#Routing Controllers, Middleware, and Request Validation

1. Problem Statement: The Overwhelmed Bakery

A Day at "Crumb & Craft Bakery"

Crumb & Craft Bakery started as a cozy neighborhood shop. Orders were simple:

- A customer would walk in, ask for a cake, and the bakers would jot down details by hand.
- Mistakes were rare, and everyone knew each other.

But after becoming a city-wide sensation, things spiraled:

- **Lost Orders:** Staff scribbled requests on sticky notes that got misplaced.
- **Wrong Assignments:** Sometimes, a bread order ended up in the cake station, or a gluten-free request was missed.
- **No Oversight:** New hires didn't know who handled which type of order, leading to confusion and delays.
- **Missed Details:** Orders with missing pickup dates or flavors slipped through, causing wasted ingredients and unhappy customers.

The breaking point:
During a holiday rush, the bakery received 500 online orders in an hour.

- **Missing Allergies:** A child with a nut allergy nearly got a peanut butter cake.
- **Inventory Chaos:** The system allowed orders for 1,000 croissants when only 100 were in stock.
- **Mixed Messages:** Some customers got confirmation emails; others heard nothing.

The challenge:
How do you redesign the bakery's workflow so that:

1. Every order goes to the right team.
2. Mistakes (like missing allergies or impossible quantities) are caught early.
3. Customers *always* get clear, timely updates.

2. Learning Objectives

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

- Organize workflows into specialized teams (routing controllers).
- Add checkpoints to catch errors early (middleware).
- Validate orders for accuracy before they're processed (request validation).

3. Concept Introduction with Analogy

Analogy: The Bakery Assembly Line

Imagine redesigning Crumb & Craft's kitchen into a well-oiled assembly line:

1. **Specialized Stations (Routing Controllers):**
    - **Order Desk:** Handles new orders and customer inquiries.
    - **Baking Team:** Manages inventory and bakes items.
    - **Quality Control:** Inspects orders before they go out.
2. **Checkpoints (Middleware):**
    - **Allergy Check:** Scans every order for dietary restrictions.
    - **Inventory Check:** Ensures stock levels are updated in real time.
    - **Logging:** Tracks how long each order takes to complete.
3. **Inspectors (Validation):**
    - **Order Form Check:** Rejects forms missing flavors, dates, or quantities.
    - **Stock Check:** Blocks orders that exceed available inventory.

What Is a Controller?

- In Express (and web frameworks in general), a **controller** is a module (usually a class or file) that groups related request handlers together.
- Each controller is responsible for a "resource" or "feature" (e.g., orders, inventory, users).
- This modular approach:
    - Makes code easier to read and maintain.
    - Prevents accidental overlap or confusion.
    - Allows teams to work on different features independently.

Without Controllers (Flat Routing):

```
// routes.js
app.post("/orders", ...);
app.get("/orders/:id", ...);
app.post("/inventory", ...);
app.get("/inventory/:item", ...);
// All logic is mixed together.
```

With Controllers (Modular Routing):

```
// OrderController.js
router.post("/", ...);
router.get("/:id", ...);

// InventoryController.js
router.post("/", ...);
router.get("/:item", ...);

// Each controller is mounted at its own base route.
app.use("/orders", OrderController);
app.use("/inventory", InventoryController);
```

A. Organizing Workflows with Routing Controllers

Problem: Orders were handled by anyone, leading to chaos.
Solution: Assign specialized teams to specific tasks.

1. Defining Teams (Controllers)

```
// Team 1: Order Desk
@Controller("/orders")
export class OrderController {
@Post("/")
createOrder(@Body() order: Order) {
  // Forward to baking team
}

@Get("/:id")
getOrderStatus(@Param("id") orderId: string) {
  // Check progress
}
}

// Team 2: Baking Team
@Controller("/baking")
export class BakingController {
@Post("/start")
startBaking(@Body() order: Order) {
  // Check inventory and bake
}
}
```

2. How It Works

- **`@Controller("/orders")`:** This team handles all requests starting with `/orders`.
- **`@Post("/")`:** The order desk's form for new orders.
- **`@Get("/:id")`:** Lets customers check their order status.

B. Adding Checkpoints (Middleware)

Problem: Mistakes like nut allergies or oversold stock slipped through.
Solution: Add checkpoints to inspect orders before they're processed.

1. Types of Checkpoints

Checkpoint	Purpose	Example
Allergy Check	Scans for nuts, gluten, etc.	allergyMiddleware
Inventory Check	Ensures stock levels are sufficient	inventoryMiddleware
Logging	Tracks order timelines	loggingMiddleware

2. Custom Checkpoint Example

```
// Allergy Check Middleware
@Middleware()
export class AllergyMiddleware implements ExpressMiddlewareInterface {
use(req: Request, res: Response, next: NextFunction) {
  const { ingredients } = req.body;
  if (ingredients.includes("peanuts")) {
    throw new Error("Peanut allergy alert!");
  }
  next();
}
}

// Attach to the order workflow
@UseBefore(AllergyMiddleware)
@Post("/orders")
createOrder(@Body() order: Order) { ... }
```

C. Validating Orders (Request Validation)

Problem: Orders with missing/wrong details wasted time and ingredients.
Solution: Validate orders before they enter the workflow.

1. Validation Rules

```
class Order {
  @IsDate()
  pickupDate: Date;

  @IsString()
  flavor: string;

  @IsInt()
  @Min(1)
  @Max(100)
  quantity: number;
}
```

2. Automatic Validation

```
@Post("/orders")
createOrder(@Body({ validate: true }) order: Order) {
  // Only runs if validation passes
}
```

Failed Validation Response:

```
{
  "status": "error",
  "error": "Quantity must be at least 1"
}
```

5. Step-by-Step Data Modeling & Code Walkthrough

Let's walk through how our "Crumb & Craft Bakery" solves its real-world chaos using controllers, step by step:

A. Defining the Order Data Structure

Remember how orders were once scribbled on sticky notes and got lost?
We'll fix that by defining a clear, digital structure for every order-so nothing is forgotten.

```
interface Order {
  id: string;
  customerName: string;
  flavor: string;
  quantity: number;
  pickupDate: string;
}
```

Explanation:

- Every order must have an ID (so it never gets mixed up), the customer's name, the cake flavor, quantity, and a pickup date.
- By making these fields required, we ensure that no order is missing crucial details (no more "mystery cakes" or missed birthdays).

B. Creating a Safe Place for Orders

Instead of sticky notes, we'll use a digital list where every order is stored and can be easily found by any team member.

```
const orders: Order[] = [
  { id: "1", customerName: "Maria", flavor: "vanilla", quantity: 2, pickupDate: "2024-07-10" },
];
```

Explanation:

- This array represents the bakery's "order book"-every new order gets added here, and staff can look up any order by its ID.

C. Building the Order Controller: The New "Order Desk"

We create a dedicated "Order Desk" team (controller) whose only job is to handle customer orders-no more confusion about who's responsible.

```
import { JsonController, Get, Post, Param, Body } from "routing-controllers";

@JsonController("/orders")
export class OrderController {
  @Get("/")
  getAll() {
    return orders;
  }

  @Get("/:id")
  getOne(@Param("id") id: string) {
    const order = orders.find(o => o.id === id);
    if (!order) {
      return { status: "error", error: "Order not found" };
    }
    return { status: "success", data: order };
  }

  @Post("/")
  create(@Body() order: Omit<Order, "id">) {
    const newOrder: Order = {
      ...order,
      id: (orders.length + 1).toString(),
    };
    orders.push(newOrder);
    return { status: "success", data: newOrder };
  }
}
```

Explanation:

- **`@Get("/")`:** Lets staff see all orders-no more lost requests.
- **`@Get("/:id")`:** Anyone can check the status of a specific order (e.g., "Is Maria's cake ready?").
- **`@Post("/")`:** New orders are added in a standard way, with all required details. The system assigns a unique ID, so two "Maria"s never get mixed up.

D. Registering the Controller in the Bakery's Workflow

Just as you'd tell new staff, "All cake orders go to the Order Desk," we register our controller so Express knows where to send each request.

```
import { JsonController, Get, Post, Param, Body } from "routing-controllers";

@JsonController("/orders")
export class OrderController {
  @Get("/")
  getAll() {
    return orders;
  }

  @Get("/:id")
  getOne(@Param("id") id: string) {
    const order = orders.find(o => o.id === id);
    if (!order) {
      return { status: "error", error: "Order not found" };
    }
    return { status: "success", data: order };
  }

  @Post("/")
  create(@Body() order: Omit<Order, "id">) {
    const newOrder: Order = {
      ...order,
      id: (orders.length + 1).toString(),
    };
    orders.push(newOrder);
    return { status: "success", data: newOrder };
  }
}
```

Explanation:

- This sets up the bakery's "front counter"-all order-related requests are routed to the OrderController, so nothing is misplaced.

6. Interactive Challenge / Mini-Project

Your Turn!

- Create a `BakingController` for `/baking` routes.
- Add a `POST /baking/start` endpoint to start baking an order.
- Add a `GET /baking/status/:id` endpoint to check the baking status of an order.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Mixing unrelated routes	Group related routes in controllers
Duplicating logic	Use shared services/inject dependencies
Not using base routes	Always prefix controllers with a base
Inconsistent responses	Standardize response format

8. Quick Recap & Key Takeaways

- **Controllers** group related routes for clarity and maintainability.
- **Modular routing** makes code easier to grow and debug.
- **Controller libraries** like `routing-controllers` add powerful features (decorators, DI, middleware).

9. Optional: Programmer’s Workflow Checklist

- Define controllers for each major resource or workflow.
- Use clear base routes (e.g., `/orders` , `/baking`).
- Standardize response formats.
- Register all controllers in your app entry point.
- Test each controller independently.

10. Coming up next

Learn how to add “checkpoints” (middleware) to your workflow-catching errors, logging actions, and ensuring every order is safe before it reaches the kitchen!