Advantages of Using InversifyJS in Node.js

Exploring the benefits of using InversifyJS for Dependency Injection and Inversion of Control in Node.js

Author: Denys Sorokin

Agenda

- Introduction to Inversion of Control
- What is Dependency Injection (DI)?
- Why use InversifyJS?
- Advantages of InversifyJS
- Code Example

Introduction to Inversion of Control (IoC)

- IoC is a design principle that inverts the control flow of a program. Rather than the program controlling its flow, external factors dictate the behavior of modules

- It is widely used to manage dependencies between components, particularly in large applications

What is Dependency Injection (DI)?

- Dependency Injection is a pattern that promotes loose coupling, making modules more modular and easier to maintain.
- Instead of creating dependencies inside the class, DI allows passing dependencies from the outside.

Why use InversifyJS?

- InversifyJS is a lightweight IoC container for JavaScript/TypeScript applications
- It helps to implement DI with decorators like @injectable and @inject
- InversifyJS encourages better architecture and easier management of dependencies in Node.js

Advantages of InversifyJS

- Decouples Code and Improves Modularity The IoC container manages dependencies, making it easier to changeimplementations without modifying the core logic
- Enhances Testability Using InversifyJS, dependencies can be mocked, making testing individual modules morestraightforward
- Increases Flexibility and Maintainability As the application grows, adding or swapping dependencies becomesseamless

Code example: Minimal package list to use InversifyJS

```
31    "devDependencies": {
32      "@types/node": "^22.7.3",
33      "ts-node": "^10.9.2",
34      "typescript": "^5.6.2"
35    },
36    "dependencies": {
37      "inversify": "^6.0.2",
38      "reflect-metadata": "^0.2.2"
39    }
40  }
41
```

Code example: Step 1. Create interfaces

```typescript
export interface Logger {
  info(msg: string, data: object): void;
  warn(msg: string, data: object): void;
  error(msg: string, data: object): void;
}

You, yesterday | 1 author (You)
export interface ILoggerList {
  conf: Logger;
  public: Logger;
}

You, yesterday | 1 author (You)
export interface IMongoConfig {
  connectionString: string;
}

You, yesterday | 1 author (You)
export interface IGlobalConfig {
  propertyA: string;
  propertyB: string;
  propertyC: string;
}
```

```typescript
export abstract class IMongoLib {
  abstract readonly logger: ILoggerList;
  abstract readonly config: IMongoConfig;

  abstract connect(): Promise<void>;

  abstract disconnect(): Promise<void>;

  abstract find<T>(): Promise<T>;

  abstract update<T>(): Promise<T | null>;

  abstract save<T>(): Promise<T | null>;

  abstract delete(): Promise<number>;
}

You, yesterday | 1 author (You)
export abstract class IApp {
  abstract readonly config: IGlobalConfig;
  abstract readonly logger: ILoggerList;
  abstract readonly mongoLib: IMongoLib;

  abstract sayHello(): void;
}
```

Code example: Step 2. Add types

```
1   export const TYPES = {
2     IMongoConfig: Symbol.for('IMongoConfig'),
3     IMongoLib: Symbol.for('IMongoLib'),
4     IGlobalConfig: Symbol.for('IGlobalConfig'),
5     IApp: Symbol.for('IApp'),
6     ILoggerList: Symbol.for('ILoggerList'),
7     Logger: Symbol.for('Logger'),
8     IConfidentialLogger: Symbol.for('IConfidentialLogger'),
9     IPublicLogger: Symbol.for('IPublicLogger'),
10  }
```

InversifyJS needs to use the type as identifiers at runtime

Code example: Step 3. Initialize components

```typescript
import { inject, injectable } from 'inversify';
import { IGlobalConfig, IApp, ILoggerList, IMongoLib } from '../interfaces';
import { TYPES } from '../types';

You, yesterday | 1 author (You)
@injectable()
export class App implements IApp {
  constructor(
    @inject(TYPES.IGlobalConfig) public readonly config: IGlobalConfig,
    @inject(TYPES.ILoggerList) public readonly logger: ILoggerList,
    @inject(TYPES.IMongoLib) public readonly mongoLib: IMongoLib,
  ) {}

  async sayHello(): Promise<void> {
    console.log('Hello InversifyJS!');
    this.logger.conf.info('show config from App: ', this.config);
    await this.mongoLib.connect();
    await this.mongoLib.disconnect();
  }
}
```

Code example: Step 4. Create container with components

```ts
import 'reflect-metadata';
import { Container } from 'inversify';
import type { IApp, IGlobalConfig, IMongoConfig, IMongoLib, Logger, ILoggerList } from
import { TYPES } from './types';
import { MongoConfig } from './components/MongoConfig';
import { MongoLib } from './components/MongoLib';
import { GlobalConfig } from './components/GlobalConfig';
import { App } from './components/App';
import { LoggerPublic } from './components/Logger.public';
import { LoggerConfidential } from './components/Logger.confidential';
import { LoggerList } from './components/LoggerList';

export const container = new Container();

container.bind<IMongoConfig>(TYPES.IMongoConfig).to(MongoConfig);
container.bind<IMongoLib>(TYPES.IMongoLib).to(MongoLib);
container.bind<IGlobalConfig>(TYPES.IGlobalConfig).to(GlobalConfig);
container.bind<IApp>(TYPES.IApp).to(App);
container.bind<Logger>(TYPES.IConfidentialLogger).to(LoggerConfidential);
container.bind<Logger>(TYPES.IPublicLogger).to(LoggerPublic);
container.bind<ILoggerList>(TYPES.ILoggerList).to(LoggerList);
```

Code example: Step 5. Run application

```ts
import { IApp } from './src/interfaces';
import { container } from './src/inversify.config';
import { TYPES } from './src/types';

const app = container.get<IApp>(TYPES.IApp);

app.sayHello();
```


```
Hello InversifyJS!
LoggerConfidential (INFO):  show config from App:  {"propertyA":"secret_property
_a","propertyB":"secret_property_b","propertyC":"secret_property_c"}
LoggerPublic (INFO):  Connected to mongodb!
 {"connectionString":"this_is_connection_to_mongodb"}
LoggerPublic (INFO):  Disconnected mongodb!
 {}
```

Thank you

Denys Sorokin