Complete API Development Cheatsheet

Table of Contents

REST API Fundamentals

HTTP Methods

```
GET    - Retrieve data
POST   - Create new resource
PUT    - Update entire resource
PATCH  - Partial update
DELETE - Remove resource
```

HTTP Status Codes

```
2xx Success
200 OK - Request successful
201 Created - Resource created
204 No Content - Success with no body

4xx Client Errors
400 Bad Request - Invalid request
401 Unauthorized - Authentication required
403 Forbidden - Access denied
404 Not Found - Resource not found
422 Unprocessable Entity - Validation error

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

RESTful URL Structure

```
GET    /api/users           - Get all users
GET    /api/users/:id       - Get specific user
POST   /api/users           - Create user
PUT    /api/users/:id       - Update user
DELETE /api/users/:id       - Delete user

GET    /api/users/:id/posts - Get user's posts
POST   /api/users/:id/posts - Create post for user
```

Node.js & Express.js

Basic Express Server Setup

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

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

// Middleware
app.use(helmet()); // Security headers
app.use(cors()); // Enable CORS
app.use(express.json({ limit: '10mb' })); // Parse JSON
app.use(express.urlencoded({ extended: true })); // Parse URL-encoded

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // limit each IP to 100 requests per windowMs
});
app.use(limiter);

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

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

Express Router & Controllers

```javascript
// routes/users.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

router.get('/', userController.getAllUsers);
router.get('/:id', userController.getUserById);
router.post('/', userController.createUser);
router.put('/:id', userController.updateUser);
router.delete('/:id', userController.deleteUser);

module.exports = router;

// controllers/userController.js
const User = require('../models/User');

exports.getAllUsers = async (req, res) => {
  try {
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;

    const users = await User.find()
      .select('-password')
      .skip(skip)
      .limit(limit)
      .sort({ createdAt: -1 });

    const total = await User.countDocuments();

    res.json({
      success: true,
      data: users,
      pagination: {
        page,
        limit,
        total,
        pages: Math.ceil(total / limit)
      }
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message
    });
  }
};

exports.createUser = async (req, res) => {
  try {
    const { name, email, password } = req.body;

    // Validation
    if (!name || !email || !password) {
      return res.status(400).json({
        success: false,
        message: 'Name, email, and password are required'
      });
    }

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

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

    res.status(201).json({
      success: true,
      data: {
        id: user._id,
        name: user.name,
        email: user.email
      }
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message
    });
  }
};
```

Middleware Examples

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

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

    if (!token) {
      return res.status(401).json({ message: 'No token provided' });
    }
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ message: 'Invalid token' });
  }
};

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

  if (!name || name.length < 2) {
    return res.status(400).json({
      message: 'Name must be at least 2 characters long'
    });
  }

  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!emailRegex.test(email)) {
    return res.status(400).json({
      message: 'Invalid email format'
    });
  }

  next();
};

// Error handling middleware
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);

  if (err.name === 'ValidationError') {
    return res.status(400).json({
      success: false,
      message: err.message,
      errors: err.errors
    });
  }

  if (err.name === 'CastError') {
    return res.status(400).json({
      success: false,
      message: 'Invalid ID format'
    });
  }

  res.status(500).json({
    success: false,
    message: 'Internal server error'
  });
};
```

Database Integration

MongoDB with Mongoose

```
// Database connection
const mongoose = require('mongoose');

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);
  }
};

// User Schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Name is required'],
    trim: true,
    minlength: [2, 'Name must be at least 2 characters']
  },
  email: {
    type: String,
    required: [true, 'Email is required'],
    unique: true,
    lowercase: true,
    match: [/^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/, 'Invalid email']
  },
  password: {
    type: String,
    required: [true, 'Password is required'],
    minlength: [6, 'Password must be at least 6 characters']
  },
  role: {
    type: String,
    enum: ['user', 'admin'],
    default: 'user'
  },
  isActive: {
    type: Boolean,
    default: true
  },
  avatar: String,
  posts: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Post'
  }]
}, {
  timestamps: true
});

// Middleware for password hashing
userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next();

  const bcrypt = require('bcryptjs');
  this.password = await bcrypt.hash(this.password, 12);
  next();
});

// Instance methods
userSchema.methods.comparePassword = async function(candidatePassword) {
  const bcrypt = require('bcryptjs');
  return bcrypt.compare(candidatePassword, this.password);
};

userSchema.methods.toJSON = function() {
  const user = this.toObject();
  delete user.password;
  return user;
};

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

PostgreSQL with Prisma

```
// Install: npm install prisma @prisma/client

// schema.prisma
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

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

  @@map("users")
}

model Post {
  id        Int      @id @default(autoincrement())
  title     String
  content   String?
  published Boolean  @default(false)
  author    User     @relation(fields: [authorId], references: [id])
  authorId  Int
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  @@map("posts")
}

// Database service
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

class UserService {
  static async createUser(data) {
    return await prisma.user.create({
      data,
      select: {
        id: true,
        email: true,
        name: true,
        createdAt: true
      }
    });
  }

  static async getUserById(id) {
    return await prisma.user.findUnique({
      where: { id: parseInt(id) },
      include: {
        posts: {
          where: { published: true },
          select: {
            id: true,
            title: true,
            createdAt: true
          }
        }
      }
    });
  }

  static async updateUser(id, data) {
    return await prisma.user.update({
      where: { id: parseInt(id) },
      data
    });
  }

  static async deleteUser(id) {
    return await prisma.user.delete({
      where: { id: parseInt(id) }
    });
  }
}
```

Supabase Integration

```
// Install: npm install @supabase/supabase-js

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

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

class SupabaseService {
  static async getUsers() {
    const { data, error } = await supabase
      .from('users')
      .select('*')
      .order('created_at', { ascending: false });

    if (error) throw error;
    return data;
  }

  static async createUser(user) {
    const { data, error } = await supabase
      .from('users')
      .insert([user])
      .select();

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

  static async updateUser(id, updates) {
    const { data, error } = await supabase
      .from('users')
      .update(updates)
      .eq('id', id)
      .select();

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

  // Real-time subscriptions
  static subscribeToUsers(callback) {
    return supabase
      .channel('users')
      .on('postgres_changes', {
        event: '*',
        schema: 'public',
        table: 'users'
      }, callback)
      .subscribe();
  }

  // File upload
  static async uploadFile(bucket, fileName, file) {
    const { data, error } = await supabase.storage
      .from(bucket)
      .upload(fileName, file);

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

Authentication & Authorization

JWT Implementation

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

// Generate JWT token
const generateToken = (payload) => {
  return jwt.sign(payload, process.env.JWT_SECRET, {
    expiresIn: process.env.JWT_EXPIRES_IN || '7d'
  });
};

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

    // Find user
    const user = await User.findOne({ email }).select('+password');
    if (!user) {
      return res.status(401).json({
        success: false,
        message: 'Invalid credentials'
      });
    }

    // Check password
    const isPasswordValid = await user.comparePassword(password);
    if (!isPasswordValid) {
      return res.status(401).json({
        success: false,
        message: 'Invalid credentials'
      });
    }

    // Generate token
    const token = generateToken({
      id: user._id,
      email: user.email,
      role: user.role
    });

    res.json({
      success: true,
      token,
      user: {
        id: user._id,
        name: user.name,
        email: user.email,
        role: user.role
      }
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message
    });
  }
};

// Role-based authorization
const authorize = (...roles) => {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        message: 'Authentication required'
      });
    }

    if (!roles.includes(req.user.role)) {
      return res.status(403).json({
        message: 'Insufficient permissions'
      });
    }

    next();
  };
};

// Usage
app.get('/api/admin/users', authenticate, authorize('admin'), getAllUsers);
```

OAuth with Passport.js

```
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;

passport.use(new GoogleStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  callbackURL: "/auth/google/callback"
}, async (accessToken, refreshToken, profile, done) => {
  try {
    let user = await User.findOne({ googleId: profile.id });

    if (user) {
      return done(null, user);
    }

    user = await User.create({
      googleId: profile.id,
      name: profile.displayName,
      email: profile.emails[0].value,
      avatar: profile.photos[0].value
    });

    done(null, user);
  } catch (error) {
    done(error, null);
  }
}));

// Routes
app.get('/auth/google',
  passport.authenticate('google', { scope: ['profile', 'email'] })
);

app.get('/auth/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  (req, res) => {
    const token = generateToken({ id: req.user._id });
    res.redirect(`${process.env.CLIENT_URL}/auth/success?token=${token}`);
  }
);
```

GraphQL

Apollo Server Setup

```
// Install: npm install apollo-server-express graphql

const { ApolloServer, gql } = require('apollo-server-express');

// Type definitions
const typeDefs = gql`
  type User {
    id: ID!
    name: String!
    email: String!
    posts: [Post!]!
    createdAt: String!
  }

  type Post {
    id: ID!
    title: String!
    content: String
    published: Boolean!
    author: User!
    createdAt: String!
  }

  type Query {
    users: [User!]!
    user(id: ID!): User
    posts: [Post!]!
    post(id: ID!): Post
  }

  type Mutation {
    createUser(input: CreateUserInput!): User!
    updateUser(id: ID!, input: UpdateUserInput!): User!
    deleteUser(id: ID!): Boolean!
    createPost(input: CreatePostInput!): Post!
    updatePost(id: ID!, input: UpdatePostInput!): Post!
    deletePost(id: ID!): Boolean!
  }

  input CreateUserInput {
    name: String!
    email: String!
    password: String!
  }

  input UpdateUserInput {
    name: String
    email: String
  }

  input CreatePostInput {
    title: String!
    content: String
    published: Boolean = false
    authorId: ID!
  }

  input UpdatePostInput {
    title: String
    content: String
    published: Boolean
  }
`;

// Resolvers
const resolvers = {
  Query: {
    users: async () => {
      return await User.find().populate('posts');
    },
    user: async (_, { id }) => {
      return await User.findById(id).populate('posts');
    },
    posts: async () => {
      return await Post.find().populate('author');
    },
    post: async (_, { id }) => {
      return await Post.findById(id).populate('author');
    }
  },

  Mutation: {
    createUser: async (_, { input }) => {
      const user = new User(input);
      await user.save();
      return user;
    },
    updateUser: async (_, { id, input }) => {
      return await User.findByIdAndUpdate(id, input, { new: true });
    },
    deleteUser: async (_, { id }) => {
      await User.findByIdAndDelete(id);
      return true;
    },
    createPost: async (_, { input }) => {
      const post = new Post(input);
      await post.save();

      // Add post to user's posts array
      await User.findByIdAndUpdate(
        input.authorId,
        { $push: { posts: post._id } }
      );

      return await Post.findById(post._id).populate('author');
    }
  },

  // Field resolvers
  User: {
    posts: async (parent) => {
      return await Post.find({ author: parent._id });
    }
  },

  Post: {
    author: async (parent) => {
      return await User.findById(parent.author);
    }
  }
};

// Create Apollo Server
const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    // Add authentication context
    const token = req.headers.authorization || '';
    let user = null;

    if (token) {
      try {
        user = jwt.verify(token.replace('Bearer ', ''), process.env.JWT_SECRET);
      } catch (err) {
        console.log('Invalid token');
      }
    }

    return { user };
  }
});

// Apply Apollo GraphQL middleware
server.applyMiddleware({ app, path: '/graphql' });
```

GraphQL Subscriptions

```
// Install: npm install graphql-subscriptions

const { PubSub } = require('graphql-subscriptions');
const pubsub = new PubSub();
```

```
// Add to typeDefs
const typeDefs = gql`
  type Subscription {
    postAdded: Post!
    userUpdated(userId: ID!): User!
  }
`;

// Add to resolvers
const resolvers = {
  Subscription: {
    postAdded: {
      subscribe: () => pubsub.asyncIterator(['POST_ADDED'])
    },
    userUpdated: {
      subscribe: (_, { userId }) =>
pubsub.asyncIterator([`USER_UPDATED_${userId}`])
    }
  },

  Mutation: {
    createPost: async (_, { input }) => {
      const post = new Post(input);
      await post.save();

      const populatedPost = await Post.findById(post._id).populate('author');

      // Publish subscription
      pubsub.publish('POST_ADDED', { postAdded: populatedPost });

      return populatedPost;
    }
  }
};
```

Next.js API Routes

Basic API Routes

```
// pages/api/users/index.js
export default async function handler(req, res) {
  const { method } = req;

  switch (method) {
    case 'GET':
      try {
        // Get all users with pagination
        const page = parseInt(req.query.page) || 1;
        const limit = parseInt(req.query.limit) || 10;
        const skip = (page - 1) * limit;
        const users = await User.find()
          .select('-password')
          .skip(skip)
          .limit(limit)
          .sort({ createdAt: -1 });

        const total = await User.countDocuments();

        res.status(200).json({
          success: true,
          data: users,
          pagination: {
            page,
            limit,
            total,
            pages: Math.ceil(total / limit)
          }
        });
      } catch (error) {
        res.status(500).json({
          success: false,
          message: error.message
        });
      }
      break;

    case 'POST':
      try {
        const { name, email, password } = req.body;

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

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

        res.status(201).json({
          success: true,
          data: {
            id: user._id,
            name: user.name,
            email: user.email
          }
        });
      } catch (error) {
        res.status(500).json({
          success: false,
          message: error.message
        });
      }
      break;

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

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

  switch (method) {
    case 'GET':
      try {
        const user = await User.findById(id).select('-password');
        if (!user) {
          return res.status(404).json({
            success: false,
            message: 'User not found'
          });
        }
        res.status(200).json({ success: true, data: user });
      } catch (error) {
        res.status(500).json({ success: false, message: error.message });
      }
      break;

    case 'PUT':
      try {
        const user = await User.findByIdAndUpdate(
          id,
          req.body,
          { new: true, runValidators: true }
        ).select('-password');

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

        res.status(200).json({ success: true, data: user });
      } catch (error) {
        res.status(500).json({ success: false, message: error.message });
      }
      break;

    case 'DELETE':
      try {
        const user = await User.findByIdAndDelete(id);
        if (!user) {
          return res.status(404).json({
            success: false,
            message: 'User not found'
          });
        }
        res.status(200).json({ success: true, message: 'User deleted' });
      } catch (error) {
        res.status(500).json({ success: false, message: error.message });
      }
      break;

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

Next.js Middleware

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

export function middleware(request) {
  // Check if it's an API route that needs authentication
  if (request.nextUrl.pathname.startsWith('/api/protected')) {
    const token = request.headers.get('authorization')?.replace('Bearer ', '');

    if (!token) {
      return new NextResponse(
        JSON.stringify({ success: false, message: 'Authentication required' }),
        { status: 401, headers: { 'content-type': 'application/json' } }
      );
    }

    try {
      verify(token, process.env.JWT_SECRET);
    } catch (error) {
      return new NextResponse(
        JSON.stringify({ success: false, message: 'Invalid token' }),
        { status: 401, headers: { 'content-type': 'application/json' } }
      );
    }
  }

  return NextResponse.next();
}

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

Frontend Integration

JavaScript Fetch API

```
class ApiClient {
  constructor(baseURL = 'http://localhost:3000/api') {
    this.baseURL = baseURL;
    this.token = localStorage.getItem('token');
  }

  async request(endpoint, options = {}) {
    const url = `${this.baseURL}${endpoint}`;

    const config = {
      headers: {
        'Content-Type': 'application/json',
        ...options.headers,
      },
      ...options,
    };

    if (this.token) {
      config.headers.Authorization = `Bearer ${this.token}`;
    }

    if (config.body && typeof config.body === 'object') {
      config.body = JSON.stringify(config.body);
    }

    try {
      const response = await fetch(url, config);
      const data = await response.json();

      if (!response.ok) {
        throw new Error(data.message || 'Request failed');
      }

      return data;
    } catch (error) {
      console.error('API Request failed:', error);
      throw error;
    }
  }

  // User methods
  async getUsers(page = 1, limit = 10) {
    return this.request(`/users?page=${page}&limit=${limit}`);
  }

  async getUserById(id) {
    return this.request(`/users/${id}`);
  }

  async createUser(userData) {
    return this.request('/users', {
      method: 'POST',
      body: userData,
    });
  }

  async updateUser(id, userData) {
    return this.request(`/users/${id}`, {
      method: 'PUT',
      body: userData,
    });
  }

  async deleteUser(id) {
    return this.request(`/users/${id}`, {
      method: 'DELETE',
    });
  }

  // Authentication
  async login(email, password) {
    const response = await this.request('/auth/login', {
      method: 'POST',
      body: { email, password },
    });

    if (response.token) {
      this.token = response.token;
      localStorage.setItem('token', response.token);
    }

    return response;
  }

  logout() {
    this.token = null;
    localStorage.removeItem('token');
  }
}

// Usage
const api = new ApiClient();

// Get users
api.getUsers(1, 10)
  .then(response => {
    console.log('Users:', response.data);
```

```
      console.log('Pagination:', response.pagination);
    })
    .catch(error => {
      console.error('Error:', error.message);
    });

  // Create user
  api.createUser({
    name: 'John Doe',
    email: 'john@example.com',
    password: 'password123'
  })
  .then(response => {
    console.log('User created:', response.data);
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

React Hooks for API

```
  // hooks/useApi.js
  import { useState, useEffect } from 'react';

  export const useApi = (url, options = {}) => {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
      const fetchData = async () => {
        try {
          setLoading(true);
          const response = await fetch(url, options);
          const result = await response.json();

          if (!response.ok) {
            throw new Error(result.message || 'Request failed');
          }

          setData(result);
        } catch (err) {
          setError(err.message);
        } finally {
          setLoading(false);
        }
      };

      fetchData();
    }, [url]);
    return { data, loading, error };
  };

  // hooks/useUsers.js
  export const useUsers = (page = 1, limit = 10) => {
    const [users, setUsers] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState(null);
    const [pagination, setPagination] = useState(null);

    const fetchUsers = async () => {
      try {
        setLoading(true);
        const response = await api.getUsers(page, limit);
        setUsers(response.data);
        setPagination(response.pagination);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    useEffect(() => {
      fetchUsers();
    }, [page, limit]);

    const createUser = async (userData) => {
      try {
        setLoading(true);
        const response = await api.createUser(userData);
        setUsers(prev => [response.data, ...prev]);
        return response;
      } catch (err) {
        setError(err.message);
        throw err;
      } finally {
        setLoading(false);
      }
    };

    const updateUser = async (id, userData) => {
      try {
        setLoading(true);
        const response = await api.updateUser(id, userData);
        setUsers(prev => prev.map(user =>
          user.id === id ? response.data : user
        ));
        return response;
      } catch (err) {
        setError(err.message);
        throw err;
      } finally {
        setLoading(false);
      }
    };

    const deleteUser = async (id) => {
      try {
        setLoading(true);
        await api.deleteUser(id);
        setUsers(prev => prev.filter(user => user.id !== id));
      } catch (err) {
        setError(err.message);
        throw err;
      } finally {
        setLoading(false);
      }
    };

    return {
      users,
      loading,
      error,
      pagination,
      createUser,
      updateUser,
      deleteUser,
      refetch: fetchUsers
    };
  };

  // Component usage
  import React, { useState } from 'react';
  import { useUsers } from '../hooks/useUsers';

  const UsersList = () => {
    const [page, setPage] = useState(1);
    const { users, loading, error, pagination, createUser, deleteUser } =
  useUsers(page);
    const [newUser, setNewUser] = useState({ name: '', email: '', password: '' });

    const handleSubmit = async (e) => {
      e.preventDefault();
      try {
        await createUser(newUser);
        setNewUser({ name: '', email: '', password: '' });
        alert('User created successfully!');
      } catch (error) {
        alert('Error creating user: ' + error.message);
      }
    };

    const handleDelete = async (id) => {
      if (window.confirm('Are you sure?')) {
        try {
          await deleteUser(id);
          alert('User deleted successfully!');
        } catch (error) {
          alert('Error deleting user: ' + error.message);
        }
      }
    };

    if (loading) return <div>Loading...</div>;
    if (error) return <div>Error: {error}</div>;

    return (
      <div>
        <h1>Users</h1>

        {/* Create User Form */}
        <form onSubmit={handleSubmit}>
          <input
            type="text"
            placeholder="Name"
            value={newUser.name}
            onChange={(e) => setNewUser({...newUser, name: e.target.value})}
            required
          />
          <input
            type="email"
            placeholder="Email"
            value={newUser.email}
            onChange={(e) => setNewUser({...newUser, email: e.target.value})}
            required
          />
          <input
            type="password"
            placeholder="Password"
            value={newUser.password}
            onChange={(e) => setNewUser({...newUser, password: e.target.value})}
            required
          />
          <button type="submit">Create User</button>
        </form>

        {/* Users List */}
        <div>
          {users.map(user => (
            <div key={user.id} style={{ border: '1px solid #ccc', padding: '10px',
margin: '10px' }}>
              <h3>{user.name}</h3>
              <p>{user.email}</p>
              <button onClick={() => handleDelete(user.id)}>Delete</button>
            </div>
          ))}
        </div>

        {/* Pagination */}
        {pagination && (
          <div>
            <button
              onClick={() => setPage(prev => Math.max(1, prev - 1))}
              disabled={page === 1}
            >
              Previous
            </button>
            <span> Page {page} of {pagination.pages} </span>
            <button
              onClick={() => setPage(prev => Math.min(pagination.pages, prev + 1))}
              disabled={page === pagination.pages}
            >
              Next
            </button>
          </div>
        )}
      </div>
    );
  };
```

Axios Alternative

```
// Install: npm install axios
import axios from 'axios';

const api = axios.create({
  baseURL: 'http://localhost:3000/api',
  timeout: 10000,
});

// Request interceptor
api.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('token');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Response interceptor
api.interceptors.response.use(
  (response) => response.data,
  (error) => {
    if (error.response?.status === 401) {
      localStorage.removeItem('token');
      window.location.href = '/login';
    }
```

```
      return Promise.reject(error.response?.data || error.message);
    }
  );

  // Usage
  const UserService = {
    getAll: (params) => api.get('/users', { params }),
    getById: (id) => api.get(`/users/${id}`),
    create: (data) => api.post('/users', data),
    update: (id, data) => api.put(`/users/${id}`, data),
    delete: (id) => api.delete(`/users/${id}`),
  };
```

Testing

Unit Testing with Jest

```
// Install: npm install --save-dev jest supertest

// tests/user.test.js
const request = require('supertest');
const app = require('../app');
const User = require('../models/User');

describe('User API', () => {
  beforeEach(async () => {
    await User.deleteMany({});
  });

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

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

      expect(response.body.success).toBe(true);
      expect(response.body.data.name).toBe(userData.name);
      expect(response.body.data.email).toBe(userData.email);
      expect(response.body.data.password).toBeUndefined();
    });

    it('should return 400 if required fields are missing', async () => {
      const response = await request(app)
```

```
        .post('/api/users')
        .send({ name: 'John Doe' })
        .expect(400);

      expect(response.body.success).toBe(false);
      expect(response.body.message).toContain('required');
    });

    it('should return 409 if user already exists', async () => {
      const userData = {
        name: 'John Doe',
        email: 'john@example.com',
        password: 'password123'
      };

      await User.create(userData);

      const response = await request(app)
        .post('/api/users')
        .send(userData)
        .expect(409);

      expect(response.body.success).toBe(false);
      expect(response.body.message).toContain('already exists');
    });
  });

  describe('GET /api/users', () => {
    it('should return all users with pagination', async () => {
      await User.create([
        { name: 'User 1', email: 'user1@example.com', password: 'password' },
        { name: 'User 2', email: 'user2@example.com', password: 'password' },
      ]);

      const response = await request(app)
        .get('/api/users?page=1&limit=10')
        .expect(200);

      expect(response.body.success).toBe(true);
      expect(response.body.data).toHaveLength(2);
      expect(response.body.pagination).toBeDefined();
    });
  });

  describe('GET /api/users/:id', () => {
    it('should return a specific user', async () => {
      const user = await User.create({
        name: 'John Doe',
        email: 'john@example.com',
        password: 'password123'
      });

      const response = await request(app)
        .get(`/api/users/${user._id}`)
```

```
          .expect(200);

        expect(response.body.success).toBe(true);
        expect(response.body.data.name).toBe(user.name);
      });

      it('should return 404 if user not found', async () => {
        const response = await request(app)
          .get('/api/users/507f1f77bcf86cd799439011')
          .expect(404);

        expect(response.body.success).toBe(false);
      });
    });
  });

  // Mock database for testing
  const mockUser = {
    _id: '507f1f77bcf86cd799439011',
    name: 'John Doe',
    email: 'john@example.com',
    save: jest.fn(),
    comparePassword: jest.fn()
  };

  jest.mock('../models/User', () => ({
    find: jest.fn(),
    findOne: jest.fn(),
    create: jest.fn(),
    findById: jest.fn(),
    findByIdAndUpdate: jest.fn(),
    findByIdAndDelete: jest.fn(),
    countDocuments: jest.fn()
  }));
```

Integration Testing

```
// tests/integration/auth.test.js
const request = require('supertest');
const app = require('../../app');

describe('Authentication Flow', () => {
  let userToken;
  let userId;

  it('should register a new user', async () => {
    const response = await request(app)
      .post('/api/auth/register')
      .send({
        name: 'Test User',
        email: 'test@example.com',
        password: 'password123'
      })
      .expect(201);

    expect(response.body.success).toBe(true);
    expect(response.body.token).toBeDefined();
    userToken = response.body.token;
    userId = response.body.user.id;
  });

  it('should login with valid credentials', async () => {
    const response = await request(app)
      .post('/api/auth/login')
      .send({
        email: 'test@example.com',
        password: 'password123'
      })
      .expect(200);

    expect(response.body.success).toBe(true);
    expect(response.body.token).toBeDefined();
  });

  it('should access protected route with valid token', async () => {
    const response = await request(app)
      .get('/api/profile')
      .set('Authorization', `Bearer ${userToken}`)
      .expect(200);

    expect(response.body.success).toBe(true);
    expect(response.body.user.id).toBe(userId);
  });

  it('should deny access to protected route without token', async () => {
    await request(app)
      .get('/api/profile')
      .expect(401);
  });
});
```

Deployment & Production

Environment Configuration

```
// .env.example
NODE_ENV=production
PORT=3000
DATABASE_URL=mongodb://localhost:27017/myapp
JWT_SECRET=your-super-secret-jwt-key
JWT_EXPIRES_IN=7d
```

```
BCRYPT_ROUNDS=12

# External Services
REDIS_URL=redis://localhost:6379
EMAIL_SERVICE=sendgrid
EMAIL_API_KEY=your-sendgrid-api-key

# OAuth
GOOGLE_CLIENT_ID=your-google-client-id
GOOGLE_CLIENT_SECRET=your-google-client-secret

# File Upload
AWS_ACCESS_KEY_ID=your-aws-access-key
AWS_SECRET_ACCESS_KEY=your-aws-secret-key
AWS_S3_BUCKET=your-bucket-name

# config/database.js
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const options = {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      maxPoolSize: 10,
      serverSelectionTimeoutMS: 5000,
      socketTimeoutMS: 45000,
    };

    if (process.env.NODE_ENV === 'production') {
      options.ssl = true;
      options.sslValidate = true;
    }

    await mongoose.connect(process.env.DATABASE_URL, options);
    console.log('MongoDB connected successfully');
  } catch (error) {
    console.error('Database connection error:', error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Production Server Setup

```
// server.js
const express = require('express');
const helmet = require('helmet');
const compression = require('compression');
const rateLimit = require('express-rate-limit');
const mongoSanitize = require('express-mongo-sanitize');
const xss = require('xss-clean');
const hpp = require('hpp');
const cors = require('cors');

const app = express();

// Trust proxy
app.set('trust proxy', 1);

// Security middleware
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'"],
      scriptSrc: ["'self'"],
      imgSrc: ["'self'", "data:", "https:"],
    },
  },
}));

// Compression
app.use(compression());

// CORS
const corsOptions = {
  origin: process.env.NODE_ENV === 'production'
    ? ['https://yourdomain.com', 'https://www.yourdomain.com']
    : ['http://localhost:3000', 'http://localhost:3001'],
  credentials: true,
  optionsSuccessStatus: 200
};
app.use(cors(corsOptions));

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: process.env.NODE_ENV === 'production' ? 100 : 1000,
  message: {
    error: 'Too many requests from this IP, please try again later.'
  },
  standardHeaders: true,
  legacyHeaders: false,
});
app.use('/api', limiter);

// Body parsing
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));

// Data sanitization
app.use(mongoSanitize()); // Against NoSQL injection
app.use(xss()); // Against XSS attacks
app.use(hpp()); // Prevent parameter pollution

// Health check
app.get('/health', (req, res) => {
  res.status(200).json({
    status: 'OK',
    timestamp: new Date().toISOString(),
    uptime: process.uptime()
  });
});

// Routes
app.use('/api/auth', require('./routes/auth'));
app.use('/api/users', require('./routes/users'));
app.use('/api/posts', require('./routes/posts'));

// 404 handler
app.use('*', (req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found'
  });
});

// Global error handler
app.use((err, req, res, next) => {
  console.error(err.stack);

  let error = { ...err };
  error.message = err.message;

  // Mongoose bad ObjectId
  if (err.name === 'CastError') {
    const message = 'Resource not found';
    error = { message, statusCode: 404 };
  }

  // Mongoose duplicate key
  if (err.code === 11000) {
    const message = 'Duplicate field value entered';
    error = { message, statusCode: 400 };
  }

  // Mongoose validation error
  if (err.name === 'ValidationError') {
    const message = Object.values(err.errors).map(val => val.message);
    error = { message: message.join(', '), statusCode: 400 };
  }

  res.status(error.statusCode || 500).json({
    success: false,
    message: error.message || 'Server Error'
  });
});
```

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running in ${process.env.NODE_ENV} mode on port ${PORT}`);
});

// Graceful shutdown
process.on('SIGTERM', () => {
  console.log('SIGTERM received. Shutting down gracefully...');
  server.close(() => {
    console.log('Process terminated');
  });
});
```

Docker Configuration

```
# Dockerfile
FROM node:18-alpine

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production && npm cache clean --force

# Copy source code
COPY . .

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

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

EXPOSE 3000

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1

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

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

```yaml
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
    restart: unless-stopped

  mongo:
    image: mongo:5.0
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    restart: unless-stopped

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
      - ./ssl:/etc/nginx/ssl
    depends_on:
      - app
    restart: unless-stopped

volumes:
  mongo_data:
  redis_data:
```

Best Practices

API Design Principles

```javascript
// 1. Consistent Response Format
const sendResponse = (res, statusCode, success, message, data = null) => {
  res.status(statusCode).json({
    success,
    message,
    data,
    timestamp: new Date().toISOString()
  });
};

// 2. Error Handling Class
class ApiError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = true;
    Error.captureStackTrace(this, this.constructor);
  }
}

// 3. Async Error Handler
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// 4. Input Validation with Joi
const Joi = require('joi');

const validateUser = (req, res, next) => {
  const schema = Joi.object({
    name: Joi.string().min(2).max(50).required(),
    email: Joi.string().email().required(),
    password: Joi.string().min(6).required(),
    age: Joi.number().integer().min(18).max(120),
    role: Joi.string().valid('user', 'admin').default('user')
  });

  const { error, value } = schema.validate(req.body);

  if (error) {
    return res.status(400).json({
      success: false,
      message: 'Validation error',
      errors: error.details.map(detail => ({
        field: detail.path[0],
        message: detail.message
      }))
    });
  }
```

```
    req.body = value;
    next();
  };

  // 5. Request Logging
  const requestLogger = (req, res, next) => {
    const start = Date.now();

    res.on('finish', () => {
      const duration = Date.now() - start;
      console.log(`${req.method} ${req.originalUrl} ${res.statusCode} -
  ${duration}ms`);
    });

    next();
  };

  // 6. API Versioning
  // Option 1: URL versioning
  app.use('/api/v1', require('./routes/v1'));
  app.use('/api/v2', require('./routes/v2'));

  // Option 2: Header versioning
  const versionMiddleware = (req, res, next) => {
    const version = req.headers['api-version'] || 'v1';
    req.apiVersion = version;
    next();
  };

  // 7. Caching with Redis
  const redis = require('redis');
  const client = redis.createClient(process.env.REDIS_URL);

  const cache = (duration = 300) => {
    return async (req, res, next) => {
      const key = req.originalUrl;

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

        // Override res.json to cache the response
        const originalJson = res.json;
        res.json = function(data) {
          client.setex(key, duration, JSON.stringify(data));
          originalJson.call(this, data);
        };

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

  // Usage: app.get('/api/users', cache(600), getUsersController);
```

Performance Optimization

```
// 1. Database Query Optimization
// Bad
const getUsersWithPosts = async () => {
  const users = await User.find();
  for (let user of users) {
    user.posts = await Post.find({ author: user._id });
  }
  return users;
};

// Good
const getUsersWithPosts = async () => {
  return await User.find().populate({
    path: 'posts',
    select: 'title createdAt',
    options: { sort: { createdAt: -1 }, limit: 5 }
  });
};

// 2. Pagination and Filtering
const getPaginatedUsers = async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = Math.min(parseInt(req.query.limit) || 10, 100); // Max 100
  const skip = (page - 1) * limit;

  // Build filter object
  const filter = {};
  if (req.query.role) filter.role = req.query.role;
  if (req.query.isActive) filter.isActive = req.query.isActive === 'true';
  if (req.query.search) {
    filter.$or = [
      { name: { $regex: req.query.search, $options: 'i' } },
      { email: { $regex: req.query.search, $options: 'i' } }
    ];
  }

  // Execute queries in parallel
  const [users, total] = await Promise.all([
    User.find(filter)
      .select('-password')
      .skip(skip)
      .limit(limit)
      .sort({ createdAt: -1 }),
    User.countDocuments(filter)
  ]);

  res.json({
    success: true,
    data: users,
    pagination: {
      page,
      limit,
      total,
      pages: Math.ceil(total / limit),
      hasNext: page < Math.ceil(total / limit),
      hasPrev: page > 1
    }
  });
};

// 3. File Upload with Multer
const multer = require('multer');
const path = require('path');

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix +
path.extname(file.originalname));
  }
});

const upload = multer({
  storage,
  limits: {
    fileSize: 5 * 1024 * 1024, // 5MB
  },
  fileFilter: (req, file, cb) => {
    const allowedTypes = /jpeg|jpg|png|gif/;
    const extname =
allowedTypes.test(path.extname(file.originalname).toLowerCase());
    const mimetype = allowedTypes.test(file.mimetype);

    if (mimetype && extname) {
      return cb(null, true);
    } else {
      cb(new Error('Only image files are allowed'));
    }
  }
});

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

  res.json({
    success: true,
    filename: req.file.filename,
    url: `/uploads/${req.file.filename}`
  });
});
```

Security Best Practices

```
// 1. Input Sanitization
const sanitizeInput = (req, res, next) => {
  if (req.body) {
    for (let key in req.body) {
      if (typeof req.body[key] === 'string') {
        req.body[key] = req.body[key].trim();
      }
    }
  }
  next();
};

// 2. SQL Injection Prevention (for SQL databases)
const db = require('./database');

// Bad
const getUserByEmail = async (email) => {
  const query = `SELECT * FROM users WHERE email = '${email}'`;
  return db.execute(query);
};

// Good
const getUserByEmail = async (email) => {
  const query = 'SELECT * FROM users WHERE email = ?';
  return db.execute(query, [email]);
};

// 3. Password Security
const bcrypt = require('bcryptjs');
const crypto = require('crypto');

const hashPassword = async (password) => {
  const salt = await bcrypt.genSalt(12);
  return bcrypt.hash(password, salt);
};

const generateSecureToken = () => {
  return crypto.randomBytes(32).toString('hex');
};
```

```
// 4. API Key Management
const validateApiKey = (req, res, next) => {
  const apiKey = req.headers['x-api-key'];

  if (!apiKey) {
    return res.status(401).json({ message: 'API key required' });
  }

  // In production, store hashed API keys in database
  const validApiKeys = process.env.VALID_API_KEYS?.split(',') || [];

  if (!validApiKeys.includes(apiKey)) {
    return res.status(401).json({ message: 'Invalid API key' });
  }

  next();
};

// 5. HTTPS Enforcement
const enforceHTTPS = (req, res, next) => {
  if (process.env.NODE_ENV === 'production' && !req.secure && req.get('X-
Forwarded-Proto') !== 'https') {
    return res.redirect(301, `https://${req.get('Host')}${req.url}`);
  }
  next();
};
```

Monitoring and Logging

```
// 1. Winston Logger
const winston = require('winston');

const logger = winston.createLogger({
  level: process.env.LOG_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()
  }));
}
```

```javascript
// 2. Request ID Middleware
const { v4: uuidv4 } = require('uuid');

const requestId = (req, res, next) => {
  req.id = uuidv4();
  res.setHeader('X-Request-ID', req.id);
  next();
};

// 3. Performance Monitoring
const performanceMonitor = (req, res, next) => {
  const start = process.hrtime.bigint();

  res.on('finish', () => {
    const end = process.hrtime.bigint();
    const duration = Number(end - start) / 1000000; // Convert to milliseconds

    logger.info('Request completed', {
      requestId: req.id,
      method: req.method,
      url: req.originalUrl,
      statusCode: res.statusCode,
      duration: `${duration.toFixed(2)}ms`,
      userAgent: req.get('User-Agent'),
      ip: req.ip
    });

    // Alert on slow requests
    if (duration > 1000) {
      logger.warn('Slow request detected', {
        requestId: req.id,
        duration: `${duration.toFixed(2)}ms`,
        url: req.originalUrl
      });
    }
  });

  next();
};
```

This comprehensive cheatsheet covers all major aspects of API development from basic concepts to advanced production practices. Each section includes practical code examples and best practices that you can immediately implement in your projects. The guide progresses from fundamental REST API concepts through specific technologies like Node.js, Express, databases, and frontend integration, concluding with production-ready deployment and monitoring strategies.