Team Collaboration Guide

Guide for dividing work between team members on Express + Prisma project.

Team Structure

```
Team Member #1: Database Designer
    ↓ provides schema
Team Member #2: Queries Developer
    ↓ provides database functions
Team Member #3: Business Logic Developer (YOU)
    ↓ uses services in controllers
```

File Locations by Role

Database Designer

Works in: `prisma/schema.prisma`

Responsibilities: - Design database models - Define relationships - Create indexes - Manage migrations

Example:

```
model User {
  id        Int      @id @default(autoincrement())
  name      String
  email     String   @unique
  createdAt DateTime @default(now())
  posts     Post[]
}

model Post {
  id        Int      @id @default(autoincrement())
  title     String
  content   String
  userId    Int
  user      User     @relation(fields: [userId], references: [id])
}
```

Deliverables: - `prisma/schema.prisma` file - ERD diagram (optional) - Migration files (auto-generated)

Queries Developer

Works in: `src/services/*.service.js`

Responsibilities: - Write all database queries - CRUD operations - Complex queries (search, filter, aggregate) - Database transactions

File naming: `src/services/[model].service.js - user.service.js - post.service.js - comment.service.js`

Example:

```
// src/services/user.service.js
const prisma = require('../config/database');

class UserService {
  /**
   * Get all users
   */
  async getAll(filters = {}) {
    return await prisma.user.findMany({
      where: filters,
      include: {
        posts: true  // Include relations if needed
      }
    });
  }

  /**
   * Get user by ID
   */
  async getById(id) {
    return await prisma.user.findUnique({
      where: { id: parseInt(id) },
      include: {
        posts: true
      }
    });
  }

  /**
   * Create new user
   */
  async create(data) {
    return await prisma.user.create({
      data: {
        name: data.name,
        email: data.email
      }
    });
  }

  /**
```

```
   * Update user
   */
  async update(id, data) {
    return await prisma.user.update({
      where: { id: parseInt(id) },
      data
    });
  }

  /**
   * Delete user
   */
  async delete(id) {
    return await prisma.user.delete({
      where: { id: parseInt(id) }
    });
  }

  /**
   * Find user by email
   */
  async findByEmail(email) {
    return await prisma.user.findUnique({
      where: { email }
    });
  }

  /**
   * Search users by name
   */
  async searchByName(nameQuery) {
    return await prisma.user.findMany({
      where: {
        name: {
          contains: nameQuery,
          mode: 'insensitive'
        }
      }
    });
  }

  /**
   * Get user with posts count
   */
  async getUserStats(id) {
    return await prisma.user.findUnique({
      where: { id: parseInt(id) },
      include: {
```

```
        _count: {
          select: { posts: true }
        }
      }
    });
  }
}

module.exports = new UserService();
```

Deliverables: - Service files for each model - All CRUD operations - Custom query methods - JSDoc comments for each method

---

Business Logic Developer (YOU)

Works in: - `src/controllers/*.controller.js` (main work) - `src/routes/*.routes.js` (route definitions) - `src/middlewares/*.middleware.js` (validation, auth) - `src/app.js` (register routes)

Responsibilities: - Use services to handle requests - Implement business logic - Validate input - Handle errors - Format responses - Define API routes - Apply middlewares

Example:

```
// src/controllers/user.controller.js
const userService = require('../services/user.service');

class UserController {
  /**
   * Get all users
   * @route GET /api/users
   */
  async index(req, res, next) {
    try {
      const { search } = req.query;

      let users;
      if (search) {
        // Use service method
        users = await userService.searchByName(search);
      } else {
        // Use service method
        users = await userService.getAll();
      }

      res.json({
        success: true,
        count: users.length,
```

```
      data: users
    });
  } catch (error) {
    next(error);
  }
}

/**
 * Get single user
 * @route GET /api/users/:id
 */
async show(req, res, next) {
  try {
    // Use service method
    const user = await userService.getById(req.params.id);

    if (!user) {
      return res.status(404).json({
        success: false,
        error: 'User not found'
      });
    }

    res.json({
      success: true,
      data: user
    });
  } catch (error) {
    next(error);
  }
}

/**
 * Create user
 * @route POST /api/users
 */
async store(req, res, next) {
  try {
    const { name, email } = req.body;

    // BUSINESS LOGIC: Input validation
    if (!name || !email) {
      return res.status(400).json({
        success: false,
        error: 'Name and email are required'
      });
    }
```

```
    if (!email.includes('@')) {
      return res.status(400).json({
        success: false,
        error: 'Invalid email format'
      });
    }

    // BUSINESS LOGIC: Check duplicates
    const existing = await userService.findByEmail(email);
    if (existing) {
      return res.status(409).json({
        success: false,
        error: 'Email already exists'
      });
    }

    // Use service method to create
    const user = await userService.create({ name, email });

    res.status(201).json({
      success: true,
      message: 'User created successfully',
      data: user
    });
  } catch (error) {
    next(error);
  }
}

/**
 * Update user
 * @route PUT /api/users/:id
 */
async update(req, res, next) {
  try {
    const { id } = req.params;
    const { name, email } = req.body;

    // BUSINESS LOGIC: Check if user exists
    const existingUser = await userService.getById(id);
    if (!existingUser) {
      return res.status(404).json({
        success: false,
        error: 'User not found'
      });
    }

    // BUSINESS LOGIC: If email changing, check for duplicates
```

```
    if (email && email !== existingUser.email) {
      const duplicate = await userService.findByEmail(email);
      if (duplicate) {
        return res.status(409).json({
          success: false,
          error: 'Email already exists'
        });
      }
    }

    // Use service method to update
    const updatedUser = await userService.update(id, { name, email });

    res.json({
      success: true,
      message: 'User updated successfully',
      data: updatedUser
    });
  } catch (error) {
    next(error);
  }
}

/**
 * Delete user
 * @route DELETE /api/users/:id
 */
async destroy(req, res, next) {
  try {
    // BUSINESS LOGIC: Check if user exists
    const user = await userService.getById(req.params.id);
    if (!user) {
      return res.status(404).json({
        success: false,
        error: 'User not found'
      });
    }

    // Use service method to delete
    await userService.delete(req.params.id);

    res.json({
      success: true,
      message: 'User deleted successfully'
    });
  } catch (error) {
    next(error);
  }
```

```
  }
}

module.exports = new UserController();
```

Routes:

```
// src/routes/user.routes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/user.controller');

router.get('/', userController.index);
router.get('/:id', userController.show);
router.post('/', userController.store);
router.put('/:id', userController.update);
router.delete('/:id', userController.destroy);

module.exports = router;
```

Register in app.js:

```
// src/app.js
app.use('/api/users', require('./routes/user.routes'));
```

Deliverables: - Controller files - Route files - Middleware files (if needed) - Business logic implementation - Error handling - Response formatting

---

Workflow Example

Let's say you need to add "User" feature:

Step 1: Database Designer

Creates schema:

```
// prisma/schema.prisma
model User {
  id    Int    @id @default(autoincrement())
  name  String
  email String @unique
}
```

Runs:

```
npx prisma migrate dev --name add_user_model
```

Delivers: Updated `schema.prisma`

Step 2: Queries Developer

Creates service with all queries:

```
// src/services/user.service.js
class UserService {
  async getAll() { /* query */ }
  async getById(id) { /* query */ }
  async create(data) { /* query */ }
  async update(id, data) { /* query */ }
  async delete(id) { /* query */ }
  async findByEmail(email) { /* query */ }
}
```

Delivers: `src/services/user.service.js`

Step 3: YOU (Business Logic)

Use the service in controller:

```
// src/controllers/user.controller.js
const userService = require('../services/user.service');

class UserController {
  async store(req, res, next) {
    // Validation
    // Business rules
    // Use: await userService.create(data)
    // Format response
  }
}
```

Create routes:

```
// src/routes/user.routes.js
router.post('/', userController.store);
```

Register:

```
// src/app.js
app.use('/api/users', require('./routes/user.routes'));
```

Communication Template

When queries developer delivers a service file, they should document:

```
/**
 * User Service
 *
 * Available Methods:
```

```
 * - getAll(filters) - Get all users with optional filters
 * - getById(id) - Get single user by ID
 * - create(data) - Create new user
 * - update(id, data) - Update user
 * - delete(id) - Delete user
 * - findByEmail(email) - Find user by email
 * - searchByName(query) - Search users by name
 *
 * Example Usage:
 * const users = await userService.getAll();
 * const user = await userService.getById(1);
 */
```

Checklist for Integration

When you receive a service file from queries developer:

- ☐ Check if all needed methods are available
- ☐ Test each method works correctly
- ☐ Understand what each method returns
- ☐ Know what parameters each method needs
- ☐ Import service in your controller: `require('../services/xxx.service')`
- ☐ Use service methods in controller actions
- ☐ Add business logic around service calls
- ☐ Handle errors properly
- ☐ Format responses consistently

Summary

Role	Location	Responsibility
Database Designer	`prisma/schema.prisma`	Models, Relations, Migrations
Queries Developer	`src/services/`	ALL database queries
You (Business Logic)	`src/controllers/`, `src/routes/`	Use services, validation, logic, routing

Key Point: You import services and use them, you don't write queries yourself!

```
// You do this:
const userService = require('../services/user.service');
const user = await userService.getById(id);

// You DON'T do this:
const user = await prisma.user.findUnique({ where: { id } });
```