# 🟦 TypeScript-Specific Questions

**🔹 1. What are the main differences between any, unknown, and never in TypeScript?**

* **any** → TypeScript says: “Do whatever you want, I won’t check.” Like removing the helmet and riding – no safety.
* **unknown** → Safer than any. TypeScript says: “You can use it, but first check properly.” Like police asking for ID before letting you through.
* **never** → This means “it’ll never happen”. For example, a function that always throws an error or keeps running forever.

**🔹 2. How do you define and use generics in TypeScript?**

* Generics = flexible types. Example: You want to write a function that works for numbers, strings, arrays – anything.

function printItem<T>(item: T): void {

console.log(item);

}

* Think of <T> like a “template”. Instead of writing the same code again for different types, use generics once and reuse it.

**🔹 3. Explain Partial<T>, Pick<T, K>, and Omit<T, K> with examples.**

Assume this type:

type User = { name: string; email: string; age: number };

* **Partial<User>** → All properties become optional.

const update: Partial<User> = { name: "Pramod" };

* **Pick<User, 'name' | 'email'>** → Pick only selected fields.

const shortInfo: Pick<User, 'name' | 'email'> = { name: "P", email: "p@x.com" };

* **Omit<User, 'age'>** → Remove a field.

const withoutAge: Omit<User, 'age'> = { name: "P", email: "p@x.com" };

👉 Useful when you want to reuse types but don't need all fields.

**🔹 4. What is type assertion? How is it different from type casting?**

* **Type assertion**: You’re telling TypeScript: “Trust me, I know better.”

const input = document.getElementById('name') as HTMLInputElement;

* **Type casting**: In other languages like Java/C, this means converting values. In TypeScript, **type assertion doesn't change the actual value**, it just tells TS how to treat it.

**🔹 5. What is the difference between an interface and a type? When would you use each?**

* Both define shapes of data (like blueprints).

| **Interface** | **Type** |
| --- | --- |
| Extendable | More flexible |
| Best for objects | Can be unions, primitives, etc. |

**Use interface**: When defining structure of objects/classes.  
**Use type**: When you want unions, combinations, or primitives + objects.

**🔹 6. How do you enforce type safety in API responses in your frontend?**

* Create a Type or Interface for the expected API data.
* Use it with Axios/fetch like this:

interface UserData { name: string; email: string }

axios.get<UserData[]>('/api/users');

🔐 This ensures: if backend sends wrong data → your code will throw error during dev time.

**🔹 7. How do you handle union types and type guards in complex components?**

type Value = string | number;

function print(val: Value) {

if (typeof val === 'string') {

console.log(val.toUpperCase()); // Safe to use string methods

} else {

console.log(val.toFixed(2)); // Safe to use number methods

}

}

* This technique (checking type before using it) is called **type guard**.
* Useful when working with APIs or forms where value type can vary.

**🔹 8. What are enums and what are their use cases in full-stack projects?**

* **Enum = a group of related constants**. Like user roles, status types, etc.

enum Role { Admin, User, Guest }

const userRole: Role = Role.Admin;

🧠 Use cases:

* API status codes
* User roles
* Dashboard filters
* Actions like START, STOP, RESTART

# 🟦 Frontend (React + TypeScript)

**🔷 1. How do you strongly type React props and state using TypeScript?**

🧠 Think of props like “input wires” to your component. State is internal memory.

interface Props {

name: string;

age: number;

}

const Welcome: React.FC<Props> = ({ name, age }) => {

return <h1>Hi {name}, you're {age} years old!</h1>;

};

🛡 TypeScript will throw an error if you pass wrong props (e.g., age="twenty").

**🔷 2. How do you type useRef, useState, and useReducer hooks?**

✅ **useRef**

const inputRef = useRef<HTMLInputElement>(null);

✅ **useState**

const [count, setCount] = useState<number>(0);

✅ **useReducer**

type Action = { type: 'increment' } | { type: 'decrement' };

const reducer = (state: number, action: Action) => {

switch (action.type) {

case 'increment': return state + 1;

case 'decrement': return state - 1;

}

};

const [state, dispatch] = useReducer(reducer, 0);

**🔷 3. What is the correct way to type event handlers in React (e.g., onClick, onChange)?**

👉 Don’t guess. Use the right types based on event target.

const handleClick = (e: React.MouseEvent<HTMLButtonElement>) => { ... };

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {

console.log(e.target.value);

};

✅ This helps TS suggest valid methods like .value, .checked, etc.

**🔷 4. How do you define reusable component props with generics?**

Imagine a List component that shows any type of items: strings, users, numbers.

interface ListProps<T> {

items: T[];

renderItem: (item: T) => JSX.Element;

}

function List<T>({ items, renderItem }: ListProps<T>) {

return <ul>{items.map(renderItem)}</ul>;

}

Use it like:

<List items={[1, 2, 3]} renderItem={(num) => <li>{num}</li>} />

🔁 Reusable for any data type.

**🔷 5. How do you structure and manage types in a large React project?**

🧱 Use a types/ or @types/ folder.

Example file structure:

src/

┣ components/

┃ ┗ Button.tsx

┣ types/

┃ ┗ user.d.ts

┣ pages/

┗ utils/

⛳ Separate types for User, Product, API, Forms, etc., helps maintain clarity and avoids duplicate code.

**🔷 6. How do you ensure type safety with fetch or axios API calls in React?**

Create a response interface and use it with Axios or Fetch:

interface User {

id: number;

name: string;

}

const fetchUsers = async () => {

const res = await axios.get<User[]>('/api/users');

return res.data;

};

🧠 This ensures you access only the correct fields (id, name). No more runtime surprises!

**🔷 7. How do you use React.FC vs function components with explicit prop types?**

✅ **React.FC** (React Function Component)

const MyComponent: React.FC<Props> = ({ title }) => <h1>{title}</h1>;

🚫 Downside: children is added by default (even if you don’t use it).

✅ **Manual typing (preferred in large apps)**

function MyComponent({ title }: Props): JSX.Element {

return <h1>{title}</h1>;

}

👉 Better control over props.

**🔷 8. How do you integrate TypeScript with Form Libraries (Formik, React Hook Form)?**

✅ **Formik**

interface FormValues {

name: string;

age: number;

}

<Formik<FormValues>

initialValues={{ name: '', age: 0 }}

onSubmit={(values) => console.log(values)}

>

...

</Formik>

✅ **React Hook Form**

interface FormData {

email: string;

}

const { register, handleSubmit } = useForm<FormData>();

📌 Tip: Always define your form schema as interface and pass it as a generic.

# 🟥 Backend (Node.js + Express + TypeScript)

**🔷 1. How do you type an Express request, response, and middleware?**

When using Express with TypeScript, always import types like Request, Response, NextFunction.

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

const getUser = (req: Request, res: Response, next: NextFunction) => {

res.send('User data');

};

You can also do:

(req: Request<{ id: string }>, res: Response)

This way, you're telling TypeScript what to expect in req.params, req.body, req.query, etc.

**🔷 2. How do you structure a REST API project using TypeScript?**

Here’s a clean structure:

src/

┣ controllers/

┃ ┗ user.controller.ts

┣ routes/

┃ ┗ user.routes.ts

┣ models/

┃ ┗ user.model.ts

┣ middlewares/

┣ services/

┣ utils/

┣ types/

┣ index.ts (entry point)

Keep your code modular: separate logic into services, use interfaces from types/, and organize like a real-world product.

**🔷 3. How do you define models and DTOs in a TypeScript backend?**

* **Model** → Defines MongoDB structure (Mongoose schema).
* **DTO (Data Transfer Object)** → Defines what data is expected in requests/responses.

// DTO

export interface CreateUserDTO {

name: string;

email: string;

}

// Mongoose model

const UserSchema = new Schema({

name: String,

email: String,

});

This helps you control what goes into and comes out of the API.

**🔷 4. What is zod or joi, and how do you validate and type check incoming data?**

🛡 These libraries **validate and parse request data**.

* **zod** (modern + TypeScript-first)

import { z } from 'zod';

const schema = z.object({

name: z.string(),

age: z.number().min(1),

});

const parsed = schema.parse(req.body); // throws error if invalid

* **joi** (older but very powerful)

const schema = Joi.object({ name: Joi.string().required() });

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

✅ Use it before processing user input to avoid bad/malicious data.

**🔷 5. How do you use TypeScript with MongoDB and Mongoose?**

✅ Step-by-step:

1. Define a TypeScript interface.
2. Pass it to your Mongoose schema.
3. Use the typed model in your code.

interface IUser {

name: string;

email: string;

}

const UserSchema = new Schema<IUser>({ name: String, email: String });

const UserModel = model<IUser>('User', UserSchema);

Now you get full IntelliSense and type safety on UserModel.find(), .save(), etc.

**🔷 6. How do you handle error types and custom error classes in an Express app?**

Create a custom class like this:

class AppError extends Error {

statusCode: number;

constructor(message: string, statusCode: number) {

super(message);

this.statusCode = statusCode;

}

}

Then, use it like:

throw new AppError('User not found', 404);

💡 Catch it in your error middleware:

app.use((err: AppError, req: Request, res: Response) => {

res.status(err.statusCode || 500).json({ message: err.message });

});

**🔷 7. How do you manage and type environment variables safely in a TypeScript backend?**

Use dotenv and create a global typing:

// env.d.ts

declare namespace NodeJS {

interface ProcessEnv {

PORT: string;

MONGO\_URI: string;

}

}

Then use with confidence:

process.env.PORT; // now it's strongly typed

Also validate them on startup using zod if needed.

**🔷 8. How do you enforce consistent response structure using TypeScript interfaces?**

Define a generic response structure:

interface ApiResponse<T> {

success: boolean;

data?: T;

message?: string;

}

Use it like:

res.json<ApiResponse<User>>({

success: true,

data: user,

});

🔁 Keeps frontend and backend in sync and predictable.

# 🟩 Full Stack / Architecture

**🔷 1. How do you share types between frontend and backend?**

📦 Best way: **create a shared package** or folder (especially in a monorepo).

root/

┣ packages/

┃ ┗ shared-types/ ← contains .ts files with interfaces

┣ frontend/

┣ backend/

Then import like this:

import { UserDTO } from 'shared-types';

✅ Keeps contracts in sync.  
❌ Don’t duplicate interfaces — causes bugs and maintenance pain.

**🔷 2. How do you type API contracts in a monorepo setup or via Swagger/OpenAPI?**

There are two popular ways:

✅ **Option 1: Manual type sharing**

* Use shared-types package (see above)

✅ **Option 2: Auto-generate types from OpenAPI**

* Define API in Swagger (.yaml)
* Use [openapi-typescript](https://github.com/drwpow/openapi-typescript)

npx openapi-typescript api.yaml --output types/api.d.ts

You’ll get all types auto-generated and synced.

**🔷 3. What are the pros/cons of using ts-node in production?**

ts-node lets you run .ts files directly without compiling. But…

| **✅ Pros** | **❌ Cons** |
| --- | --- |
| Fast for dev/debugging | Slow in prod |
| No build step | Consumes more memory |
| Good for scripts/CLI | Lacks optimization |

🚫 **Don’t use in production**. Instead:

tsc

node dist/index.js

**🔷 4. How do you manage absolute imports and module resolution in a TypeScript project?**

🎯 Instead of writing:

import Header from '../../../components/Header';

Use absolute path like:

import Header from 'components/Header';

✅ How to set it up:

In tsconfig.json:

{

"compilerOptions": {

"baseUrl": "./src",

"paths": {

"\*": ["\*"]

}

}

}

For React (Vite, CRA, or Next.js), also check vite.config.js or jsconfig.json.

**🔷 5. What is a good directory structure for a scalable MERN app with TypeScript?**

✅ Split by features and responsibilities:

backend/

┣ controllers/

┣ routes/

┣ models/

┣ services/

┣ middlewares/

┣ utils/

┣ types/

┗ index.ts

frontend/

┣ components/

┣ pages/

┣ services/

┣ hooks/

┣ store/

┣ types/

┗ main.tsx

👍 Benefits:

* Easy to scale
* Separation of concerns
* Clean folder layout

**🔷 6. How do you ensure type safety across microservices or APIs using tools like tRPC or GraphQL?**

🛠 Tools that help with shared types:

✅ **tRPC** – auto type-safe APIs (no need for Swagger or REST docs).

* You define backend routes like functions
* Frontend auto gets return type

✅ **GraphQL + codegen** – auto-generate TS types from your schema and queries.

graphql-codegen init

⚙ Setup auto-syncs types:

* Request types
* Response types
* Error handling

📌 These tools help **avoid mismatches** and give **autocomplete** on frontend calls.

# 🟩 Testing & Tooling

**🔷 1. How do you write unit tests with Jest in a TypeScript codebase?**

✅ Setup:

* Install jest, ts-jest, and @types/jest:

npm install --save-dev jest ts-jest @types/jest

* In jest.config.js:

module.exports = {

preset: 'ts-jest',

testEnvironment: 'node',

};

✅ Example test:

// add.ts

export const add = (a: number, b: number): number => a + b;

// add.test.ts

import { add } from './add';

test('adds 2 + 3 = 5', () => {

expect(add(2, 3)).toBe(5);

});

📌 TypeScript ensures add() is only called with numbers, even in tests.

**🔷 2. How do you mock and type services and modules in tests?**

✅ Example: mocking a service in a controller test

// userService.ts

export const getUser = () => ({ name: 'Pramod' });

// userController.test.ts

import \* as userService from './userService';

jest.mock('./userService');

const mockedGetUser = userService.getUser as jest.Mock;

mockedGetUser.mockReturnValue({ name: 'Test User' });

test('controller returns mocked user', () => {

const result = userService.getUser();

expect(result.name).toBe('Test User');

});

🎯 Type assertion like as jest.Mock helps Jest understand and type your mocks correctly.

**🔷 3. What TypeScript configuration (tsconfig.json) optimizations do you use in real-world projects?**

📦 Recommended tsconfig.json:

{

"compilerOptions": {

"target": "ES2020",

"module": "CommonJS",

"strict": true,

"esModuleInterop": true,

"forceConsistentCasingInFileNames": true,

"skipLibCheck": true,

"outDir": "./dist",

"baseUrl": "./src",

"paths": {

"\*": ["\*"]

}

}

}

💡 Key Flags:

* "strict": true – enables strict typing across the project.
* "esModuleInterop" – lets you import both default & named exports safely.
* "forceConsistentCasingInFileNames" – avoids bugs on different OS.
* "skipLibCheck" – faster build, skips checking node\_modules.

**🔷 4. What are the benefits of strict typing in CI/CD pipelines and how do you enforce it?**

✅ Benefits:

* **Catch bugs before deploy** (TS will fail the build on type errors)
* **Improves collaboration** – no guessing function usage
* **Reliable backend/frontend sync** – especially when sharing types

✅ How to enforce:

1. Add this to your CI/CD pipeline (GitHub Actions, GitLab CI, etc.)

tsc --noEmit

1. Optional: Run type checking with linting

eslint --ext .ts,.tsx src/

1. Use a pre-push Git hook with husky:

npx husky add .husky/pre-push "tsc --noEmit"

🛡 Result: No one can push broken types — strong protection in teams.