1. Problem Statement: A Day at "FreshMart" Grocery Stores

The Crisis at FreshMart

FreshMart, a bustling grocery chain, launched a digital loyalty program to reward customers. But chaos ensued:

- **Missing Data:** Customers tried to redeem points, but the app crashed if their profile was incomplete.
- **Invalid Requests:** A customer entered "1000 points" instead of "100," redeeming $1,000 by accident.
- **Confusing Responses:** Staff couldn't tell if a request failed due to a server error or invalid input.

The breaking point:
A customer named Sarah tried to redeem points for a birthday cake. The app accepted her request but didn't check if she had enough points. The system deducted 500 points (her entire balance), but the cake order failed because the store was out of stock. Sarah left furious, and FreshMart's reputation tanked.

Your mission:
Redesign FreshMart's loyalty API to:

1. Validate every request.
2. Ensure responses are clear and structured.
3. Prevent invalid data from causing errors.

2. Learning Objectives

By the end of this lesson, you'll be able to:

- Type request bodies, query params, and responses with TypeScript.
- Validate JSON data using schemas and middleware.
- Return consistent error messages and status codes.
- Use advanced Express patterns for type safety.
- Prevent bugs like Sarah's mishap through strict validation.

3. Concept Introduction with Analogy

Analogy: The Restaurant Kitchen

Imagine a restaurant kitchen:

- **Waiters (Requests):** Take orders from customers.
- **Chefs (API Logic):** Prepare meals based on orders.
- **Quality Control (Validation):** Checks orders for completeness (e.g., "No gluten, please").
- **Server (Responses):** Delivers meals or explains why an order can't be fulfilled.

How it maps to FreshMart's API:

- If a waiter (request) doesn't specify "no gluten" (missing data), the kitchen (API) might serve a dish that makes the customer sick.
- If the chef (API) doesn't check stock (validation), they might promise a cake they can't deliver.
- Quality control (TypeScript types) ensures every order (request) has the right details before the kitchen starts cooking.

A. Typing Requests and Responses

1. Defining Strict Interfaces

Problem: Without types, requests like `{ points: "100" }` (string) might be processed as `100` (number), causing bugs.
Solution: Use TypeScript interfaces to enforce data shapes.

```
// Loyalty program interfaces
interface RedeemRequest {
customerId: string;
points: number;
}

interface ApiResponse<T = any> {
status: "success" | "error";
data?: T;
error?: string;
}
```

2. Typing Express `Request` and `Response`

Use generics to lock down body, query, and params:

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

router.post(
"/redeem",
(req: Request<{}, {}, RedeemRequest>, // Body type
 res: Response<ApiResponse<{ remainingPoints: number }>>) => { // Response type
  // ...logic...
}
);
```

Why?

- `req.body` is now strictly typed. Passing `{ points: "100" }` will throw a TypeScript error.
- The response shape is predefined, ensuring consistency.

B. Validating Data with Zod

1. Schema Validation

Problem: TypeScript types are stripped at runtime. Malicious clients can still send invalid data.
Solution: Use Zod to validate requests at runtime.

```
import { z } from "zod";

const RedeemSchema = z.object({
customerId: z.string().uuid(),
points: z.number().int().positive(),
});

router.post("/redeem", (req, res) => {
const result = RedeemSchema.safeParse(req.body);
if (!result.success) {
  return res.status(400).json({
    status: "error",
    error: result.error.errors[0].message,
  });
}
// Proceed with validated data (result.data)
});
```

2. Centralized Validation Middleware

Avoid repeating validation logic:

```
import { RequestHandler } from "express";

function validate<T extends z.ZodTypeAny>(schema: T): RequestHandler {
  return (req, res, next) => {
    const result = schema.safeParse(req.body);
    if (!result.success) {
      return res.status(400).json({
        status: "error",
        error: result.error.errors[0].message
      });
    }
    req.body = result.data; // Override req.body with parsed data
    next();
  };
}

// Usage
router.post("/redeem", validate(RedeemSchema), (req, res) => {
  // req.body is now validated and typed!
});
```

C. Structured Error Handling

1. Error Hierarchy

```
Define custom errors for clarity:

class ApiError extends Error {
  constructor(
    public statusCode: number,
    message: string,
    public details?: any
  ) {
    super(message);
  }
}

class InsufficientPointsError extends ApiError {
  constructor() {
    super(400, "Insufficient points");
  }
}
```

2. Global Error Middleware

Catch all errors and send consistent responses:

```
app.use((err: Error, req: Request, res: Response, next: NextFunction) => {
  if (err instanceof ApiError) {
    res.status(err.statusCode).json({
      status: "error",
      error: err.message,
      details: err.details,
    });
  } else {
    // Log unexpected errors but don't expose details
    console.error(err);
    res.status(500).json({
      status: "error",
      error: "Internal server error"
    });
  }
});
```

Usage in routes:

```
router.post("/redeem", validate(RedeemSchema), (req, res) => {
  const member = findMember(req.body.customerId);
  if (member.points < req.body.points) {
    throw new InsufficientPointsError(); // Caught by global middleware
  }
  // ...redeem points...
});
```

D. Advanced: Custom Request Properties

1. Extending `Request` for Authentication

Add user data to `req` after authentication:

```
declare global {
  namespace Express {
    interface Request {
      user?: {
        id: string;
        role: "customer" | "admin";
      };
    }
  }
}

// Auth middleware
router.use((req, res, next) => {
  const token = req.headers.authorization;
  if (token) {
    req.user = decodeToken(token); // Assume this returns a user object
  }
  next();
});

// Usage in routes
router.get("/profile", (req, res) => {
  if (!req.user) {
    throw new ApiError(401, "Unauthorized");
  }
  res.json({ status: "success", data: req.user });
});
```

E. Real-World Example: Full Redeem Endpoint

```
router.post(
  "/redeem",
  validate(RedeemSchema),
  async (req: Request<{}, {}, RedeemRequest>, res: Response) => {
    const { customerId, points } = req.body;

    // Check customer exists
    const member = await db.loyaltyMembers.findOne({ customerId });
    if (!member) {
      throw new ApiError(404, "Customer not found");
    }

    // Check points
    if (member.points < points) {
      throw new InsufficientPointsError();
    }

    // Check item availability
    const item = await db.inventory.findOne({ sku: "CAKE" });
    if (!item || item.stock < 1) {
      throw new ApiError(409, "Item out of stock");
    }

    // Deduct points and update inventory
    await db.loyaltyMembers.updateOne(
      { customerId },
      { $inc: { points: -points } }
    );
    await db.inventory.updateOne({ sku: "CAKE" }, { $inc: { stock: -1 } });

    // Success!
    res.json({
      status: "success",
      data: {
        customerId,
        remainingPoints: member.points - points,
        item: "CAKE"
      },
    });
  }
);
```

5. Step-by-Step Data Flow

1. **Request Received:**
    - Client sends `POST /redeem` with `{ customerId: "123", points: 500 }`.
2. **Validation Middleware:**
    - Zod checks if `points` is a positive integer.
    - If invalid, returns `400 Bad Request` with error details.
3. **Authentication Middleware:**
    - Checks `Authorization` header and attaches `req.user`.

4. **Business Logic:**
    - Checks customer existence, points, and inventory.
    - Throws specific errors for each failure case.
5. **Global Error Handler:**
    - Catches errors and sends structured responses.
6. **Success Response:**
    - Returns `200 OK` with remaining points and redeemed item.

6. Interactive Challenge

Your Turn!

1. Add a `POST /transfer` endpoint allowing customers to transfer points to another account.
2. Validate:
    - Both `fromCustomerId` and `toCustomerId` must be valid UUIDs.
    - `points` must be a positive integer.
    - The sender must have enough points.
3. Return appropriate errors for each failure case.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Using any for request bodies	Always type with interfaces/Zod schemas
Ignoring runtime validation	Use Zod even if TypeScript is enabled
Generic error messages	Throw specific errors (e.g., InsufficientPointsError)
Mixing response formats	Standardize responses with ApiResponse

8. Quick Recap & Key Takeaways

- **Type Everything:** From request bodies to custom `Request` properties.
- **Validate at Runtime:** TypeScript isn't enough-Zod catches malicious data.
- **Structured Errors:** Use custom errors and global middleware for consistency.
- **Centralized Logic:** Middleware for auth/validation reduces code duplication.

9. Optional: Programmer's Workflow Checklist

- Define Zod schemas for all incoming data.
- Use `ApiResponse` for all JSON responses.

- Throw `ApiError` or subclasses for known failures.
- Test edge cases: invalid UUIDs, negative points, etc.
- Log unexpected errors but don’t expose details.