structure is similar to the one used for user creation but includes only the updated fields within the `data` object.
3. **Entity Created** This webhook validates entity details before an entity is created. The payload includes the following fields:

Field	Description
`projectId`	Always included
`data`	Object containing entity details:
	- `foreignId`: If entity data is attached to a resource from an external data set, a foreign ID will be provided.
	- `resourceId`: Entities could be grouped by resource ID to separate feeds.
	- `userId`: Optional, creator's ID
	- `title`: Optional, title of the entity
	- `content`: Optional, content of the entity
	- `attachments`: Optional, file attachments
	- `mentions`: Optional, array of user mentions
	- `keywords`: Optional, keywords for the entity
	- `location`: Optional, geographic location
	- `metadata`: Optional, free-form metadata
`initiatorId`	Optional, ID of the user initiating the action

4. **Entity Updated** This webhook validates entity details before an entity is updated. The payload structure is similar to the one used for entity creation but includes only the updated fields within the `data` object.

HMAC Signature and Security

To ensure secure communication between Replyke and your server, each webhook request includes an HMAC signature. This signature verifies the authenticity of the request and prevents tampering. The HMAC signature is calculated using the shared secret and the payload.

Supporting Functions

Here are supporting functions for HMAC validation and response signing:

```
import crypto from "crypto";
import { Request, Response } from "express";
```

```ts
/**
 * Validate HMAC signature of an incoming request.
 */
export function validateIncomingHmac(req: Request, secret: string): void {
  const signature = req.headers["x-signature"] as string;
  const timestamp = req.headers["x-timestamp"] as string;

  if (!signature || !timestamp) {
    throw new Error("Missing HMAC signature or timestamp in headers");
  }

  // Reject requests older than 5 minutes to prevent replay attacks
  if (Date.now() - parseInt(timestamp, 10) > 5 * 60 * 1000) {
    throw new Error("Request timestamp expired");
  }

  // Compute the expected signature
  const payload = JSON.stringify(req.body);
  const expectedSignature = crypto
    .createHmac("sha256", secret)
    .update(`${timestamp}.${payload}`)
    .digest("hex");

  if (signature !== expectedSignature) {
    throw new Error("Invalid HMAC signature");
  }
}

/**
 * Generate an HMAC signature for the response payload.
 */
export function signResponsePayload(payload: any, secret: string): string {
  return crypto
    .createHmac("sha256", secret)
    .update(JSON.stringify(payload))
    .digest("hex");
}

/**
 * Handles unexpected errors, logs them, signs the error response, and sends it.
 */
export function handleError(
  res: Response,
  error: any,
  sharedSecret: string
): void {
  console.error("Error processing request:", error.message || error);

  const responsePayload = { valid: false, message: "Server error" };
  const responseSignature = signResponsePayload(responsePayload, sharedSecret);

  res
    .status(500)
```

```
    .header("X-Response-Signature", responseSignature)
    .send(responsePayload);
}
```

Example Implementation: User Creation Webhook

```
import { Request as ExReq, Response as ExRes } from "express";
import {
  validateIncomingHmac,
  signResponsePayload,
  handleError,
} from "./utility-functions";

export default async (req: ExReq, res: ExRes) => {
  const sharedSecret = process.env.REPLYKE_PROJECT_SECRET;

  try {
    const { projectId, data } = req.body;

    // Step 1: Validate the incoming HMAC signature
    validateIncomingHmac(req, sharedSecret!);

    // Step 2: Add your user-specific validation logic here
    console.log("Project ID:", projectId, "User Data:", data);

    const responsePayload = { valid: true };
    const responseSignature = signResponsePayload(
      responsePayload,
      sharedSecret!
    );

    res
      .status(200)
      .header("X-Response-Signature", responseSignature)
      .send(responsePayload);
  } catch (err) {
    handleError(res, err, sharedSecret!);
  }
};
```

Example Implementation: Entity Creation Webhook

```
import { Request as ExReq, Response as ExRes } from "express";
import {
  validateIncomingHmac,
  signResponsePayload,
  handleError,
```

```
} from "./utility-functions";

export default async (req: ExReq, res: ExRes) => {
  const sharedSecret = process.env.REPLYKE_PROJECT_SECRET;

  try {
    const { projectId, data, initiatorId } = req.body;

    // Step 1: Validate the incoming HMAC signature
    validateIncomingHmac(req, sharedSecret!);

    // Step 2: Add your entity-specific validation logic here
    console.log(
      "Project ID:",
      projectId,
      "Entity Data:",
      data,
      "Initiator ID:",
      initiatorId
    );

    const responsePayload = { valid: true };
    const responseSignature = signResponsePayload(
      responsePayload,
      sharedSecret!
    );

    res
      .status(200)
      .header("X-Response-Signature", responseSignature)
      .send(responsePayload);
  } catch (err) {
    handleError(res, err, sharedSecret!);
  }
};
```

These examples demonstrate complete webhook implementations and supporting functions, ensuring secure and accurate handling of data. Developers using other languages should replicate the HMAC signature logic to maintain compatibility.

Last updated on May 7, 2025