Complete JavaScript, TypeScript, Express.js & Project Building Guide

Table of Contents

JavaScript Fundamentals

Variables and Data Types

```javascript
// Variable declarations
let name = "John";          // String
const age = 25;             // Number
var isActive = true;        // Boolean
let data = null;            // Null
let undefined_var;          // Undefined

// Objects and Arrays
const person = {
    name: "Alice",
    age: 30,
    hobbies: ["reading", "coding"]
};

const numbers = [1, 2, 3, 4, 5];
```

Functions

javascript

```javascript
// Function declaration
function greet(name) {
    return `Hello, ${name}!`;
}

// Arrow functions
const add = (a, b) => a + b;

// Higher-order functions
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(n => n * 2);
const evens = numbers.filter(n => n % 2 === 0);
```

Asynchronous JavaScript

javascript

```javascript
// Promises
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data fetched");
        }, 1000);
    });
}

// Async/Await
async function getData() {
    try {
        const result = await fetchData();
        console.log(result);
    } catch (error) {
        console.error(error);
    }
}
```

ES6+ Features

javascript

```javascript
// Destructuring
const { name, age } = person;
const [first, second] = numbers;

// Spread operator
const newNumbers = [...numbers, 6, 7];
const newPerson = { ...person, city: "New York" };

// Template literals
const message = `Hello ${name}, you are ${age} years old`;

// Classes
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a sound`);
    }
}
```

TypeScript Complete Guide

Basic Types

typescript

```typescript
// Basic types
let username: string = "john";
let age: number = 25;
let isActive: boolean = true;
let data: null = null;
let value: undefined = undefined;

// Arrays
let numbers: number[] = [1, 2, 3];
let names: Array<string> = ["Alice", "Bob"];

// Tuples
let coordinates: [number, number] = [10, 20];

// Enums
enum Status {
  PENDING = "pending",
  APPROVED = "approved",
  REJECTED = "rejected"
}
```

Interfaces and Types

typescript

```typescript
// Interface
interface User {
  id: number;
  name: string;
  email: string;
  age?: number; // Optional property
  readonly createdAt: Date; // Readonly
}

// Type alias
type UserRole = "admin" | "user" | "moderator";

// Generic interfaces
interface ApiResponse<T> {
  data: T;
  status: number;
  message: string;
}

// Function interfaces
interface MathOperation {
  (a: number, b: number): number;
}

const add: MathOperation = (a, b) => a + b;
```

Classes and Inheritance

typescript

```
abstract class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    abstract makeSound(): void;

    move(): void {
        console.log(`${this.name} is moving`);
    }
}

class Dog extends Animal {
    private breed: string;

    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }

    makeSound(): void {
        console.log(`${this.name} barks`);
    }

    getBreed(): string {
        return this.breed;
    }
}
```

Generics

typescript

```typescript
// Generic function
function identity<T>(arg: T): T {
    return arg;
}

// Generic class
class Stack<T> {
    private items: T[] = [];

    push(item: T): void {
        this.items.push(item);
    }

    pop(): T | undefined {
        return this.items.pop();
    }
}

// Generic constraints
interface Lengthwise {
    length: number;
}

function logLength<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);
    return arg;
}
```

TypeScript Configuration (tsconfig.json)

```json
```

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "lib": ["ES2022"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Express.js Framework Guide

Installation and Setup

```bash
npm init -y
npm install express
npm install -D @types/express typescript ts-node nodemon
```

Basic Express Server

```typescript
```

```typescript
import express, { Request, Response, NextFunction } from 'express';

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

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

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

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

// POST route
app.post('/users', (req: Request, res: Response) => {
  const userData = req.body;
  res.status(201).json({ user: userData });
});

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

Middleware

typescript

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

// Error handling middleware
const errorHandler = (err: Error, req: Request, res: Response, next: NextFunction) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Something went wrong!' });
};

app.use(logger);
app.use('/api', apiRoutes);
app.use(errorHandler);
```

Router

typescript

```
import { Router } from 'express';

const userRouter = Router();

userRouter.get('/', (req, res) => {
  res.json({ users: [] });
});

userRouter.post('/', (req, res) => {
  res.json({ message: 'User created' });
});

userRouter.put('/:id', (req, res) => {
  res.json({ message: 'User updated' });
});

userRouter.delete('/:id', (req, res) => {
  res.json({ message: 'User deleted' });
});

export default userRouter;
```

Building a Complete Project

Project Structure

```
my-app/
├── src/
│   ├── controllers/
│   ├── middleware/
│   ├── models/
│   ├── routes/
│   ├── services/
│   ├── utils/
│   ├── types/
│   └── app.ts
├── dist/
├── tests/
├── package.json
├── tsconfig.json
├── .env
├── .gitignore
└── README.md
```

Environment Configuration

typescript

```typescript
// src/config/environment.ts
import dotenv from 'dotenv';

dotenv.config();

export const config = {
  port: process.env.PORT || 3000,
  nodeEnv: process.env.NODE_ENV || 'development',
  databaseUrl: process.env.DATABASE_URL || '',
  jwtSecret: process.env.JWT_SECRET || 'secret',
  jwtExpiresIn: process.env.JWT_EXPIRES_IN || '24h',
};
```

Database Connection

typescript

```typescript
// src/config/database.ts
import { Pool } from 'pg';
import { config } from './environment';

export const pool = new Pool({
  connectionString: config.databaseUrl,
  ssl: config.nodeEnv === 'production' ? { rejectUnauthorized: false } : false,
});

export const connectDB = async () => {
  try {
    await pool.connect();
    console.log('Database connected successfully');
  } catch (error) {
    console.error('Database connection failed:', error);
    process.exit(1);
  }
};
```

REST API Configuration

RESTful API Principles

typescript

```typescript
// CRUD operations following REST conventions
interface ApiRoutes {
  'GET /api/users': 'Get all users';
  'GET /api/users/:id': 'Get user by ID';
  'POST /api/users': 'Create new user';
  'PUT /api/users/:id': 'Update user';
  'DELETE /api/users/:id': 'Delete user';
}

// Status codes
const HTTP_STATUS = {
  OK: 200,
  CREATED: 201,
  NO_CONTENT: 204,
  BAD_REQUEST: 400,
  UNAUTHORIZED: 401,
  FORBIDDEN: 403,
  NOT_FOUND: 404,
  CONFLICT: 409,
  INTERNAL_SERVER_ERROR: 500,
} as const;
```

API Response Format

typescript

```typescript
interface ApiResponse<T = any> {
  success: boolean;
  data?: T;
  message?: string;
  error?: string;
  pagination?: {
    page: number;
    limit: number;
    total: number;
    totalPages: number;
  };
}

// Response helper
class ResponseHelper {
  static success<T>(data: T, message?: string): ApiResponse<T> {
    return {
      success: true,
      data,
      message,
    };
  }

  static error(error: string, message?: string): ApiResponse {
    return {
      success: false,
      error,
      message,
    };
  }
}
```

Validation Middleware

typescript

```
import Joi from 'joi';

const validateSchema = (schema: Joi.ObjectSchema) => {
    return (req: Request, res: Response, next: NextFunction) => {
        const { error } = schema.validate(req.body);
        if (error) {
            return res.status(400).json(
                ResponseHelper.error(error.details[0].message)
            );
        }
        next();
    };
};

// User validation schema
const userSchema = Joi.object({
    name: Joi.string().min(2).max(50).required(),
    email: Joi.string().email().required(),
    password: Joi.string().min(6).required(),
});
```

HTTP Server from Scratch

Basic HTTP Server (No Framework)

javascript

```
const http = require('http');
const url = require('url');
const querystring = require('querystring');

class HTTPServer {
    constructor() {
        this.routes = {
            GET: {},
            POST: {},
            PUT: {},
            DELETE: {}
        };
        this.middlewares = [];
    }

    // Add middleware
    use(middleware) {
        this.middlewares.push(middleware);
    }

    // Route handlers
    get(path, handler) {
        this.routes.GET[path] = handler;
    }

    post(path, handler) {
        this.routes.POST[path] = handler;
    }

    put(path, handler) {
        this.routes.PUT[path] = handler;
    }

    delete(path, handler) {
        this.routes.DELETE[path] = handler;
    }

    // Parse request body
    async parseBody(req) {
        return new Promise((resolve) => {
            let body = '';
            req.on('data', chunk => {
                body += chunk.toString();
            });
            req.on('end', () => {
                try {
```

```
                resolve(JSON.parse(body));
            } catch {
                resolve(body);
            }
        });
    });
}

// Handle requests
async handleRequest(req, res) {
    const parsedUrl = url.parse(req.url, true);
    const path = parsedUrl.pathname;
    const method = req.method;

    // Set CORS headers
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS');
    res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');

    if (method === 'OPTIONS') {
        res.writeHead(200);
        res.end();
        return;
    }

    // Create request/response objects
    const request = {
        method,
        url: req.url,
        path,
        query: parsedUrl.query,
        headers: req.headers,
        body: method !== 'GET' ? await this.parseBody(req) : null
    };

    const response = {
        statusCode: 200,
        headers: {},
        setHeader: (key, value) => {
            response.headers[key] = value;
        },
        json: (data) => {
            response.setHeader('Content-Type', 'application/json');
            res.writeHead(response.statusCode, response.headers);
            res.end(JSON.stringify(data));
        },
        send: (data) => {
```

```
            res.writeHead(response.statusCode, response.headers);
            res.end(data);
        },
        status: (code) => {
            response.statusCode = code;
            return response;
        }
    };

    // Execute middlewares
    for (const middleware of this.middlewares) {
        await middleware(request, response);
    }

    // Find and execute route handler
    const handler = this.routes[method] && this.routes[method][path];

    if (handler) {
        try {
            await handler(request, response);
        } catch (error) {
            response.status(500).json({ error: 'Internal server error' });
        }
    } else {
        response.status(404).json({ error: 'Route not found' });
    }
  }

  // Start server
  listen(port, callback) {
    const server = http.createServer((req, res) => {
        this.handleRequest(req, res);
    });

    server.listen(port, callback);
    return server;
  }
}

// Usage example
const app = new HTTPServer();

// Middleware
app.use(async (req, res) => {
  console.log(`${req.method} ${req.path} - ${new Date().toISOString()}`);
});
```

```
// Routes
app.get('/', (req, res) => {
    res.json({ message: 'Hello from scratch HTTP server!' });
});

app.get('/users', (req, res) => {
    res.json({ users: [{ id: 1, name: 'John' }] });
});

app.post('/users', (req, res) => {
    res.status(201).json({ message: 'User created', user: req.body });
});

// Start server
app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

Jira-like Ticket System with Express

Project Structure for Ticket System

typescript

```typescript
// src/types/ticket.ts
export interface Ticket {
  id: string;
  title: string;
  description: string;
  status: TicketStatus;
  priority: Priority;
  assigneeId?: string;
  reporterId: string;
  projectId: string;
  labels: string[];
  createdAt: Date;
  updatedAt: Date;
  dueDate?: Date;
}

export enum TicketStatus {
  TODO = 'todo',
  IN_PROGRESS = 'in_progress',
  IN_REVIEW = 'in_review',
  DONE = 'done'
}

export enum Priority {
  LOW = 'low',
  MEDIUM = 'medium',
  HIGH = 'high',
  CRITICAL = 'critical'
}

export interface Project {
  id: string;
  name: string;
  description: string;
  ownerId: string;
  members: string[];
  createdAt: Date;
}

export interface User {
  id: string;
  name: string;
  email: string;
  role: UserRole;
  avatar?: string;
}
```

```
export enum UserRole {
    ADMIN = 'admin',
    PROJECT_MANAGER = 'project_manager',
    DEVELOPER = 'developer',
    TESTER = 'tester'
}
```

Ticket Controller

typescript

```typescript
// src/controllers/ticketController.ts
import { Request, Response } from 'express';
import { TicketService } from '../services/ticketService';
import { ResponseHelper } from '../utils/responseHelper';

export class TicketController {
  private ticketService: TicketService;

  constructor() {
    this.ticketService = new TicketService();
  }

  getAllTickets = async (req: Request, res: Response) => {
    try {
      const { page = 1, limit = 10, status, priority, assigneeId } = req.query;

      const filters = {
        status: status as string,
        priority: priority as string,
        assigneeId: assigneeId as string
      };

      const tickets = await this.ticketService.getAllTickets(
        Number(page),
        Number(limit),
        filters
      );

      res.json(ResponseHelper.success(tickets));
    } catch (error) {
      res.status(500).json(ResponseHelper.error('Failed to fetch tickets'));
    }
  };

  getTicketById = async (req: Request, res: Response) => {
    try {
      const { id } = req.params;
      const ticket = await this.ticketService.getTicketById(id);

      if (!ticket) {
        return res.status(404).json(ResponseHelper.error('Ticket not found'));
      }

      res.json(ResponseHelper.success(ticket));
    } catch (error) {
      res.status(500).json(ResponseHelper.error('Failed to fetch ticket'));
```

```
    }
  };

  createTicket = async (req: Request, res: Response) => {
    try {
      const ticketData = req.body;
      const ticket = await this.ticketService.createTicket(ticketData);

      res.status(201).json(ResponseHelper.success(ticket, 'Ticket created successfully'));
    } catch (error) {
      res.status(500).json(ResponseHelper.error('Failed to create ticket'));
    }
  };

  updateTicket = async (req: Request, res: Response) => {
    try {
      const { id } = req.params;
      const updateData = req.body;

      const ticket = await this.ticketService.updateTicket(id, updateData);

      if (!ticket) {
        return res.status(404).json(ResponseHelper.error('Ticket not found'));
      }

      res.json(ResponseHelper.success(ticket, 'Ticket updated successfully'));
    } catch (error) {
      res.status(500).json(ResponseHelper.error('Failed to update ticket'));
    }
  };

  deleteTicket = async (req: Request, res: Response) => {
    try {
      const { id } = req.params;
      const deleted = await this.ticketService.deleteTicket(id);

      if (!deleted) {
        return res.status(404).json(ResponseHelper.error('Ticket not found'));
      }

      res.json(ResponseHelper.success(null, 'Ticket deleted successfully'));
    } catch (error) {
      res.status(500).json(ResponseHelper.error('Failed to delete ticket'));
    }
  };
}
```

Ticket Routes

typescript

```typescript
// src/routes/ticketRoutes.ts
import { Router } from 'express';
import { TicketController } from '../controllers/ticketController';
import { validateSchema } from '../middleware/validation';
import { authenticate } from '../middleware/auth';
import { ticketSchema, updateTicketSchema } from '../schemas/ticketSchema';

const router = Router();
const ticketController = new TicketController();

// All routes require authentication
router.use(authenticate);

router.get('/', ticketController.getAllTickets);
router.get('/:id', ticketController.getTicketById);
router.post('/', validateSchema(ticketSchema), ticketController.createTicket);
router.put('/:id', validateSchema(updateTicketSchema), ticketController.updateTicket);
router.delete('/:id', ticketController.deleteTicket);

export default router;
```

PostgreSQL Integration

Database Setup

sql

```sql
-- Database schema for ticket system
CREATE DATABASE ticket_system;

-- Users table
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    role VARCHAR(50) NOT NULL DEFAULT 'developer',
    avatar_url VARCHAR(500),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Projects table
CREATE TABLE projects (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(200) NOT NULL,
    description TEXT,
    owner_id UUID REFERENCES users(id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Project members junction table
CREATE TABLE project_members (
    project_id UUID REFERENCES projects(id) ON DELETE CASCADE,
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    joined_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (project_id, user_id)
);

-- Tickets table
CREATE TABLE tickets (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    title VARCHAR(300) NOT NULL,
    description TEXT,
    status VARCHAR(50) NOT NULL DEFAULT 'todo',
    priority VARCHAR(50) NOT NULL DEFAULT 'medium',
    assignee_id UUID REFERENCES users(id) ON DELETE SET NULL,
    reporter_id UUID REFERENCES users(id) ON DELETE CASCADE,
    project_id UUID REFERENCES projects(id) ON DELETE CASCADE,
    labels TEXT[],
    due_date TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Comments table
CREATE TABLE comments (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    ticket_id UUID REFERENCES tickets(id) ON DELETE CASCADE,
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Indexes for better performance
CREATE INDEX idx_tickets_status ON tickets(status);
CREATE INDEX idx_tickets_assignee ON tickets(assignee_id);
CREATE INDEX idx_tickets_project ON tickets(project_id);
CREATE INDEX idx_comments_ticket ON comments(ticket_id);
```

Database Connection with pg

typescript

```typescript
// src/config/database.ts
import { Pool, PoolClient } from 'pg';
import { config } from './environment';

class Database {
  private pool: Pool;

  constructor() {
    this.pool = new Pool({
      connectionString: config.databaseUrl,
      ssl: config.nodeEnv === 'production' ? { rejectUnauthorized: false } : false,
      max: 20,
      idleTimeoutMillis: 30000,
      connectionTimeoutMillis: 2000,
    });
  }

  async query(text: string, params?: any[]): Promise<any> {
    const client = await this.pool.connect();
    try {
      const result = await client.query(text, params);
      return result;
    } finally {
      client.release();
    }
  }

  async getClient(): Promise<PoolClient> {
    return this.pool.connect();
  }

  async close(): Promise<void> {
    await this.pool.end();
  }
}

export const db = new Database();
```

Prisma ORM Guide

Installation and Setup

```bash
```

```
npm install prisma @prisma/client
npx prisma init
```

Prisma Schema

prisma

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

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

model User {
  id        String   @id @default(uuid())
  name      String
  email     String   @unique
  password  String
  role      UserRole @default(DEVELOPER)
  avatarUrl String?  @map("avatar_url")
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  // Relations
  ownedProjects    Project[]       @relation("ProjectOwner")
  projectMembers   ProjectMember[]
  assignedTickets  Ticket[]        @relation("TicketAssignee")
  reportedTickets  Ticket[]        @relation("TicketReporter")
  comments         Comment[]

  @@map("users")
}

model Project {
  id          String   @id @default(uuid())
  name        String
  description String?
  ownerId     String   @map("owner_id")
  createdAt   DateTime @default(now()) @map("created_at")
  updatedAt   DateTime @updatedAt @map("updated_at")

  // Relations
  owner   User           @relation("ProjectOwner", fields: [ownerId], references: [id], onDelete: Cascade)
  members ProjectMember[]
  tickets Ticket[]

  @@map("projects")
}
```

```
model ProjectMember {
  projectId String   @map("project_id")
  userId    String   @map("user_id")
  joinedAt  DateTime @default(now()) @map("joined_at")

  // Relations
  project Project @relation(fields: [projectId], references: [id], onDelete: Cascade)
  user    User    @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@id([projectId, userId])
  @@map("project_members")
}

model Ticket {
  id          String       @id @default(uuid())
  title       String
  description String?
  status      TicketStatus  @default(TODO)
  priority    Priority      @default(MEDIUM)
  assigneeId  String?       @map("assignee_id")
  reporterId  String        @map("reporter_id")
  projectId   String        @map("project_id")
  labels      String[]
  dueDate     DateTime?     @map("due_date")
  createdAt   DateTime      @default(now()) @map("created_at")
  updatedAt   DateTime      @updatedAt @map("updated_at")

  // Relations
  assignee User?     @relation("TicketAssignee", fields: [assigneeId], references: [id], onDelete: SetNull)
  reporter User      @relation("TicketReporter", fields: [reporterId], references: [id], onDelete: Cascade)
  project  Project   @relation(fields: [projectId], references: [id], onDelete: Cascade)
  comments Comment[]

  @@map("tickets")
}

model Comment {
  id       String   @id @default(uuid())
  ticketId String   @map("ticket_id")
  userId   String   @map("user_id")
  content  String
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  // Relations
  ticket Ticket @relation(fields: [ticketId], references: [id], onDelete: Cascade)
  user   User   @relation(fields: [userId], references: [id], onDelete: Cascade)
```

```
  @@map("comments")
}

enum UserRole {
  ADMIN
  PROJECT_MANAGER
  DEVELOPER
  TESTER
}

enum TicketStatus {
  TODO
  IN_PROGRESS
  IN_REVIEW
  DONE
}

enum Priority {
  LOW
  MEDIUM
  HIGH
  CRITICAL
}
```

Prisma Service

typescript

```typescript
// src/services/prismaService.ts
import { PrismaClient } from '@prisma/client';

class PrismaService {
  private prisma: PrismaClient;

  constructor() {
    this.prisma = new PrismaClient({
      log: ['query', 'info', 'warn', 'error'],
    });
  }

  async onModuleInit() {
    await this.prisma.$connect();
  }

  async onModuleDestroy() {
    await this.prisma.$disconnect();
  }

  get client() {
    return this.prisma;
  }
}

export const prismaService = new PrismaService();
export const prisma = prismaService.client;
```

Using Prisma in Services

typescript

```typescript
// src/services/ticketService.ts
import { prisma } from './prismaService';
import { Ticket, TicketStatus, Priority } from '@prisma/client';

export class TicketService {
  async getAllTickets(
    page: number = 1,
    limit: number = 10,
    filters: {
      status?: TicketStatus;
      priority?: Priority;
      assigneeId?: string;
      projectId?: string;
    } = {}
  ) {
    const skip = (page - 1) * limit;

    const where: any = {};
    if (filters.status) where.status = filters.status;
    if (filters.priority) where.priority = filters.priority;
    if (filters.assigneeId) where.assigneeId = filters.assigneeId;
    if (filters.projectId) where.projectId = filters.projectId;

    const [tickets, total] = await Promise.all([
      prisma.ticket.findMany({
        where,
        skip,
        take: limit,
        include: {
          assignee: {
            select: { id: true, name: true, email: true }
          },
          reporter: {
            select: { id: true, name: true, email: true }
          },
          project: {
            select: { id: true, name: true }
          },
          comments: {
            select: { id: true, content: true, createdAt: true }
          }
        },
        orderBy: { createdAt: 'desc' }
      }),
      prisma.ticket.count({ where })
    ]);
```

```
    return {
      tickets,
      pagination: {
        page,
        limit,
        total,
        totalPages: Math.ceil(total / limit)
      }
    };
  }

  async getTicketById(id: string) {
    return prisma.ticket.findUnique({
      where: { id },
      include: {
        assignee: true,
        reporter: true,
        project: true,
        comments: {
          include: {
            user: {
              select: { id: true, name: true, email: true }
            }
          },
          orderBy: { createdAt: 'asc' }
        }
      }
    });
  }

  async createTicket(data: {
    title: string;
    description?: string;
    status?: TicketStatus;
    priority?: Priority;
    assigneeId?: string;
    reporterId: string;
    projectId: string;
    labels?: string[];
    dueDate?: Date;
  }) {
    return prisma.ticket.create({
      data,
      include: {
        assignee: true,
        reporter: true,
```

```
            project: true
        }
    });
}

async updateTicket(id: string, data: Partial<Ticket>) {
    return prisma.ticket.update({
        where: { id },
        data: {
            ...data,
            updatedAt: new Date()
        },
        include: {
            assignee: true,
            reporter: true,
            project: true
        }
    });
}

async deleteTicket(id: string) {
    try {
        await prisma.ticket.delete({
            where: { id }
        });
        return true;
    } catch (error) {
        return false;
    }
}

async getTicketsByProject(projectId: string) {
    return prisma.ticket.findMany({
        where: { projectId },
        include: {
            assignee: true,
            reporter: true
        },
        orderBy: { createdAt: 'desc' }
    });
}

async getTicketsByUser(userId: string) {
    return prisma.ticket.findMany({
        where: {
            OR: [
                { assigneeId: userId },
```

```
                { reporterId: userId }
            ]
        },
        include: {
            project: true,
            assignee: true,
            reporter: true
        },
        orderBy: { createdAt: 'desc' }
    });
  }
}
```

Arcjet Security

Installation and Setup

bash

```
npm install @arcjet/node
```

Arcjet Configuration

typescript

```ts
// src/config/arcjet.ts
import arcjet, { detectBot, fixedWindow, shield } from '@arcjet/node';
import { config } from './environment';

export const aj = arcjet({
    key: process.env.ARCJET_KEY!,
    characteristics: ['ip.src'],
    rules: [
        // Rate limiting
        fixedWindow({
            mode: 'LIVE',
            window: '1m',
            max: 100,
        }),

        // Bot detection
        detectBot({
            mode: 'LIVE',
            block: ['AUTOMATED'],
        }),

        // Shield protection against common attacks
        shield({
            mode: 'LIVE',
        }),
    ],
});

// API-specific protection
export const apiProtection = arcjet({
    key: process.env.ARCJET_KEY!,
    characteristics: ['ip.src'],
    rules: [
        fixedWindow({
            mode: 'LIVE',
            window: '1m',
            max: 60, // More restrictive for API
        }),
        shield({
            mode: 'LIVE',
        }),
    ],
});

// Authentication endpoint protection
export const authProtection = arcjet({
```

```
    key: process.env.ARCJET_KEY!,
    characteristics: ['ip.src'],
    rules: [
      fixedWindow({
        mode: 'LIVE',
        window: '15m',
        max: 5, // Very restrictive for login attempts
      }),
      shield({
        mode: 'LIVE',
      }),
    ],
});
```

Arcjet Middleware

typescript

```typescript
// src/middleware/arcjet.ts
import { Request, Response, NextFunction } from 'express';
import { aj, apiProtection, authProtection } from '../config/arcjet';

export const arcjetMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const decision = aj.protect(req);

  if (decision.isDenied()) {
    if (decision.reason.isRateLimit()) {
      return res.status(429).json({
        error: 'Too many requests',
        retryAfter: decision.reason.resetTime
      });
    }

    if (decision.reason.isBot()) {
      return res.status(403).json({
        error: 'Bot traffic not allowed'
      });
    }

    return res.status(403).json({
      error: 'Request blocked by security policy'
    });
  }

  next();
};

export const apiArcjetMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const decision = apiProtection.protect(req);

  if (decision.isDenied()) {
    return res.status(429).json({
      error: 'API rate limit exceeded',
      retryAfter: decision.reason.resetTime
    });
  }

  next();
};

export const authArcjetMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const decision = authProtection.protect(req);

  if (decision.isDenied()) {
```

```
      return res.status(429).json({
        error: 'Too many authentication attempts',
        retryAfter: decision.reason.resetTime
      });
    }

    next();
};
```

JWT Authentication

Installation

bash

```
npm install jsonwebtoken bcryptjs
npm install -D @types/jsonwebtoken @types/bcryptjs
```

JWT Service

typescript

```typescript
// src/services/jwtService.ts
import jwt from 'jsonwebtoken';
import { config } from '../config/environment';

export interface JWTPayload {
  userId: string;
  email: string;
  role: string;
}

export class JWTService {
  private secret: string;
  private expiresIn: string;

  constructor() {
    this.secret = config.jwtSecret;
    this.expiresIn = config.jwtExpiresIn;
  }

  generateToken(payload: JWTPayload): string {
    return jwt.sign(payload, this.secret, {
      expiresIn: this.expiresIn,
      issuer: 'ticket-system',
      audience: 'ticket-system-users'
    });
  }

  generateRefreshToken(userId: string): string {
    return jwt.sign({ userId }, this.secret, {
      expiresIn: '7d',
      issuer: 'ticket-system',
      audience: 'ticket-system-refresh'
    });
  }

  verifyToken(token: string): JWTPayload {
    try {
      return jwt.verify(token, this.secret, {
        issuer: 'ticket-system',
        audience: 'ticket-system-users'
      }) as JWTPayload;
    } catch (error) {
      throw new Error('Invalid token');
    }
  }
```

```
  verifyRefreshToken(token: string): { userId: string } {
    try {
      return jwt.verify(token, this.secret, {
        issuer: 'ticket-system',
        audience: 'ticket-system-refresh'
      }) as { userId: string };
    } catch (error) {
      throw new Error('Invalid refresh token');
    }
  }

  decodeToken(token: string): any {
    return jwt.decode(token);
  }
}

export const jwtService = new JWTService();
```

Authentication Service

typescript

```typescript
// src/services/authService.ts
import bcrypt from 'bcryptjs';
import { prisma } from './prismaService';
import { jwtService, JWTPayload } from './jwtService';
import { User, UserRole } from '@prisma/client';

export interface LoginCredentials {
  email: string;
  password: string;
}

export interface RegisterData {
  name: string;
  email: string;
  password: string;
  role?: UserRole;
}

export interface AuthResponse {
  user: Omit<User, 'password'>;
  accessToken: string;
  refreshToken: string;
}

export class AuthService {
  async register(data: RegisterData): Promise<AuthResponse> {
    // Check if user already exists
    const existingUser = await prisma.user.findUnique({
      where: { email: data.email }
    });

    if (existingUser) {
      throw new Error('User already exists with this email');
    }

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

    // Create user
    const user = await prisma.user.create({
      data: {
        name: data.name,
        email: data.email,
        password: hashedPassword,
        role: data.role || UserRole.DEVELOPER
```

```
    }
  });

  // Generate tokens
  const payload: JWTPayload = {
    userId: user.id,
    email: user.email,
    role: user.role
  };

  const accessToken = jwtService.generateToken(payload);
  const refreshToken = jwtService.generateRefreshToken(user.id);

  // Remove password from response
  const { password, ...userWithoutPassword } = user;

  return {
    user: userWithoutPassword,
    accessToken,
    refreshToken
  };
}

async login(credentials: LoginCredentials): Promise<AuthResponse> {
  // Find user
  const user = await prisma.user.findUnique({
    where: { email: credentials.email }
  });

  if (!user) {
    throw new Error('Invalid credentials');
  }

  // Verify password
  const isValidPassword = await bcrypt.compare(credentials.password, user.password);

  if (!isValidPassword) {
    throw new Error('Invalid credentials');
  }

  // Generate tokens
  const payload: JWTPayload = {
    userId: user.id,
    email: user.email,
    role: user.role
  };
```

```
    const accessToken = jwtService.generateToken(payload);
    const refreshToken = jwtService.generateRefreshToken(user.id);

    // Remove password from response
    const { password, ...userWithoutPassword } = user;

    return {
      user: userWithoutPassword,
      accessToken,
      refreshToken
    };
  }

  async refreshToken(refreshToken: string): Promise<{ accessToken: string }> {
    try {
      const { userId } = jwtService.verifyRefreshToken(refreshToken);

      const user = await prisma.user.findUnique({
        where: { id: userId }
      });

      if (!user) {
        throw new Error('User not found');
      }

      const payload: JWTPayload = {
        userId: user.id,
        email: user.email,
        role: user.role
      };

      const accessToken = jwtService.generateToken(payload);

      return { accessToken };
    } catch (error) {
      throw new Error('Invalid refresh token');
    }
  }

  async getCurrentUser(userId: string): Promise<Omit<User, 'password'> | null> {
    const user = await prisma.user.findUnique({
      where: { id: userId },
      select: {
        id: true,
        name: true,
        email: true,
        role: true,
```

```
            avatarUrl: true,
            createdAt: true,
            updatedAt: true
        }
    });

    return user;
  }

  async changePassword(userId: string, currentPassword: string, newPassword: string): Promise<void> {
    const user = await prisma.user.findUnique({
      where: { id: userId }
    });

    if (!user) {
      throw new Error('User not found');
    }

    const isValidPassword = await bcrypt.compare(currentPassword, user.password);

    if (!isValidPassword) {
      throw new Error('Current password is incorrect');
    }

    const saltRounds = 12;
    const hashedNewPassword = await bcrypt.hash(newPassword, saltRounds);

    await prisma.user.update({
      where: { id: userId },
      data: { password: hashedNewPassword }
    });
  }
}

export const authService = new AuthService();
```

Authentication Middleware

typescript

```typescript
// src/middleware/auth.ts
import { Request, Response, NextFunction } from 'express';
import { jwtService } from '../services/jwtService';
import { authService } from '../services/authService';

// Extend Request interface to include user
declare global {
  namespace Express {
    interface Request {
      user?: {
        id: string;
        email: string;
        role: string;
      };
    }
  }
}

export const authenticate = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith('Bearer ')) {
      return res.status(401).json({ error: 'Access token required' });
    }

    const token = authHeader.substring(7); // Remove 'Bearer ' prefix

    const payload = jwtService.verifyToken(token);

    // Verify user still exists
    const user = await authService.getCurrentUser(payload.userId);

    if (!user) {
      return res.status(401).json({ error: 'User not found' });
    }

    req.user = {
      id: payload.userId,
      email: payload.email,
      role: payload.role
    };

    next();
  } catch (error) {
    return res.status(401).json({ error: 'Invalid or expired token' });
```

```
  }
};

export const authorize = (roles: string[]) => {
  return (req: Request, res: Response, next: NextFunction) => {
    if (!req.user) {
      return res.status(401).json({ error: 'Authentication required' });
    }

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

    next();
  };
};

// Optional authentication - doesn't fail if no token
export const optionalAuth = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const authHeader = req.headers.authorization;

    if (authHeader && authHeader.startsWith('Bearer ')) {
      const token = authHeader.substring(7);
      const payload = jwtService.verifyToken(token);

      const user = await authService.getCurrentUser(payload.userId);

      if (user) {
        req.user = {
          id: payload.userId,
          email: payload.email,
          role: payload.role
        };
      }
    }

    next();
  } catch (error) {
    // Continue without authentication
    next();
  }
};
```

Authentication Controller

typescript

```ts
// src/controllers/authController.ts
import { Request, Response } from 'express';
import { authService } from '../services/authService';
import { ResponseHelper } from '../utils/responseHelper';

export class AuthController {
  register = async (req: Request, res: Response) => {
    try {
      const { name, email, password, role } = req.body;

      const result = await authService.register({
        name,
        email,
        password,
        role
      });

      res.status(201).json(ResponseHelper.success(result, 'User registered successfully'));
    } catch (error) {
      const errorMessage = error instanceof Error ? error.message : 'Registration failed';
      res.status(400).json(ResponseHelper.error(errorMessage));
    }
  };

  login = async (req: Request, res: Response) => {
    try {
      const { email, password } = req.body;

      const result = await authService.login({ email, password });

      res.json(ResponseHelper.success(result, 'Login successful'));
    } catch (error) {
      const errorMessage = error instanceof Error ? error.message : 'Login failed';
      res.status(401).json(ResponseHelper.error(errorMessage));
    }
  };

  refreshToken = async (req: Request, res: Response) => {
    try {
      const { refreshToken } = req.body;

      if (!refreshToken) {
        return res.status(400).json(ResponseHelper.error('Refresh token required'));
      }

      const result = await authService.refreshToken(refreshToken);
```

```
      res.json(ResponseHelper.success(result));
    } catch (error) {
      const errorMessage = error instanceof Error ? error.message : 'Token refresh failed';
      res.status(401).json(ResponseHelper.error(errorMessage));
    }
};

getCurrentUser = async (req: Request, res: Response) => {
    try {
      if (!req.user) {
        return res.status(401).json(ResponseHelper.error('User not authenticated'));
      }

      const user = await authService.getCurrentUser(req.user.id);

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

      res.json(ResponseHelper.success(user));
    } catch (error) {
      res.status(500).json(ResponseHelper.error('Failed to fetch user'));
    }
};

changePassword = async (req: Request, res: Response) => {
    try {
      if (!req.user) {
        return res.status(401).json(ResponseHelper.error('User not authenticated'));
      }

      const { currentPassword, newPassword } = req.body;

      await authService.changePassword(req.user.id, currentPassword, newPassword);

      res.json(ResponseHelper.success(null, 'Password changed successfully'));
    } catch (error) {
      const errorMessage = error instanceof Error ? error.message : 'Password change failed';
      res.status(400).json(ResponseHelper.error(errorMessage));
    }
};

logout = async (req: Request, res: Response) => {
    // In a production app, you might want to blacklist the token
    // For now, just return success (client should remove token)
    res.json(ResponseHelper.success(null, 'Logout successful'));
```

```
  };
}
```

Complete Application Setup

typescript

```typescript
// src/app.ts
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';
import morgan from 'morgan';
import { config } from './config/environment';
import { connectDB } from './config/database';
import { arcjetMiddleware, apiArcjetMiddleware, authArcjetMiddleware } from './middleware/arcjet';

// Route imports
import authRoutes from './routes/authRoutes';
import userRoutes from './routes/userRoutes';
import projectRoutes from './routes/projectRoutes';
import ticketRoutes from './routes/ticketRoutes';

class App {
  public app: express.Application;

  constructor() {
    this.app = express();
    this.configureMiddleware();
    this.configureRoutes();
    this.configureErrorHandling();
  }

  private configureMiddleware() {
    // Security middleware
    this.app.use(helmet());
    this.app.use(cors({
      origin: process.env.FRONTEND_URL || 'http://localhost:3000',
      credentials: true
    }));

    // Arcjet protection
    this.app.use(arcjetMiddleware);

    // Logging
    this.app.use(morgan('combined'));

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

  private configureRoutes() {
    // Health check
```

```
        this.app.get('/health', (req, res) => {
            res.json({ status: 'OK', timestamp: new Date().toISOString() });
        });

        // API routes with additional protection
        this.app.use('/api/auth', authArcjetMiddleware, authRoutes);
        this.app.use('/api', apiArcjetMiddleware);
        this.app.use('/api/users', userRoutes);
        this.app.use('/api/projects', projectRoutes);
        this.app.use('/api/tickets', ticketRoutes);

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

    private configureErrorHandling() {
        this.app.use((err: Error, req: express.Request, res: express.Response, next: express.NextFunction) => {
            console.error(err.stack);
            res.status(500).json({ error: 'Something went wrong!' });
        });
    }

    public async start() {
        try {
            await connectDB();

            const PORT = config.port;
            this.app.listen(PORT, () => {
                console.log(`Server running on port ${PORT}`);
                console.log(`Environment: ${config.nodeEnv}`);
            });
        } catch (error) {
            console.error('Failed to start server:', error);
            process.exit(1);
        }
    }
}

// Start the application
const app = new App();
app.start();

export default app;
```

Package.json Scripts

```json
```

```
{
  "name": "ticket-system-api",
  "version": "1.0.0",
  "description": "Jira-like ticket system API",
  "main": "dist/app.js",
  "scripts": {
    "build": "tsc",
    "start": "node dist/app.js",
    "dev": "nodemon --exec ts-node src/app.ts",
    "db:generate": "prisma generate",
    "db:migrate": "prisma migrate deploy",
    "db:migrate:dev": "prisma migrate dev",
    "db:seed": "ts-node prisma/seed.ts",
    "db:studio": "prisma studio",
    "test": "jest",
    "test:watch": "jest --watch",
    "lint": "eslint src/**/*.ts",
    "lint:fix": "eslint src/**/*.ts --fix"
  },
  "dependencies": {
    "express": "^4.18.2",
    "typescript": "^5.0.0",
    "@types/express": "^4.17.17",
    "@prisma/client": "^5.0.0",
    "prisma": "^5.0.0",
    "jsonwebtoken": "^9.0.0",
    "bcryptjs": "^2.4.3",
    "@arcjet/node": "^1.0.0",
    "joi": "^17.9.0",
    "cors": "^2.8.5",
    "helmet": "^7.0.0",
    "morgan": "^1.10.0",
    "dotenv": "^16.0.0",
    "pg": "^8.11.0"
  },
  "devDependencies": {
    "@types/node": "^20.0.0",
    "@types/jsonwebtoken": "^9.0.0",
    "@types/bcryptjs": "^2.4.0",
    "@types/cors": "^2.8.0",
    "@types/morgan": "^1.9.0",
    "@types/pg": "^8.10.0",
    "ts-node": "^10.9.0",
    "nodemon": "^3.0.0",
    "jest": "^29.0.0",
    "@types/jest": "^29.0.0",
```

```
    "eslint": "^8.0.0",
    "@typescript-eslint/eslint-plugin": "^6.0.0",
    "@typescript-eslint/parser": "^6.0.0"
  }
}
```

Summary

This comprehensive guide covers:

1. **JavaScript Fundamentals** - Variables, functions, async programming, ES6+ features
2. **TypeScript** - Types, interfaces, classes, generics, configuration
3. **Express.js** - Server setup, middleware, routing, error handling
4. **Project Structure** - Organized codebase with proper separation of concerns
5. **REST API** - RESTful principles, response formatting, validation
6. **HTTP Server from Scratch** - Pure Node.js HTTP server implementation
7. **Jira-like Ticket System** - Complete ticket management system with Express
8. **PostgreSQL** - Database schema, connection management, queries
9. **Prisma ORM** - Schema definition, client usage, relationships
10. **Arcjet Security** - Rate limiting, bot detection, attack protection
11. **JWT Authentication** - Token generation, validation, middleware

Each section builds upon the previous ones, culminating in a production-ready ticket management system with security, authentication, and database integration. The code examples are comprehensive and can be used as reference for building real applications.