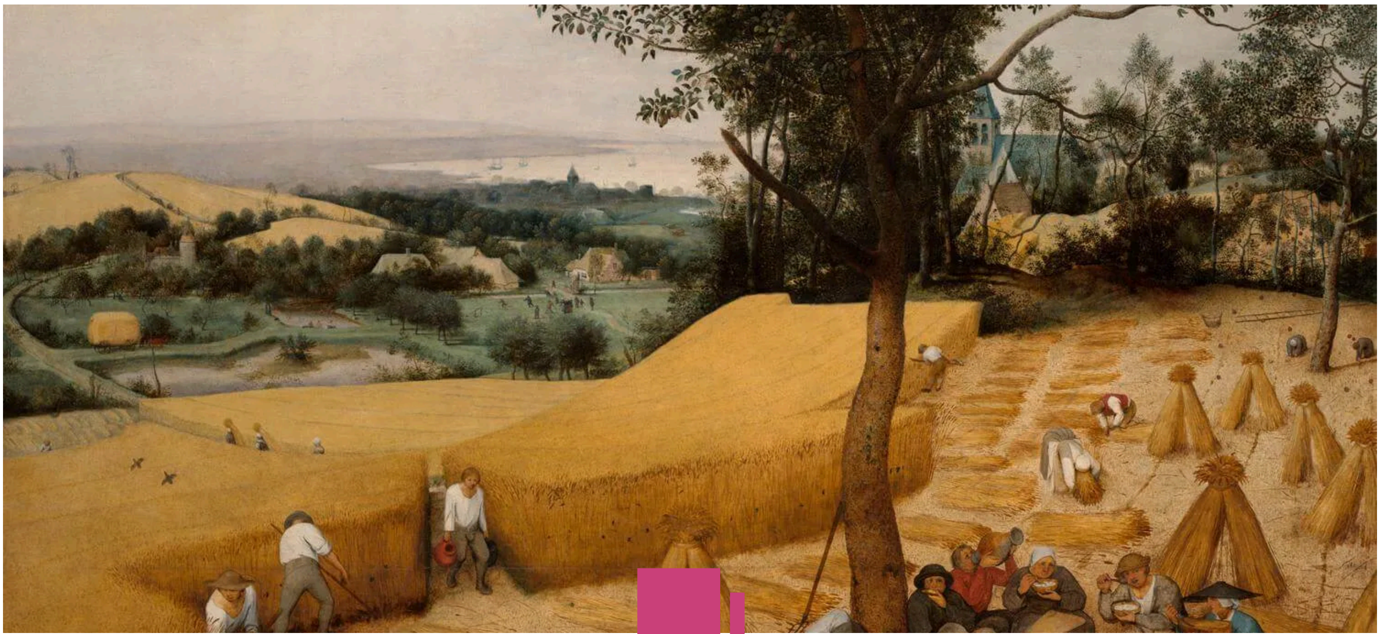
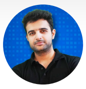


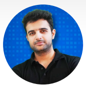
Week 12.2

Advance Typescript APIs

In today's lecture, Harkirat delves into advanced TypeScript utility types such as `Pick` , `Partial` , `Readonly` , `Record` , `Exclude` and the `Map` type, providing insights into their practical applications. Additionally, the lecture covered type inference in Zod, a TypeScript-first schema declaration and validation library, highlighting how these advanced features can enhance type safety and developer productivity in TypeScript projects.



Before diving into an advanced TypeScript API, it's important to have a solid understanding of the basics of TypeScript, especially when it comes to using it in a Node.js environment. Here's



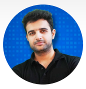
1. **Understand Basic TypeScript Classes:** Familiarity with how classes are defined and used in TypeScript, including constructors, properties, methods, and inheritance.
2. **Understand Interfaces and Types:** Know how to define and use interfaces and types to enforce the structure of objects and function parameters.
3. **Experience with TypeScript in Node.js:** Have experience setting up a simple Node.js application with TypeScript and understand how to run and compile TypeScript code.

The following code snippet is a test to check your understanding:

```
interface User {
    name: string;
    age: number;
}

function sumOfAge(user1: User, user2: User) {
  return user1.age + user2.age;
};

// Example usage
const result = sumOfAge({
    name: "harkirat",
    age: 20
}, {
    name: "raman",
    age: 21
});
console.log(result); // Output: 41
```

In this example, you should understand the following concepts:

- **Interface** `User` : Defines the structure for a user object with `name` and `age` properties.
- **Function** `sumOfAge` : Takes two `User` objects as parameters and returns the sum of their ages.
- **Example Usage:** Demonstrates how to call `sumOfAge` with two user objects and logs the



Recap Setup Procedure

To start a TypeScript project locally, follow these steps:

1. **Initialize TypeScript:**
   Run `npx tsc --init` in your project directory to create a `tsconfig.json` file, which is the configuration file for TypeScript.
2. **Configure `tsconfig.json` :**
   Edit the `tsconfig.json` file to specify the root directory and the output directory for the compiled JavaScript files.

```
{
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist",
    // ... other options
  }
}
```

- `"rootDir": "./src"` : Tells TypeScript to look for `.ts` files in the `src` directory.
- `"outDir": "./dist"` : Compiled `.js` files will be output to the `dist` directory.

1] Pick

The `Pick` utility type in TypeScript is a powerful feature that allows you to construct new types by selecting a subset of properties from an existing type. This can be particularly useful when you need to work with only certain fields of a complex type, enhancing type safety and code readability without redundancy.

Understanding `Pick`

- `Type` : The original type you want to pick properties from.
- `Keys` : The keys (property names) you want to pick from the `Type` , separated by `|` (the union operator).

Example Usage of `Pick`

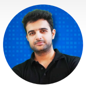
Consider an interface `User` that represents a user in your application:

```
interface User {
  id: number;
  name: string;
  email: string;
  createdAt: Date;
}
```

Suppose you're creating a function to display a user profile, but you only need the `name` and `email` properties for this purpose. You can use `Pick` to create a new type, `UserProfile` , that includes only these properties:

```
// Creating a new type with only `name` and `email` properties
type UserProfile = Pick<User, 'name' | 'email'>;

// Function that accepts a UserProfile type
const displayUserProfile = (user: UserProfile) => {
  console.log(`Name: ${user.name}, Email: ${user.email}`);
};
```

In this example, `UserProfile` is a new type that has only the `name` and `email` properties from the original `User` interface. The `displayUserProfile` function then uses this `UserProfile` type for its parameter, ensuring that it can only receive objects that have `name` and `email` properties.



2. **Code Readability:** Using `Pick` to create descriptive types can make your code more readable and self-documenting.
3. **Reduced Redundancy:** Instead of defining new interfaces manually for subsets of properties, `Pick` allows you to reuse existing types, keeping your code DRY (Don't Repeat Yourself).

> The **`Pick`** utility type in TypeScript allows you to create types that are subsets of existing types. It allows you to be explicit about what properties a function or component expects, leading to more maintainable and error-resistant code.

2] Partial

The `Partial` utility type in TypeScript is used to create a new type by making all properties of an existing type optional. This is particularly useful when you want to update a subset of an object's properties without needing to provide the entire object.

Understanding **`Partial`**

The `Partial` utility type takes a single type argument and produces a type with all the properties of the provided type set to optional. Here's the syntax for using `Partial` :

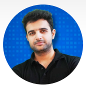
```
Partial<Type>
```

- `Type` : The original type you want to convert to a type with optional properties.

Example Usage of **`Partial`**

Let's say you have a `User` interface representing a user in your application:

```
interface User {
```



If you're creating a function to update a user, you might only want to update their `name`, `age`, or `email`, and not all properties at once. You can use `Pick` to select these properties and then apply `Partial` to make them optional:

```
// Selecting 'name', 'age', and 'email' properties from User
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>

// Making the selected properties optional
type UpdatePropsOptional = Partial<UpdateProps>

// Function that accepts an object with optional 'name', 'age',
function updateUser(updatedProps: UpdatePropsOptional) {
    // hit the database to update the user
}

// Example usage of updateUser
updateUser({ name: "Alice" }); // Only updating the name
updateUser({ age: "30", email: "alice@example.com" }); // Updat
updateUser({}); // No updates, but still a valid call
```

In this example, `UpdatePropsOptional` is a new type where the `name`, `age`, and `email` properties are all optional, thanks to `Partial`. The `updateUser` function can then accept an object with any combination of these properties, including an empty object.

Benefits of Using `Partial`

1. **Flexibility in Updates:** `Partial` is ideal for update operations where you may only want to modify a few properties of an object.
2. **Type Safety:** Even though the properties are optional, you still get the benefits of type checking for the properties that are provided.
3. **Code Simplicity:** Using `Partial` can simplify function signatures by not requiring clients to pass an entire object when only a part of it is needed.

> allows you to create types that are more flexible for update operations while still maintaining type safety.

3] Readonly

The `Readonly` utility type in TypeScript is used to make all properties of a given type read-only. This means that once an object of this type is created, its properties cannot be reassigned. It's particularly useful for defining configuration objects, constants, or any other data structure that should not be modified after initialization.

Understanding `Readonly`

The `Readonly` utility type takes a type `T` and returns a type with all properties of `T` set as read-only. Here's the basic syntax:

```
Readonly<Type>
```

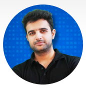
- `Type` : The original type you want to convert to a read-only version.

Example Usage of `Readonly`

Consider an interface `Config` that represents configuration settings for an application:

```
interface Config {
  endpoint: string;
  apiKey: string;
}
```

To ensure that a `Config` object cannot be modified after it's created, you can use the `Readonly` utility type:



```
// Attempting to modify the object will result in a TypeScript
// config.apiKey = 'newkey'; // Error: Cannot assign to 'apiKey
```

In this example, `config` is an object that cannot be modified after its initialization. Trying to reassign `config.apiKey` will result in a compile-time error, ensuring the immutability of the `config` object.

Benefits of Using `Readonly`

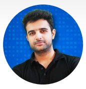
1. **Immutability**: Ensures that objects are immutable after they are created, preventing accidental modifications.
2. **Compile-Time Checking**: The immutability is enforced at compile time, catching potential errors early in the development process.
3. **Clarity and Intent**: Using `Readonly` clearly communicates the intent that an object should not be modified, making the code easier to understand.

Important Note

It's crucial to remember that the `Readonly` utility type enforces immutability at the TypeScript level, which means it's a compile-time feature. JavaScript, which is the output of TypeScript compilation, does not have built-in immutability, so the `Readonly` constraint does not exist at runtime. This distinction is important for understanding the limitations of `Readonly` and recognizing that it's a tool for improving code quality and safety during development.

> The `Readonly` utility type is a valuable feature in TypeScript for creating immutable objects. By preventing reassignment of properties, it helps maintain the integrity of objects that represent fixed configurations or constants.

4] Record & Map



the requirements of your application.

Record

The `Record<K, T>` utility type is used to construct a type with a set of properties `K` of a given type `T`. It provides a cleaner and more concise syntax for typing objects when you know the shape of the values but not the keys in advance.

Example Using `Record`

```
interface User {
  id: string;
  name: string;
}

// Using Record to type an object with string keys and User val
type Users = Record<string, User>;

const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};

console.log(users['abc123']); // Output: { id: 'abc123', name:
```

In this example, `Users` is a type that represents an object with any string as a key and `User` objects as values. The `Record` utility type simplifies the declaration of such structures, making your code more readable and maintainable.

Map

The `Map` object in TypeScript (inherited from JavaScript) represents a collection of key-value pairs where both the keys and values can be of any type. Maps remember the original insertion order of the keys, which is a significant difference from plain JavaScript objects.

```
  name: string;
}

// Initialize an empty Map with string keys and User values
const usersMap = new Map<string, User>();

// Add users to the map using .set
usersMap.set('abc123', { id: 'abc123', name: 'John Doe' });
usersMap.set('xyz789', { id: 'xyz789', name: 'Jane Doe' });

// Accessing a value using .get
console.log(usersMap.get('abc123')); // Output: { id: 'abc123',
```

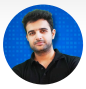
In this example, `usersMap` is a `Map` object that stores `User` objects with string keys. The `Map` provides methods like `.set` to add key-value pairs and `.get` to retrieve values by key. Maps are particularly useful when you need to maintain the order of elements, perform frequent additions and deletions, or use non-string keys.

Record vs. Map

- **Use `Record` when**: You are working with objects that have a fixed shape for values and string keys. It's ideal for typing object literals with known value types.
- **Use `Map` when**: You need more flexibility with keys (not just strings or numbers), or you need to maintain the insertion order of your keys. Maps also provide better performance for large sets of data, especially when frequently adding and removing key-value pairs.

> Both **`Record`** and **`Map`** enhance TypeScript's ability to work with collections of data in a type-safe manner, each offering unique benefits suited to different scenarios in application development.

5] Exclude



create a type that is a subset of another type, with some elements removed.

Understanding `Exclude`

The `Exclude<T, U>` utility type takes two arguments:

- `T` : The original union type from which you want to exclude some members.
- `U` : The union type containing the members you want to exclude from `T` .

The result is a type that includes all members of `T` that are not assignable to `U` .

Example Using `Exclude`

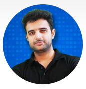
Let's say you have a union type `Event` that represents different types of events in your application:

```
type Event = 'click' | 'scroll' | 'mousemove';
```

If you have a function that should handle all events except for `scroll` events, you can use `Exclude` to create a new type that omits `scroll` :

```
// Using Exclude to create a new type without 'scroll'
type ExcludeEvent = Exclude<Event, 'scroll'>; // 'click' | 'mou

// Function that accepts only 'click' and 'mousemove' events
const handleEvent = (event: ExcludeEvent) => {
  console.log(`Handling event: ${event}`);
};

handleEvent('click'); // OK
handleEvent('scroll'); // Error: Argument of type '"scroll"' is
```

In this example, `ExcludeEvent` is a new type that includes only `'click'` and



variables, preventing unwanted types from being used.

2. **Code Readability**: Using `Exclude` can make your type intentions clearer to other developers, as it explicitly shows which types are not allowed.
3. **Utility**: It's a built-in utility type that saves you from having to manually construct new types, making your code more concise and maintainable.

> The `Exclude` utility type in TypeScript allows to create types that exclude certain members from a union. It allows you to refine type definitions for specific use cases, enhancing type safety and clarity in your code.

6] Type Inferences In Zod

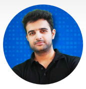
Type inference in Zod is a powerful feature that allows TypeScript to automatically determine the type of data validated by a Zod schema. This capability is particularly useful in applications where runtime validation coincides with compile-time type safety, ensuring that your code not only runs correctly but is also correctly typed according to your Zod schemas.

How Type Inference Works in Zod

Zod schemas define the shape and constraints of your data at runtime. When you use Zod with TypeScript, you can leverage Zod's type inference to automatically generate TypeScript types based on your Zod schemas. This means you don't have to manually define TypeScript interfaces or types that replicate your Zod schema definitions, reducing redundancy and potential for error.

Example: Type Inference with Zod in an Express App

Consider an Express application where you want to validate and update a user's profile information. You define a Zod schema for the profile update request body:

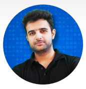


```
// Define the schema for profile update
const userProfileSchema = z.object({
  name: z.string().min(1, { message: "Name cannot be empty" }),
  email: z.string().email({ message: "Invalid email format" }),
  age: z.number().min(18, { message: "You must be at least 18 y
});

app.put("/user", (req, res) => {
  const result = userProfileSchema.safeParse(req.body);

  if (!result.success) {
    res.status(400).json({ error: result.error });
    return;
  }

  // Type of updateBody is inferred from userProfileSchema
  const updateBody = result.data;

  // update database here
  res.json({
    message: "User updated",
    updateBody
  });
});

app.listen(3000, () => console.log("Server running on port 3000
```

In this example, `userProfileSchema.safeParse(req.body)` validates the request body against the `userProfileSchema` . The `safeParse` method returns an object that includes a `success` boolean and, on success, a `data` property containing the validated data.

Assigning a Type to `updateBody`

Thanks to Zod's type inference, the type of `updateBody` is automatically inferred to be:



```
  email: string;
  age?: number;
}
```

This inferred type is derived directly from the `userProfileSchema` definition. If you try to access a property on `updateBody` that isn't defined in the schema, TypeScript will raise a compile-time error, providing an additional layer of type safety.

Benefits of Type Inference in Zod

1. **Reduced Boilerplate**: You don't need to manually define TypeScript types that mirror your Zod schemas.
2. **Type Safety**: Ensures that your data conforms to the specified schema both at runtime (through validation) and at compile-time (through type checking).
3. **Developer Productivity**: Type inference, combined with Zod's expressive API for defining schemas, makes it easier to write, read, and maintain your validation logic and related type definitions.

> Type inference in Zod bridges the gap between runtime validation and compile-time type safety in TypeScript applications. By automatically generating TypeScript types from Zod schemas, Zod helps ensure that your data validation logic is both correct and type-safe, enhancing the reliability and maintainability of your code.

1 Comment

Most upvotes All comments

Add a comment...

word as he points out things in beginner friendly way!! just a suggestion

10 15 3 Replies