Dynamic Return Types in TypeScript

Using TypeScript to Return Type Based on Parameter

The Challenge

- API endpoints return different data structures
- We want type safety when making API calls
- Desire for flexibility in handling query parameters

The Building Blocks

- Define types for our data structures
- User type with id, name, and password
- Task type with id and name

```
type User = {
  id: number;
  name: string;
  password: string;
}

type Task = {
  id: number;
  name: string;
}
```

Mapping Endpoints to Types

- Create a lookup table for API endpoints
- Map string literals to corresponding types
- '/user' maps to User type
- '/users' maps to User (array of Users)
- Essential for our dynamically typed fetch function

```
type ResponseJsonByEndpoint = {
  '/user': User;
  '/users': User[];
  '/task': Task;
  '/tasks': Task[];
}
```

The Magic: Dynamically Typed Fetch Function

- Asynchronous 'get' function
- Uses generic type parameter 'Endpoint'
- Accepts endpoint with optional query params
- Returns Promise of type based on endpoint
- Ensures type safety and flexibility

```
export async function get<
    Endpoint extends keyof ResponseJsonByEndpoint
>(
  endpoint: Endpoint | `${Endpoint}?${string}`
): Promise<ResponseJsonByEndpoint[Endpoint]> {
  const response = await fetch(`/api${endpoint}`);
  return await response.json();
}

type ResponseJsonByEndpoint = {
  '/user': User;
  '/users': User[];
  '/task': Task;
  '/tasks': Task[];
}
```

Putting It All Together

- Use the 'get' function for API calls
- TypeScript infers correct return types
- Works with and without query parameters
- Full type safety and autocompletion
- Robust and maintainable code

```
export async function get<
    Endpoint extends keyof ResponseJsonByEndpoint
>(
    endpoint: Endpoint | `${Endpoint}?${string}`
): Promise<ResponseJsonByEndpoint[Endpoint]> {
    const response = await fetch(`/api${endpoint}`);
    return await response.json();
}

type ResponseJsonByEndpoint = {
    '/user': User;
    '/users': User[];
    '/task': Task;
    '/tasks': Task[];
}

const users = await get('/users');
const limitedUsers = await get('/users?limit=10');
const task = await get('/task');
```

Benefits of This Approach

- Type safety throughout the API request process
- Autocompletion and IntelliSense support
- Flexible handling of endpoints and query parameters
- Self-documenting code

Conclusion

- TypeScript offers powerful tools for type-safe API interactions
- Leverage generics, mapped types, and template literal types
- Improve code quality and developer experience
- Encourage exploration of advanced TypeScript features

Questions?

Have you encountered challenges with typing API calls in your TypeScript projects?

Thank You!

Remember: In TypeScript, leverage the type system for safer, more maintainable code!