Table of Contents

# Bab 20: TypeScript dan Modern Web Development

## Penjelasan Materi

TypeScript telah menjadi bagian integral dari pengembangan web modern. Dengan fitur-fitur canggihnya, TypeScript memungkinkan developer untuk membangun aplikasi web yang lebih aman, maintainable, dan scalable. Bab ini akan membahas bagaimana TypeScript berinteraksi dengan berbagai teknologi web modern dan best practices dalam pengembangan aplikasi web.

## Analogi yang Mudah Dipahami

Bayangkan pengembangan web modern seperti membangun gedung pencakar langit: - TypeScript seperti blueprint yang detail dan terukur - Framework modern seperti fondasi dan kerangka bangunan - Tools dan libraries seperti peralatan konstruksi - Testing dan deployment seperti quality control dan proses serah terima - Monitoring dan maintenance seperti perawatan gedung

## Point Penting

1. **Modern Frameworks**
   * React with TypeScript
   * Vue with TypeScript
   * Angular (Native TypeScript)
   * Next.js with TypeScript
   * Nuxt with TypeScript
2. **Build Tools**
   * Vite
   * Webpack
   * Rollup
   * esbuild
   * Parcel
3. **State Management**
   * Redux Toolkit
   * Vuex/Pinia
   * MobX
   * Zustand
   * Jotai/Recoil
4. **API Integration**
   * REST
   * GraphQL
   * WebSocket
   * gRPC
   * tRPC
5. **Modern Features**
   * Server Components
   * Static Site Generation
   * Incremental Static Regeneration
   * API Routes
   * Middleware

## Contoh Kode dan Penjelasan

```typescript // 1. React Component with TypeScript // UserProfile.tsx interface User { id: number; name: string; email: string; role: ‘admin’ | ‘user’; }

interface UserProfileProps { user: User; onUpdate: (user: User) => Promise; }

const UserProfile: React.FC = ({ user, onUpdate }) => { const [isEditing, setIsEditing] = useState(false); const [formData, setFormData] = useState(user);

const handleSubmit = async (e: React.FormEvent) => {  
 e.preventDefault();  
 await onUpdate(formData);  
 setIsEditing(false);  
};  
  
return (  
 <div className="user-profile">  
 {isEditing ? (  
 <form onSubmit={handleSubmit}>  
 <input  
 type="text"  
 value={formData.name}  
 onChange={e => setFormData({ ...formData, name: e.target.value })}  
 />  
 <button type="submit">Save</button>  
 </form>  
 ) : (  
 <div>  
 <h2>{user.name}</h2>  
 <p>{user.email}</p>  
 <button onClick={() => setIsEditing(true)}>Edit</button>  
 </div>  
 )}  
 </div>  
);

};

// 2. Vue Component with TypeScript // UserProfile.vue

// 3. Next.js API Route // pages/api/users/[id].ts import type { NextApiRequest, NextApiResponse } from ‘next’; import { prisma } from ‘../../../lib/prisma’;

export default async function handler( req: NextApiRequest, res: NextApiResponse ) { const { id } = req.query; const userId = Number(id);

switch (req.method) {  
 case 'GET':  
 try {  
 const user = await prisma.user.findUnique({  
 where: { id: userId }  
 });  
 if (!user) {  
 return res.status(404).json({ error: 'User not found' });  
 }  
 return res.status(200).json(user);  
 } catch (error) {  
 return res.status(500).json({ error: 'Internal server error' });  
 }  
   
 case 'PUT':  
 try {  
 const updatedUser = await prisma.user.update({  
 where: { id: userId },  
 data: req.body  
 });  
 return res.status(200).json(updatedUser);  
 } catch (error) {  
 return res.status(500).json({ error: 'Internal server error' });  
 }  
   
 default:  
 res.setHeader('Allow', ['GET', 'PUT']);  
 return res.status(405).json({ error: \`Method \${req.method} not allowed\` });  
}

}

// 4. GraphQL Schema and Resolvers // schema.ts import { gql } from ‘apollo-server-express’; import { makeExecutableSchema } from ‘@graphql-tools/schema’;

const typeDefs = gql` type User { id: ID! name: String! email: String! role: UserRole! posts: [Post!]! }

enum UserRole {  
 ADMIN  
 USER  
}  
  
type Post {  
 id: ID!  
 title: String!  
 content: String!  
 author: User!  
}  
  
type Query {  
 user(id: ID!): User  
 users: [User!]!  
 posts: [Post!]!  
}  
  
type Mutation {  
 createUser(input: CreateUserInput!): User!  
 updateUser(id: ID!, input: UpdateUserInput!): User!  
}  
  
input CreateUserInput {  
 name: String!  
 email: String!  
 role: UserRole!  
}  
  
input UpdateUserInput {  
 name: String  
 email: String  
 role: UserRole  
}

`;

// resolvers.ts interface Context { prisma: PrismaClient; user?: AuthenticatedUser; }

const resolvers = { Query: { user: async (*: any, { id }: { id: string }, ctx: Context) => { return ctx.prisma.user.findUnique({ where: { id: Number(id) } }); }, users: async (*: any, \_\_: any, ctx: Context) => { return ctx.prisma.user.findMany(); } }, Mutation: { createUser: async (\_: any, { input }: { input: CreateUserInput }, ctx: Context) => { return ctx.prisma.user.create({ data: input }); } } };

// 5. Redux Toolkit Store // store/userSlice.ts import { createSlice, createAsyncThunk } from ‘@reduxjs/toolkit’;

interface UserState { users: User[]; loading: boolean; error: string | null; }

const initialState: UserState = { users: [], loading: false, error: null };

export const fetchUsers = createAsyncThunk( ‘users/fetchUsers’, async (\_, { rejectWithValue }) => { try { const response = await fetch(‘/api/users’); if (!response.ok) { throw new Error(‘Failed to fetch users’); } return await response.json(); } catch (error) { return rejectWithValue(error.message); } } );

const userSlice = createSlice({ name: ‘users’, initialState, reducers: {}, extraReducers: (builder) => { builder .addCase(fetchUsers.pending, (state) => { state.loading = true; }) .addCase(fetchUsers.fulfilled, (state, action) => { state.loading = false; state.users = action.payload; }) .addCase(fetchUsers.rejected, (state, action) => { state.loading = false; state.error = action.payload as string; }); } });

// 6. WebSocket Service // services/websocket.ts interface WebSocketMessage { type: string; payload: any; }

class WebSocketService { private ws: WebSocket | null = null; private messageHandlers: Map<string, (payload: any) => void> = new Map();

connect(url: string): Promise<void> {  
 return new Promise((resolve, reject) => {  
 this.ws = new WebSocket(url);  
  
 this.ws.onopen = () => {  
 console.log('WebSocket connected');  
 resolve();  
 };  
  
 this.ws.onerror = (error) => {  
 console.error('WebSocket error:', error);  
 reject(error);  
 };  
  
 this.ws.onmessage = (event) => {  
 const message: WebSocketMessage = JSON.parse(event.data);  
 const handler = this.messageHandlers.get(message.type);  
 if (handler) {  
 handler(message.payload);  
 }  
 };  
 });  
}  
  
on(type: string, handler: (payload: any) => void): void {  
 this.messageHandlers.set(type, handler);  
}  
  
send(type: string, payload: any): void {  
 if (this.ws?.readyState === WebSocket.OPEN) {  
 this.ws.send(JSON.stringify({ type, payload }));  
 }  
}

}

// 7. Custom Hook with TypeScript // hooks/useDebounce.ts function useDebounce(value: T, delay: number): T { const [debouncedValue, setDebouncedValue] = useState(value);

useEffect(() => {  
 const timer = setTimeout(() => {  
 setDebouncedValue(value);  
 }, delay);  
  
 return () => {  
 clearTimeout(timer);  
 };  
}, [value, delay]);  
  
return debouncedValue;

}

// 8. API Client with TypeScript // api/client.ts interface ApiConfig { baseUrl: string; headers?: Record<string, string>; timeout?: number; }

class ApiClient { private config: ApiConfig;

constructor(config: ApiConfig) {  
 this.config = {  
 timeout: 5000,  
 ...config  
 };  
}  
  
async request<T>(  
 method: string,  
 path: string,  
 data?: any  
): Promise<T> {  
 const controller = new AbortController();  
 const timeoutId = setTimeout(() => controller.abort(), this.config.timeout);  
  
 try {  
 const response = await fetch(\`\${this.config.baseUrl}\${path}\`, {  
 method,  
 headers: {  
 'Content-Type': 'application/json',  
 ...this.config.headers  
 },  
 body: data ? JSON.stringify(data) : undefined,  
 signal: controller.signal  
 });  
  
 if (!response.ok) {  
 throw new Error(\`HTTP error! status: \${response.status}\`);  
 }  
  
 return await response.json();  
 } finally {  
 clearTimeout(timeoutId);  
 }  
}  
  
get<T>(path: string): Promise<T> {  
 return this.request('GET', path);  
}  
  
post<T>(path: string, data: any): Promise<T> {  
 return this.request('POST', path, data);  
}

}

// 9. Server-Side Props with Next.js // pages/users/[id].tsx export const getServerSideProps: GetServerSideProps = async (context) => { const { id } = context.params as { id: string };

try {  
 const user = await prisma.user.findUnique({  
 where: { id: Number(id) },  
 include: { posts: true }  
 });  
  
 if (!user) {  
 return {  
 notFound: true  
 };  
 }  
  
 return {  
 props: {  
 user: JSON.parse(JSON.stringify(user))  
 }  
 };  
} catch (error) {  
 return {  
 props: {  
 error: 'Failed to fetch user'  
 }  
 };  
}

};

// 10. Middleware with TypeScript // middleware.ts import { NextResponse } from ‘next/server’; import type { NextRequest } from ‘next/server’;

export function middleware(request: NextRequest) { const token = request.cookies.get(‘token’);

if (!token && request.nextUrl.pathname.startsWith('/api')) {  
 return NextResponse.json(  
 { error: 'Authentication required' },  
 { status: 401 }  
 );  
}  
  
if (!token && request.nextUrl.pathname.startsWith('/dashboard')) {  
 return NextResponse.redirect(new URL('/login', request.url));  
}  
  
return NextResponse.next();

}

export const config = { matcher: [‘/api/:path*’, ’/dashboard/:path*’] }; ```

## Cara Kerja Modern Web Development

1. **Development Flow**:
   * Code writing
   * Type checking
   * Building
   * Testing
   * Deployment
2. **Application Architecture**:
   * Component structure
   * State management
   * API integration
   * Routing
   * Data flow
3. **Performance Optimization**:
   * Code splitting
   * Lazy loading
   * Caching
   * Server-side rendering
   * Static generation

## Tips dan Trik

1. **Project Structure**

* // ✅ Organized project structure  
  src/  
   ├── components/  
   │ ├── common/  
   │ └── features/  
   ├── hooks/  
   ├── services/  
   ├── store/  
   ├── types/  
   └── utils/

1. **Type Safety**

* // ✅ Use strict type checking  
  {  
   "compilerOptions": {  
   "strict": true,  
   "noImplicitAny": true,  
   "strictNullChecks": true  
   }  
  }

1. **API Type Safety**

* // ✅ Type-safe API calls  
  async function fetchUser<T extends { id: number }>(  
   id: number  
  ): Promise<T> {  
   const response = await fetch(\`/api/users/\${id}\`);  
   if (!response.ok) {  
   throw new Error('Failed to fetch user');  
   }  
   return response.json();  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Mengabaikan Type Safety**

* // ❌ Buruk: Implicit any  
  function processData(data) {  
   return data.map(item => item.value);  
  }  
    
  // ✅ Baik: Proper typing  
  interface DataItem {  
   value: string;  
  }  
    
  function processData(data: DataItem[]): string[] {  
   return data.map(item => item.value);  
  }

1. **Tidak Menggunakan Modern Features**

* // ❌ Buruk: Old-style components  
  class UserList extends React.Component {  
   state = { users: [] };  
   // ...  
  }  
    
  // ✅ Baik: Modern hooks and patterns  
  const UserList: React.FC = () => {  
   const [users, setUsers] = useState<User[]>([]);  
   // ...  
  };

1. **Mengabaikan Build Optimization**

* // ❌ Buruk: No code splitting  
  import { heavyComponent } from './heavy';  
    
  // ✅ Baik: Lazy loading  
  const HeavyComponent = React.lazy(() => import('./heavy'));

### Solusi:

1. Gunakan TypeScript strict mode
2. Implementasi modern patterns
3. Optimize build process
4. Use proper project structure
5. Implement proper error handling