Complete Backend Development Cheatsheet

From Beginner to Advanced

Table of Contents

Backend Fundamentals

What is Backend Development?

Backend development involves server-side programming, database management, API creation, and server configuration. The backend handles business logic, data storage, authentication, and serves data to frontend applications.

Key Backend Concepts

- **Server**: Computer that provides services to other computers
- **API**: Application Programming Interface for communication
- **Database**: Storage system for application data
- **HTTP/HTTPS**: Protocol for web communication
- **REST**: Architectural style for web services
- **CRUD**: Create, Read, Update, Delete operations

HTTP Methods

```
GET     - Retrieve data
POST    - Create new data
PUT     - Update entire resource
PATCH   - Partial update
DELETE  - Remove data
HEAD    - Headers only
OPTIONS - Allowed methods
```

HTTP Status Codes

```
2xx Success
200 OK
201 Created
204 No Content

3xx Redirection
301 Moved Permanently
302 Found
304 Not Modified

4xx Client Error
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
422 Unprocessable Entity

5xx Server Error
500 Internal Server Error
502 Bad Gateway
503 Service Unavailable
```

JavaScript Essentials

ES6+ Features for Backend

```
// Arrow Functions
const getData = async () => {
  return await fetchData();
};

// Destructuring
const { name, email } = user;
const [first, second] = array;

// Template Literals
const message = `Hello ${name}, welcome!`;

// Spread Operator
const newArray = [...oldArray, newItem];
const newObject = { ...oldObject, newProperty: value };

// Promises and Async/Await
const fetchData = async () => {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
```

```
    return data;
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
};

// Classes
class User {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }

  async save() {
    // Save to database
  }
}

// Modules
// Export
export const helper = () => {};
export default MyClass;

// Import
import MyClass, { helper } from './module.js';
```

Error Handling

```
// Try-Catch
try {
  const result = riskyOperation();
} catch (error) {
  console.error('Error occurred:', error.message);
} finally {
  // Cleanup code
}

// Custom Error Classes
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}

// Error Middleware (Express)
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Something went wrong!' });
};
```

Node.js

Core Concepts

Node.js is a JavaScript runtime built on Chrome's V8 engine, allowing JavaScript to run on servers.

NPM (Node Package Manager)

```
# Initialize project
npm init -y

# Install packages
npm install express mongoose
npm install -D nodemon jest

# Scripts in package.json
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "jest"
  }
}
```

Core Modules

```
// File System
const fs = require('fs');
const fsPromises = require('fs').promises;

// Read file
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Write file
fs.writeFile('output.txt', 'Hello World', (err) => {
  if (err) throw err;
  console.log('File saved!');
});

// Path
const path = require('path');
const fullPath = path.join(__dirname, 'files', 'data.txt');
const extension = path.extname('file.txt');
```

```
// HTTP
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World');
});

// URL
const url = require('url');
const parsedUrl = url.parse(req.url, true);

// Crypto
const crypto = require('crypto');
const hash = crypto.createHash('sha256').update('password').digest('hex');

// Events
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('event', () => console.log('Event fired!'));
```

Environment Variables

```
// Using dotenv
require('dotenv').config();

const PORT = process.env.PORT || 3000;
const DB_URL = process.env.DATABASE_URL;

// .env file
PORT=3000
DATABASE_URL=mongodb://localhost:27017/myapp
JWT_SECRET=your-secret-key
```

Process Management

```
// Process events
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  process.exit(1);
});

process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection at:', promise, 'reason:', reason);
});

// Graceful shutdown
process.on('SIGTERM', () => {
  console.log('SIGTERM received, shutting down gracefully');
```

```
  server.close(() => {
    process.exit(0);
  });
});
```

Express.js

Basic Setup

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static('public'));

// Routes
app.get('/', (req, res) => {
  res.json({ message: 'Hello World!' });
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Routing

```
// Basic routes
app.get('/users', getAllUsers);
app.get('/users/:id', getUserById);
app.post('/users', createUser);
app.put('/users/:id', updateUser);
app.patch('/users/:id', partialUpdateUser);
app.delete('/users/:id', deleteUser);

// Route parameters
app.get('/users/:id/posts/:postId', (req, res) => {
  const { id, postId } = req.params;
  res.json({ userId: id, postId });
});

// Query parameters
app.get('/search', (req, res) => {
  const { q, page, limit } = req.query;
  res.json({ query: q, page, limit });
});
```

```
// Express Router
const router = express.Router();

router.get('/', (req, res) => {
  res.json({ message: 'Users route' });
});

router.post('/', (req, res) => {
  // Create user logic
});

app.use('/api/users', router);
```

Middleware

```
// Custom middleware
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.path} - ${new Date().toISOString()}`);
  next();
};

// Authentication middleware
const authenticate = (req, res, next) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'Access denied' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(400).json({ error: 'Invalid token' });
  }
};

// Validation middleware
const validateUser = (req, res, next) => {
  const { name, email } = req.body;

  if (!name || !email) {
    return res.status(400).json({ error: 'Name and email required' });
  }

  next();
};

// Error handling middleware
```

```
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);

  if (err.name === 'ValidationError') {
    return res.status(400).json({ error: err.message });
  }

  res.status(500).json({ error: 'Internal server error' });
};

// Apply middleware
app.use(logger);
app.use('/api/protected', authenticate);
app.use(errorHandler);
```

Request and Response

```
// Request object
app.post('/api/users', (req, res) => {
  console.log('Body:', req.body);
  console.log('Params:', req.params);
  console.log('Query:', req.query);
  console.log('Headers:', req.headers);
  console.log('Cookies:', req.cookies);
  console.log('Method:', req.method);
  console.log('URL:', req.url);
  console.log('IP:', req.ip);
});

// Response methods
app.get('/api/users', (req, res) => {
  // Send JSON
  res.json({ users: [] });

  // Send status with data
  res.status(201).json({ message: 'Created' });

  // Send file
  res.sendFile(path.join(__dirname, 'public', 'index.html'));

  // Set headers
  res.set('Custom-Header', 'value');

  // Redirect
  res.redirect('/login');

  // Send cookies
  res.cookie('token', 'abc123', { httpOnly: true });
});
```

Next.js API Routes

API Routes Structure

```
pages/
  api/
    users/
      index.js      // /api/users
      [id].js       // /api/users/:id
      create.js     // /api/users/create
    auth/
      login.js      // /api/auth/login
      register.js   // /api/auth/register
```

Basic API Route

```
// pages/api/users/index.js
export default function handler(req, res) {
  if (req.method === 'GET') {
    // Get all users
    res.status(200).json({ users: [] });
  } else if (req.method === 'POST') {
    // Create user
    const { name, email } = req.body;
    res.status(201).json({ message: 'User created', user: { name, email } });
  } else {
    res.setHeader('Allow', ['GET', 'POST']);
    res.status(405).end(`Method ${req.method} Not Allowed`);
  }
}
```

Dynamic API Routes

```
// pages/api/users/[id].js
export default function handler(req, res) {
  const { id } = req.query;

  switch (req.method) {
    case 'GET':
      res.status(200).json({ user: { id, name: 'John' } });
      break;
    case 'PUT':
      const { name, email } = req.body;
      res.status(200).json({ message: `User ${id} updated` });
      break;
    case 'DELETE':
      res.status(200).json({ message: `User ${id} deleted` });
```

```
      break;
    default:
      res.setHeader('Allow', ['GET', 'PUT', 'DELETE']);
      res.status(405).end(`Method ${req.method} Not Allowed`);
  }
}
```

Middleware in Next.js

```
// middleware.js
import { NextResponse } from 'next/server';

export function middleware(request) {
  // Check authentication
  const token = request.cookies.get('token');

  if (!token && request.nextUrl.pathname.startsWith('/api/protected')) {
    return NextResponse.json({ error: 'Unauthorized' }, { status: 401 });
  }

  return NextResponse.next();
}

export const config = {
  matcher: '/api/protected/:path*'
};
```

App Router API Routes (Next.js 13+)

```
// app/api/users/route.js
import { NextRequest, NextResponse } from 'next/server';

export async function GET(request) {
  return NextResponse.json({ users: [] });
}

export async function POST(request) {
  const body = await request.json();
  return NextResponse.json({ message: 'User created' }, { status: 201 });
}

// app/api/users/[id]/route.js
export async function GET(request, { params }) {
  const { id } = params;
  return NextResponse.json({ user: { id } });
}
```

Database Management

MongoDB with Mongoose

Setup and Connection

```
const mongoose = require('mongoose');

// Connection
const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected');
  } catch (error) {
    console.error('Database connection error:', error);
    process.exit(1);
  }
};

// Schema Definition
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Name is required'],
    trim: true,
    maxlength: 50
  },
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
    match: [/^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/, 'Invalid email']
  },
  password: {
    type: String,
    required: true,
    minlength: 6
  },
  role: {
    type: String,
    enum: ['user', 'admin'],
    default: 'user'
  },
  createdAt: {
    type: Date,
    default: Date.now
  },
  profile: {
```

```
    age: Number,
    bio: String
  }
}, {
  timestamps: true
});

// Middleware
userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next();
  this.password = await bcrypt.hash(this.password, 12);
  next();
});

// Methods
userSchema.methods.comparePassword = async function(candidatePassword) {
  return await bcrypt.compare(candidatePassword, this.password);
};

// Static methods
userSchema.statics.findByEmail = function(email) {
  return this.findOne({ email });
};

const User = mongoose.model('User', userSchema);
```

CRUD Operations

```
// Create
const createUser = async (userData) => {
  try {
    const user = new User(userData);
    await user.save();
    return user;
  } catch (error) {
    throw error;
  }
};

// Read
const getAllUsers = async () => {
  return await User.find().select('-password');
};

const getUserById = async (id) => {
  return await User.findById(id).select('-password');
};

// Update
const updateUser = async (id, updateData) => {
  return await User.findByIdAndUpdate(
```

```
    id,
    updateData,
    { new: true, runValidators: true }
  ).select('-password');
};

// Delete
const deleteUser = async (id) => {
  return await User.findByIdAndDelete(id);
};

// Complex queries
const searchUsers = async (query, page = 1, limit = 10) => {
  const skip = (page - 1) * limit;

  return await User.find({
    $or: [
      { name: { $regex: query, $options: 'i' } },
      { email: { $regex: query, $options: 'i' } }
    ]
  })
  .select('-password')
  .skip(skip)
  .limit(limit)
  .sort({ createdAt: -1 });
};
```

SQL with PostgreSQL (using pg)

```
const { Pool } = require('pg');

// Database connection
const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
});

// Create table
const createUsersTable = async () => {
  const query = `
    CREATE TABLE IF NOT EXISTS users (
      id SERIAL PRIMARY KEY,
      name VARCHAR(100) NOT NULL,
      email VARCHAR(100) UNIQUE NOT NULL,
      password VARCHAR(255) NOT NULL,
      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
  `;
```

```
  try {
    await pool.query(query);
    console.log('Users table created');
  } catch (error) {
    console.error('Error creating table:', error);
  }
};

// CRUD Operations
const userQueries = {
  // Create
  createUser: async (name, email, hashedPassword) => {
    const query = 'INSERT INTO users (name, email, password) VALUES ($1, $2, $3)
RETURNING id, name, email';
    const values = [name, email, hashedPassword];
    const result = await pool.query(query, values);
    return result.rows[0];
  },

  // Read
  getAllUsers: async () => {
    const query = 'SELECT id, name, email, created_at FROM users ORDER BY
created_at DESC';
    const result = await pool.query(query);
    return result.rows;
  },

  getUserById: async (id) => {
    const query = 'SELECT id, name, email, created_at FROM users WHERE id = $1';
    const result = await pool.query(query, [id]);
    return result.rows[0];
  },

  getUserByEmail: async (email) => {
    const query = 'SELECT * FROM users WHERE email = $1';
    const result = await pool.query(query, [email]);
    return result.rows[0];
  },

  // Update
  updateUser: async (id, name, email) => {
    const query = 'UPDATE users SET name = $1, email = $2 WHERE id = $3 RETURNING
id, name, email';
    const result = await pool.query(query, [name, email, id]);
    return result.rows[0];
  },

  // Delete
  deleteUser: async (id) => {
    const query = 'DELETE FROM users WHERE id = $1 RETURNING id';
    const result = await pool.query(query, [id]);
    return result.rows[0];
```

```
  }
};
```

Supabase

```
const { createClient } = require('@supabase/supabase-js');

const supabaseUrl = process.env.SUPABASE_URL;
const supabaseKey = process.env.SUPABASE_ANON_KEY;
const supabase = createClient(supabaseUrl, supabaseKey);

// CRUD Operations
const supabaseOperations = {
  // Create
  createUser: async (userData) => {
    const { data, error } = await supabase
      .from('users')
      .insert([userData])
      .select();

    if (error) throw error;
    return data[0];
  },

  // Read
  getAllUsers: async () => {
    const { data, error } = await supabase
      .from('users')
      .select('*')
      .order('created_at', { ascending: false });

    if (error) throw error;
    return data;
  },

  getUserById: async (id) => {
    const { data, error } = await supabase
      .from('users')
      .select('*')
      .eq('id', id)
      .single();

    if (error) throw error;
    return data;
  },

  // Update
  updateUser: async (id, updateData) => {
    const { data, error } = await supabase
      .from('users')
      .update(updateData)
```

```
      .eq('id', id)
      .select();

    if (error) throw error;
    return data[0];
  },

  // Delete
  deleteUser: async (id) => {
    const { data, error } = await supabase
      .from('users')
      .delete()
      .eq('id', id)
      .select();

    if (error) throw error;
    return data[0];
  },

  // Advanced queries
  searchUsers: async (searchTerm) => {
    const { data, error } = await supabase
      .from('users')
      .select('*')
      .or(`name.ilike.%${searchTerm}%,email.ilike.%${searchTerm}%`);

    if (error) throw error;
    return data;
  }
};

// Real-time subscriptions
const subscribeToUsers = () => {
  const subscription = supabase
    .channel('users')
    .on('postgres_changes',
      { event: '*', schema: 'public', table: 'users' },
      (payload) => {
        console.log('Change received!', payload);
      }
    )
    .subscribe();

  return subscription;
};
```

API Development

RESTful API Design

```
// Users Resource
const express = require('express');
const router = express.Router();

// GET /api/users - Get all users
router.get('/', async (req, res) => {
  try {
    const { page = 1, limit = 10, search } = req.query;
    const users = await User.find(search ? {
      $or: [
        { name: { $regex: search, $options: 'i' } },
        { email: { $regex: search, $options: 'i' } }
      ]
    } : {})
    .select('-password')
    .limit(limit * 1)
    .skip((page - 1) * limit)
    .sort({ createdAt: -1 });

    const total = await User.countDocuments();

    res.json({
      users,
      totalPages: Math.ceil(total / limit),
      currentPage: page,
      total
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// GET /api/users/:id - Get user by id
router.get('/:id', async (req, res) => {
  try {
    const user = await User.findById(req.params.id).select('-password');
    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }
    res.json(user);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// POST /api/users - Create new user
router.post('/', async (req, res) => {
  try {
    const { name, email, password } = req.body;

    // Validation
    if (!name || !email || !password) {
      return res.status(400).json({ error: 'All fields are required' });
```

```
    }

    // Check if user exists
    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(409).json({ error: 'User already exists' });
    }

    const user = new User({ name, email, password });
    await user.save();

    const userResponse = user.toObject();
    delete userResponse.password;

    res.status(201).json({
      message: 'User created successfully',
      user: userResponse
    });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// PUT /api/users/:id - Update user
router.put('/:id', async (req, res) => {
  try {
    const { name, email } = req.body;

    const user = await User.findByIdAndUpdate(
      req.params.id,
      { name, email },
      { new: true, runValidators: true }
    ).select('-password');

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

    res.json({
      message: 'User updated successfully',
      user
    });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// DELETE /api/users/:id - Delete user
router.delete('/:id', async (req, res) => {
  try {
    const user = await User.findByIdAndDelete(req.params.id);
    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }
```

```
    res.json({ message: 'User deleted successfully' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

API Validation

```
const { body, validationResult } = require('express-validator');

// Validation rules
const userValidationRules = () => {
  return [
    body('name')
      .isLength({ min: 2, max: 50 })
      .withMessage('Name must be between 2 and 50 characters')
      .trim()
      .escape(),
    body('email')
      .isEmail()
      .withMessage('Invalid email format')
      .normalizeEmail(),
    body('password')
      .isLength({ min: 6 })
      .withMessage('Password must be at least 6 characters')
      .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
      .withMessage('Password must contain uppercase, lowercase, and number')
  ];
};

// Validation middleware
const validate = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({
      error: 'Validation failed',
      details: errors.array()
    });
  }
  next();
};

// Usage
router.post('/users', userValidationRules(), validate, createUser);
```

API Documentation with Swagger

```
const swaggerJsdoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'My API',
      version: '1.0.0',
      description: 'A simple Express API'
    },
    servers: [
      {
        url: 'http://localhost:3000',
        description: 'Development server'
      }
    ]
  },
  apis: ['./routes/*.js']
};

const specs = swaggerJsdoc(options);
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(specs));

/**
 * @swagger
 * components:
 *   schemas:
 *     User:
 *       type: object
 *       required:
 *         - name
 *         - email
 *       properties:
 *         id:
 *           type: string
 *           description: Auto-generated user ID
 *         name:
 *           type: string
 *           description: User's name
 *         email:
 *           type: string
 *           description: User's email
 */

/**
 * @swagger
 * /api/users:
 *   get:
 *     summary: Get all users
 *     tags: [Users]
 *     parameters:
 *       - in: query
```

```
 *         name: page
 *         schema:
 *           type: integer
 *         description: Page number
 *     responses:
 *       200:
 *         description: List of users
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 users:
 *                   type: array
 *                   items:
 *                     $ref: '#/components/schemas/User'
 */
```

Rate Limiting

```
const rateLimit = require('express-rate-limit');

// Basic rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP'
});

// Strict rate limiting for auth endpoints
const authLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 5,
  message: 'Too many authentication attempts'
});

app.use('/api/', limiter);
app.use('/api/auth/', authLimiter);
```

Authentication & Security

JWT Authentication

```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');

// Generate JWT Token
const generateToken = (userId) => {
```

```
  return jwt.sign(
    { userId },
    process.env.JWT_SECRET,
    { expiresIn: '7d' }
  );
};

// Auth Routes
const authRoutes = {
  // Register
  register: async (req, res) => {
    try {
      const { name, email, password } = req.body;

      // Check if user exists
      const existingUser = await User.findOne({ email });
      if (existingUser) {
        return res.status(409).json({ error: 'User already exists' });
      }

      // Hash password
      const hashedPassword = await bcrypt.hash(password, 12);

      // Create user
      const user = new User({
        name,
        email,
        password: hashedPassword
      });

      await user.save();

      // Generate token
      const token = generateToken(user._id);

      res.status(201).json({
        message: 'User created successfully',
        token,
        user: {
          id: user._id,
          name: user.name,
          email: user.email
        }
      });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  },

  // Login
  login: async (req, res) => {
    try {
      const { email, password } = req.body;
```

```
      // Find user
      const user = await User.findOne({ email });
      if (!user) {
        return res.status(401).json({ error: 'Invalid credentials' });
      }

      // Check password
      const isValidPassword = await bcrypt.compare(password, user.password);
      if (!isValidPassword) {
        return res.status(401).json({ error: 'Invalid credentials' });
      }

      // Generate token
      const token = generateToken(user._id);

      res.json({
        message: 'Login successful',
        token,
        user: {
          id: user._id,
          name: user.name,
          email: user.email
        }
      });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  },

  // Get current user
  getMe: async (req, res) => {
    try {
      const user = await User.findById(req.user.userId).select('-password');
      res.json({ user });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  }
};

// Authentication Middleware
const authenticate = async (req, res, next) => {
  try {
    const token = req.header('Authorization')?.replace('Bearer ', '');

    if (!token) {
      return res.status(401).json({ error: 'Access denied, no token provided' });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const user = await User.findById(decoded.userId);

    if (!user) {
      return res.status(401).json({ error: 'Invalid token' });
```

```
    }

    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
  }
};

// Role-based Authorization
const authorize = (...roles) => {
  return async (req, res, next) => {
    try {
      const user = await User.findById(req.user.userId);

      if (!roles.includes(user.role)) {
        return res.status(403).json({ error: 'Access denied' });
      }

      next();
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  };
};

// Usage
app.post('/api/auth/register', authRoutes.register);
app.post('/api/auth/login', authRoutes.login);
app.get('/api/auth/me', authenticate, authRoutes.getMe);
app.get('/api/admin', authenticate, authorize('admin'), adminRoutes);
```

Password Reset

```
const crypto = require('crypto');
const nodemailer = require('nodemailer');

// Password Reset Schema addition
const userSchema = new mongoose.Schema({
  // ... existing fields
  resetPasswordToken: String,
  resetPasswordExpires: Date
});

// Email transporter
const transporter = nodemailer.createTransporter({
  service: 'gmail',
  auth: {
    user: process.env.EMAIL_USER,
    pass: process.env.EMAIL_PASS
  }
```

```
});

const passwordResetRoutes = {
  // Request password reset
  requestReset: async (req, res) => {
    try {
      const { email } = req.body;
      const user = await User.findOne({ email });

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

      // Generate reset token
      const resetToken = crypto.randomBytes(32).toString('hex');
      user.resetPasswordToken = resetToken;
      user.resetPasswordExpires = Date.now() + 3600000; // 1 hour

      await user.save();

      // Send email
      const resetUrl = `${process.env.FRONTEND_URL}/reset-password/${resetToken}`;

      await transporter.sendMail({
        to: user.email,
        subject: 'Password Reset Request',
        html: `
          <h2>Password Reset Request</h2>
          <p>Click the link below to reset your password:</p>
          <a href="${resetUrl}">Reset Password</a>
        `
      });

      res.json({ message: 'Password reset email sent' });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  },

  // Reset password
  resetPassword: async (req, res) => {
    try {
      const { token, newPassword } = req.body;

      const user = await User.findOne({
        resetPasswordToken: token,
        resetPasswordExpires: { $gt: Date.now() }
      });

      if (!user) {
        return res.status(400).json({ error: 'Invalid or expired token' });
      }

      // Update password
```

```
      user.password = await bcrypt.hash(newPassword, 12);
      user.resetPasswordToken = undefined;
      user.resetPasswordExpires = undefined;

      await user.save();

      res.json({ message: 'Password reset successful' });
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  }
};
```

Security Best Practices

```
const helmet = require('helmet');
const cors = require('cors');
const mongoSanitize = require('express-mongo-sanitize');
const xss = require('xss-clean');
const hpp = require('hpp');

// Security middleware
app.use(helmet()); // Set security headers
app.use(cors({
  origin: process.env.FRONTEND_URL,
  credentials: true
}));
app.use(mongoSanitize()); // Prevent NoSQL injection
app.use(xss()); // Clean user input from malicious HTML
app.use(hpp()); // Prevent HTTP Parameter Pollution

// Input sanitization
const sanitizeInput = (input) => {
  if (typeof input === 'string') {
    return input.trim().replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*
<\/script>/gi, '');
  }
  return input;
};

// HTTPS redirect (production)
const forceHTTPS = (req, res, next) => {
  if (!req.secure && req.get('x-forwarded-proto') !== 'https') {
    return res.redirect(`https://${req.get('host')}${req.url}`);
  }
  next();
};

// Rate limiting by user
const createAccountLimiter = rateLimit({
  windowMs: 60 * 60 * 1000, // 1 hour
```

```
  max: 5, // 5 accounts per hour
  message: 'Too many accounts created from this IP'
});
```

Testing

Unit Testing with Jest

```
// __tests__/user.test.js
const User = require('../models/User');
const { createUser, getUserById } = require('../controllers/userController');

// Mock database
jest.mock('../models/User');

describe('User Controller', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  describe('createUser', () => {
    it('should create a new user successfully', async () => {
      const userData = {
        name: 'John Doe',
        email: 'john@example.com',
        password: 'password123'
      };

      const mockUser = { id: '123', ...userData };
      User.prototype.save = jest.fn().mockResolvedValue(mockUser);

      const req = { body: userData };
      const res = {
        status: jest.fn().mockReturnThis(),
        json: jest.fn()
      };

      await createUser(req, res);

      expect(res.status).toHaveBeenCalledWith(201);
      expect(res.json).toHaveBeenCalledWith({
        message: 'User created successfully',
        user: expect.objectContaining({
          name: userData.name,
          email: userData.email
        })
      });
    });

    it('should return error for invalid data', async () => {
```

```
      const req = { body: {} };
      const res = {
        status: jest.fn().mockReturnThis(),
        json: jest.fn()
      };

      await createUser(req, res);

      expect(res.status).toHaveBeenCalledWith(400);
      expect(res.json).toHaveBeenCalledWith({
        error: 'All fields are required'
      });
    });
  });
});

// Integration Testing
const request = require('supertest');
const app = require('../app');

describe('User API', () => {
  it('GET /api/users should return all users', async () => {
    const response = await request(app)
      .get('/api/users')
      .expect(200);

    expect(response.body).toHaveProperty('users');
    expect(Array.isArray(response.body.users)).toBe(true);
  });

  it('POST /api/users should create new user', async () => {
    const userData = {
      name: 'Test User',
      email: 'test@example.com',
      password: 'password123'
    };

    const response = await request(app)
      .post('/api/users')
      .send(userData)
      .expect(201);

    expect(response.body.user).toHaveProperty('name', userData.name);
    expect(response.body.user).toHaveProperty('email', userData.email);
    expect(response.body.user).not.toHaveProperty('password');
  });
});
```

API Testing with Postman/Newman

```
// package.json scripts
{
  "scripts": {
    "test:api": "newman run postman_collection.json -e postman_environment.json"
  }
}

// Test helper functions
const testHelpers = {
  // Create test user
  createTestUser: async () => {
    const userData = {
      name: 'Test User',
      email: 'test@example.com',
      password: 'password123'
    };

    const response = await request(app)
      .post('/api/users')
      .send(userData);

    return response.body.user;
  },

  // Get auth token
  getAuthToken: async () => {
    const loginData = {
      email: 'test@example.com',
      password: 'password123'
    };

    const response = await request(app)
      .post('/api/auth/login')
      .send(loginData);

    return response.body.token;
  },

  // Clean up test data
  cleanup: async () => {
    await User.deleteMany({ email: /test/ });
  }
};
```

Deployment & DevOps

Environment Configuration

```
// config/config.js
const config = {
  development: {
    port: process.env.PORT || 3000,
    db: {
      host: process.env.DB_HOST || 'localhost',
      port: process.env.DB_PORT || 27017,
      name: process.env.DB_NAME || 'myapp_dev'
    },
    jwt: {
      secret: process.env.JWT_SECRET || 'dev-secret',
      expiresIn: '7d'
    },
    email: {
      host: process.env.EMAIL_HOST,
      port: process.env.EMAIL_PORT,
      user: process.env.EMAIL_USER,
      pass: process.env.EMAIL_PASS
    }
  },
  production: {
    port: process.env.PORT,
    db: {
      uri: process.env.DATABASE_URL
    },
    jwt: {
      secret: process.env.JWT_SECRET,
      expiresIn: '1d'
    },
    redis: {
      url: process.env.REDIS_URL
    }
  }
};

module.exports = config[process.env.NODE_ENV || 'development'];
```

Docker Configuration

```
# Dockerfile
FROM node:18-alpine

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production
```

```
# Copy source code
COPY . .

# Create non-root user
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001

# Change ownership
RUN chown -R nextjs:nodejs /app
USER nextjs

EXPOSE 3000

CMD ["npm", "start"]
```

```
# docker-compose.yml
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=mongodb://mongo:27017/myapp
      - REDIS_URL=redis://redis:6379
    depends_on:
      - mongo
      - redis
    volumes:
      - ./logs:/app/logs

  mongo:
    image: mongo:5
    volumes:
      - mongo_data:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data

volumes:
  mongo_data:
  redis_data:
```

Process Management with PM2

```
// ecosystem.config.js
module.exports = {
  apps: [{
    name: 'myapp',
    script: 'server.js',
    instances: 'max',
    exec_mode: 'cluster',
    env: {
      NODE_ENV: 'development'
    },
    env_production: {
      NODE_ENV: 'production',
      PORT: 3000
    },
    error_file: './logs/err.log',
    out_file: './logs/out.log',
    log_file: './logs/combined.log',
    time: true
  }]
};

// package.json scripts
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "pm2:start": "pm2 start ecosystem.config.js --env production",
    "pm2:reload": "pm2 reload ecosystem.config.js --env production",
    "pm2:stop": "pm2 stop ecosystem.config.js",
    "pm2:delete": "pm2 delete ecosystem.config.js"
  }
}
```

Nginx Configuration

```
# /etc/nginx/sites-available/myapp
server {
    listen 80;
    server_name yourdomain.com;

    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Health Checks and Monitoring

```
// Health check endpoint
app.get('/health', async (req, res) => {
  try {
    // Check database connection
    await mongoose.connection.db.admin().ping();

    // Check Redis connection (if using)
    // await redis.ping();

    res.status(200).json({
      status: 'OK',
      timestamp: new Date().toISOString(),
      uptime: process.uptime(),
      memory: process.memoryUsage(),
      database: 'connected'
    });
  } catch (error) {
    res.status(503).json({
      status: 'ERROR',
      error: error.message
    });
  }
});

// Graceful shutdown
const gracefulShutdown = (signal) => {
  console.log(`Received ${signal}. Graceful shutdown...`);

  server.close(() => {
    console.log('HTTP server closed.');

    mongoose.connection.close(false, () => {
      console.log('MongoDB connection closed.');
      process.exit(0);
    });
  });
};

process.on('SIGTERM', gracefulShutdown);
process.on('SIGINT', gracefulShutdown);
```

Advanced Concepts

Caching with Redis

```
const redis = require('redis');
const client = redis.createClient(process.env.REDIS_URL);

// Cache middleware
const cache = (duration = 300) => {
  return async (req, res, next) => {
    const key = `cache:${req.originalUrl}`;

    try {
      const cached = await client.get(key);
      if (cached) {
        return res.json(JSON.parse(cached));
      }

      // Store original res.json
      const originalJson = res.json;

      res.json = function(data) {
        // Cache the response
        client.setex(key, duration, JSON.stringify(data));

        // Call original json method
        originalJson.call(this, data);
      };

      next();
    } catch (error) {
      next();
    }
  };
};

// Usage
app.get('/api/users', cache(600), getAllUsers);

// Cache invalidation
const invalidateCache = (pattern) => {
  return async (req, res, next) => {
    try {
      const keys = await client.keys(pattern);
      if (keys.length > 0) {
        await client.del(keys);
      }
    } catch (error) {
      console.error('Cache invalidation error:', error);
    }
    next();
  };
};
```

```
// Invalidate user cache after operations
app.post('/api/users', invalidateCache('cache:/api/users*'), createUser);
```

File Upload Handling

```
const multer = require('multer');
const path = require('path');
const fs = require('fs').promises;

// Storage configuration
const storage = multer.diskStorage({
  destination: async (req, file, cb) => {
    const uploadPath = 'uploads/';
    try {
      await fs.mkdir(uploadPath, { recursive: true });
      cb(null, uploadPath);
    } catch (error) {
      cb(error);
    }
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix +
path.extname(file.originalname));
  }
});

// File filter
const fileFilter = (req, file, cb) => {
  const allowedTypes = ['image/jpeg', 'image/png', 'image/gif'];

  if (allowedTypes.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(new Error('Invalid file type'), false);
  }
};

const upload = multer({
  storage,
  fileFilter,
  limits: {
    fileSize: 5 * 1024 * 1024 // 5MB
  }
});

// Upload routes
app.post('/api/upload/single', upload.single('image'), (req, res) => {
  if (!req.file) {
    return res.status(400).json({ error: 'No file uploaded' });
  }
```

```
  res.json({
    message: 'File uploaded successfully',
    file: {
      filename: req.file.filename,
      originalname: req.file.originalname,
      mimetype: req.file.mimetype,
      size: req.file.size
    }
  });
});

app.post('/api/upload/multiple', upload.array('images', 5), (req, res) => {
  if (!req.files || req.files.length === 0) {
    return res.status(400).json({ error: 'No files uploaded' });
  }

  const files = req.files.map(file => ({
    filename: file.filename,
    originalname: file.originalname,
    mimetype: file.mimetype,
    size: file.size
  }));

  res.json({
    message: 'Files uploaded successfully',
    files
  });
});
```

WebSocket Implementation

```
const http = require('http');
const socketIo = require('socket.io');

const server = http.createServer(app);
const io = socketIo(server, {
  cors: {
    origin: process.env.FRONTEND_URL,
    methods: ["GET", "POST"]
  }
});

// Socket authentication middleware
io.use(async (socket, next) => {
  try {
    const token = socket.handshake.auth.token;
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const user = await User.findById(decoded.userId);

    socket.user = user;
```

```
    next();
  } catch (error) {
    next(new Error('Authentication error'));
  }
});

// Socket connection handling
io.on('connection', (socket) => {
  console.log(`User ${socket.user.name} connected`);

  // Join user to their room
  socket.join(`user_${socket.user._id}`);

  // Handle chat messages
  socket.on('send_message', async (data) => {
    try {
      const message = new Message({
        sender: socket.user._id,
        content: data.content,
        room: data.room
      });

      await message.save();

      // Broadcast to room
      socket.to(data.room).emit('new_message', {
        id: message._id,
        content: message.content,
        sender: {
          id: socket.user._id,
          name: socket.user.name
        },
        timestamp: message.createdAt
      });
    } catch (error) {
      socket.emit('error', { message: error.message });
    }
  });

  // Handle disconnect
  socket.on('disconnect', () => {
    console.log(`User ${socket.user.name} disconnected`);
  });
});

server.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Background Jobs with Bull Queue

```
const Queue = require('bull');
const nodemailer = require('nodemailer');

// Create queues
const emailQueue = new Queue('email processing', process.env.REDIS_URL);
const imageQueue = new Queue('image processing', process.env.REDIS_URL);

// Email job processor
emailQueue.process(async (job) => {
  const { to, subject, html } = job.data;

  const transporter = nodemailer.createTransporter({
    // email config
  });

  await transporter.sendMail({
    from: process.env.FROM_EMAIL,
    to,
    subject,
    html
  });

  return { sent: true };
});

// Image processing job
imageQueue.process(async (job) => {
  const { imagePath, sizes } = job.data;

  // Process image with sharp or similar
  const sharp = require('sharp');

  for (const size of sizes) {
    await sharp(imagePath)
      .resize(size.width, size.height)
      .toFile(`processed/${size.name}_${path.basename(imagePath)}`);
  }

  return { processed: true };
});

// Add jobs to queue
const sendWelcomeEmail = async (userEmail, userName) => {
  await emailQueue.add('welcome email', {
    to: userEmail,
    subject: 'Welcome!',
    html: `<h1>Welcome ${userName}!</h1>`
  });
};

const processUserImage = async (imagePath) => {
  await imageQueue.add('resize image', {
    imagePath,
```

```
      sizes: [
        { name: 'thumbnail', width: 150, height: 150 },
        { name: 'medium', width: 500, height: 500 }
      ]
    });
};

// Queue monitoring
emailQueue.on('completed', (job) => {
  console.log(`Email job ${job.id} completed`);
});

emailQueue.on('failed', (job, err) => {
  console.error(`Email job ${job.id} failed:`, err);
});
```

Database Migrations

```
// migrations/001_create_users_table.js
const mongoose = require('mongoose');

const up = async () => {
  // Create indexes
  await mongoose.connection.collection('users').createIndex({ email: 1 }, {
unique: true });
  await mongoose.connection.collection('users').createIndex({ createdAt: -1 });
};

const down = async () => {
  // Drop indexes
  await mongoose.connection.collection('users').dropIndex({ email: 1 });
  await mongoose.connection.collection('users').dropIndex({ createdAt: -1 });
};

module.exports = { up, down };

// Migration runner
const runMigrations = async () => {
  const migrations = [
    require('./migrations/001_create_users_table'),
    // Add more migrations
  ];

  for (const migration of migrations) {
    try {
      await migration.up();
      console.log(`Migration ${migration.name} completed`);
    } catch (error) {
      console.error(`Migration ${migration.name} failed:`, error);
      break;
    }
```

```
  }
};
```

Logging

```
const winston = require('winston');

// Configure logger
const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  defaultMeta: { service: 'api' },
  transports: [
    new winston.transports.File({ filename: 'logs/error.log', level: 'error' }),
    new winston.transports.File({ filename: 'logs/combined.log' }),
  ],
});

if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.simple()
  }));
}

// Request logging middleware
const requestLogger = (req, res, next) => {
  const start = Date.now();

  res.on('finish', () => {
    const duration = Date.now() - start;
    logger.info({
      method: req.method,
      url: req.url,
      status: res.statusCode,
      duration: `${duration}ms`,
      userAgent: req.get('User-Agent'),
      ip: req.ip
    });
  });

  next();
};

app.use(requestLogger);

// Usage in routes
app.get('/api/users', async (req, res) => {
```

```
  try {
    logger.info('Fetching users');
    const users = await User.find();
    res.json(users);
  } catch (error) {
    logger.error('Error fetching users', { error: error.message, stack:
error.stack });
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Quick Reference Commands

NPM Commands

```
# Initialize project
npm init -y

# Install dependencies
npm install express mongoose dotenv
npm install -D nodemon jest

# Run scripts
npm start
npm run dev
npm test

# Update packages
npm update
npm audit fix
```

Git Commands

```
# Initialize repository
git init
git add .
git commit -m "Initial commit"

# Create and switch branches
git checkout -b feature/api-endpoints
git push -u origin feature/api-endpoints

# Merge changes
git checkout main
git merge feature/api-endpoints
```

Docker Commands

```
# Build image
docker build -t myapp .

# Run container
docker run -p 3000:3000 myapp

# Docker compose
docker-compose up -d
docker-compose down
docker-compose logs -f
```

Database Commands

```
# MongoDB
mongosh
use myapp
db.users.find()
db.users.createIndex({ email: 1 }, { unique: true })

# PostgreSQL
psql -U username -d database
\dt
SELECT * FROM users;
CREATE INDEX idx_users_email ON users(email);
```

This comprehensive cheatsheet covers all essential backend development concepts from beginner to advanced levels. Use it as a reference guide for building robust, scalable backend applications with modern technologies and best practices.