DOI: https://doi.org/10.26642/ten-2023-1(91)-166-171
UDK 681.3.06

S.I. Ronskyi, post-graduate student
Zhytomyr Polytechnic State University

Practical examples of using AsyncLocalStorage in NodeJS corporate web applications

(Presented by Ph.D. in Engineering, Associate Professor Kubrak Yu.O.)

Based on the results of a study of the popularity of programming languages conducted by the DOU community (dou.ua in the overall rating for 2022, the JavaScript programming language took the first place (18,8 %), and the TypeScript language – the sixth (10,4 %). If we consider the popularity of languages in the context of backend development, TypeScript took the seventh place (3,3 %), JavaScript – the eighth (3 %) [1]. Since these languages allow you to develop both the backend and frontend parts of an application, the popularity of these languages and the number of applications written in them is growing. The main environment for executing backend code in these languages is the NodeJS platform [2].

The NodeJS environment is actively developing and supplemented with new modules. The NodeJS module «async_hooks» was introduced in Version 16. Today it has the «Experimental» status. But one of its parts, namely «Asynchronous context tracking», has become more widely used, has been placed in a separate section of the documentation and has the status «Stable».

Article that provides practical usage examples AsyncLocalStorage, will be a valuable resource for developers and researchers who want to improve their skills in NodeJS and use it effectively to develop enterprise web applications.

Keywords: *corporate applications; web applications; journaling; multitenancy; JavaScript; NodeJS.*

Analysis of recent research and publications. An analysis of recent research shows considerable interest in NodeJS development, including enterprise application development. Aishna Gupta, Anuska Rakshit, Mansi Raturi, Nishant Raj, Pallavi Mishra study the development of an application for bibliophiles using NodeJS and MongoDB [3]. Nidhi Daulat, Mihir Chheda, Mishkat Shaikh, Sarvesh Sharma, Theres Bemila, Ankita Awsarmal explore the development of a travel collaboration application using the MERN technology stack [4]. Krutika Desai and Jinan Fiaidhi consider developing a social network using the MERN technology stack [5].

The purpose of the article is a demonstration of the possibilities of practical use of the async_hooks module, namely the class *AsyncLocalStorage* in corporate web applications to simplify the transfer of auxiliary data that does not affect the business logic layer.

Presentation of the main material. NodeJS uses an asynchronous programming model, where many operations are performed in parallel, without waiting for previous operations to complete. Module *node:async_hooks* provides an API for tracking asynchronous resources and is in experimental status. *AsyncLocalStorage* allows you to store certain data and access it during the entire execution time of an asynchronous chain of operations. Class *AsyncLocalStorage* is a part of the module *node:async_hooks*, but it is stable [2]. In the context of web applications, we can store and retrieve certain data during the execution of a web request.

In this article, we will investigate the following practical use cases of *AsyncLocalStorage*:

– passing traceId for logging;
– passing tenantId between application layers.

First, let's look at the application architecture for our example. In corporate web applications with complex business logic, it is advisable to use elements of pure or hexagonal architecture. We will use the layer dependency diagram shown at Fig.1.

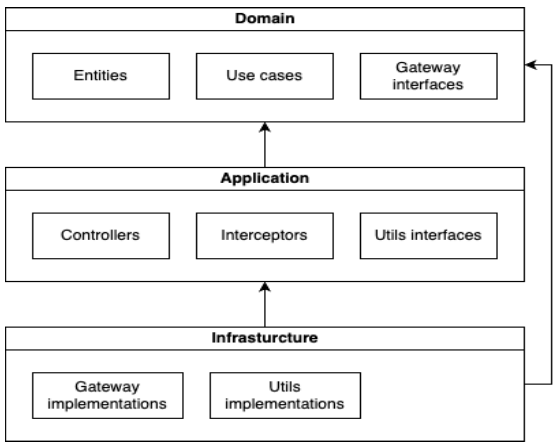

Fig. 1. Diagram of application layer dependencies

We have obtained a multi-layered architecture with inversion of dependencies relative to the program execution flow, which corresponds to a «pure architecture» [7]. We will use the TypeScript language, since JavaScript does not have interfaces, without which the principle of dependency inversion is not obvious. To simplify implementation, we will use the NestJS framework in the examples. NestJS is a framework for building scalable and modular server applications based on NodeJS and TypeScript. This framework has a built-in DI container. If you use other frameworks (Express, Fastify, Koa), you can use other libraries, such as TypeDI or Inversify.

First, let's look at the basic usage of *AsyncLocalStorage* from the NodeJS documentation. We create a sample of the class *AsyncLocalStorage* and wrap the asynchronous code call in its *run* method. This method takes as the first argument the initial state of our repository and callback:

```
const asyncLocalStorage = new AsyncLocalStorage();

function logWithId(msg) {
  const id = asyncLocalStorage.getStore();
  console.log(`${id !== undefined ? id : '-'}:`, msg);
}

let idSeq = 0;
http.createServer((req, res) => {
  asyncLocalStorage.run(idSeq++, () => {
    logWithId('start');
    // ... async code
  });
}).listen(8080);
```

Now let's look at initialization in our application. We create a module with a sample *AsyncLocalStorage*, that will act as a singleton:

```
const asyncLocalStorage = new AsyncLocalStorage<Map<string, string>>();
export default asyncLocalStorage;
```

Let's create middleware to wrap the call context.

```
export const asyncLocalStorageMiddleware = (
  req: Request,
  res: Response,
  next: NextFunction,
) => {
  const store = new Map<string, string>();
  asyncLocalStorage.run(store, () => {
    next();
  });
};
```

We connect our middleware globally at the time of initialization of the NestJS web server. In this case, the activation procedure is important. This middleware must be connected to other parts of the code that will use the asynchronous call context.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, { // configs });
  // this middleware is required for asyncLocalStorage to work
  app.use(asyncLocalStorageMiddleware);
  // ... other global middlewares, interceptors, etc
  await app.listen(3000);
}
bootstrap();
```

To access the asynchronous call context, we will create an abstraction and put it in our DI container:

```
import { Injectable, Logger } from '@nestjs/common';
import asyncLocalStorage from '~/@core/async-local-storage';

@Injectable()
export class AsyncLocalStorageService {
  public get(key: string): string {
    const store = asyncLocalStorage.getStore();
    return store.get(key);
  }

  public set(key: string, value: string): void {
    const store = asyncLocalStorage.getStore();
    store.set(key, value);
  }
}
```

Passing traceId for logging. When using microservices, utility, we can use the distributed tracing pattern [8]. In this case, we can get the traceId in the controller, but we will be interested in it in the Logger class and when calling another microservice. TraceId can also be useful for monolithic architectures. For example, for logging at the DEBUG and INFO levels, which are useful for development and debugging, you can get related query data objects and database query data.

Let’s create a global interceptor that will be responsible for working with traceId. We check the request header for traceId and create a new one if it is missing.

```
@Injectable()
export class RestLoggingInterceptor implements NestInterceptor {
  public static CONTEXT_ID_HEADER = 'X-Request-Id';
  public static TRACE_ID_KEY = 'X-Request-Id';

  private logger = new Logger(RestLoggingInterceptor.name);

  constructor(private readonly asyncLocalStorage: AsyncLocalStorageService) {}

  public intercept(
    context: ExecutionContext,
    next: CallHandler,
  ): Observable<unknown> {
    const request = context.switchToHttp().getRequest<Request>();
    const requestTraceId = request.header(
      RestLoggingInterceptor.CONTEXT_ID_HEADER,
    );
    const traceId = requestTraceId ?? randomBytes(16).toString('hex');

    this.asyncLocalStorage.set(RestLoggingInterceptor.TRACE_ID_KEY, traceId);
    context
      .switchToHttp()
      .getResponse<Response>()
      .set(RestLoggingInterceptor.CONTEXT_ID_HEADER, traceId);

    // ... intercept REST request and return next()
  }
}
```

Now, using our AsyncLocalStorageService abstraction, we can get traceId in other layers of the application. For example, when logging database calls, calling functions of other microservices, or sending data to message broker.

Using AsyncLocalStorage to simplify multitenancy. In traditional enterprise client-server applications, which are typically used in a single organization, each client or organization has its own separate software sample. However, in order to reduce infrastructure costs and optimize resources, there is a need to be able to serve multiple clients with a single sample of the application. In addition, with the proliferation of web servers and browser technologies, more and more developers are choosing the «software as a service» distribution model, where applications are provided as a service over the Internet. The «software as a service» (SaaS) distribution model opens up a wide range of opportunities for developers and users and has the following advantages: reduced costs, flexibility and scalability, constant updates and support, and availability. Developers of software distributed under the SaaS model must take care to support the work of several separate customers. The property of software to serve several separate customers, the so-called tenants, by transparently allocating available resources is called multitenancy [9].

Let’s consider work with multitenancy in a web application. The code of any repository must know the ID of the current tenant. On the other hand, since data and business logic are isolated between tenants, the business logic layer may not depend on the tenant as such, so passing a parameter to this layer is redundant. One solution to the problem may be to create services in the «request scope» [3], but this method is not effective in terms of resource usage. Let’s consider the controller code:

```
@Controller('resource')
export class ResourceController {
  constructor(private readonly useCase: DomainLogicUseCase) {
  }

  @Post()
  public async createResource(
    @User() user: UserIdentity,
    @Body() dto: CreateResourcePayloadDto,
    @Tenant() tenant: TenantIdentity,
  ): Promise<CreateResourceResponseDto> {
    const input = dto.toInput();
    input.user = user;
    input.tenant = tenant;
    return await this.useCase.createResource(input);
  }
}
```

Then we consider the solution of this problem using *AsyncLocalStorage* [2]. *AsyncLocalStorage* allows storing and transmitting certain data in the context of asynchronous calls. As with traceId, we can create an interceptor [3] for the HTTP call context and connect it globally or separately for each controller. The Interceptor can operate with a sample of *AsyncLocalStorage* and set *tenantId* in it:

```
@Injectable()
export class TenantInterceptor implements NestInterceptor {
  private logger = new Logger(TenantInterceptor.name);

  constructor(
    private readonly asyncTenantService: AsyncTenantService,
    private tenantService: TenantService,
  ) {}

  async intercept(
    context: ExecutionContext,
    next: CallHandler,
  ): Promise<Observable<any>> {
    const host = context.switchToHttp().getRequest().host;
    const tenant = await this.tenantService.getByDomain(host);
    if (!tenant) {
      this.logger.error(`Tenant not found for host ${host}`);
      throw new NotFoundException();
    }
    this.asyncTenantService.setTenantId(tenant.id);
    return next.handle();
  }
}
```

Now we can not pass data about the current tenant to the business logic layer. In the infrastructure layer, we can work with *AsyncLocalStorage* and get *tenantId*from it:

```
@Injectable()
export class ContactRepository implements IContactGateway {
  constructor(
    @InjectRepository(ContactSchema)
    protected readonly repository: Repository<ContactSchema>,
    protected readonly tenantService: AsyncTenantService,
  ) {}

  public async getOneById(id: string): Promise<Contact | null> {
    const tenantId = this.tenantService.getTenantId();
    const entity = await this.repository.findOne({
      where: { id: id, tenant: { id: tenantId } },
    });
    return entity ? entity.toEntity() : null;
  }
}
```

In this way, we can simplify the interfaces for calling the business logic layer.

Conclusions. As a result of our research, we have considered practical examples of using *AsyncLocalStorage* to simplify the development of enterprise web applications in the TypeScript programming language. The module is useful for practical use in cases where certain information needs to be transferred between layers to different layers of the application, but such information does not directly affect the operation of business logic.

References:

1. *Шевченко Р.* Рейтинг мов програмування 2022 / *Р.Шевченко, І.Яновський*. – 2022 [Електронний ресурс]. – Режим доступу : https://dou.ua/lenta/articles/language-rating-2022.
2. NodeJS documentation [Electronic resource]. – Access mode : https://nodejs.org/en/docs.
3. A Web-based Book Application using MongoDB & Nodejs / *Aishna Gupta, Anuska Rakshit, Mansi Raturi and other* // International Research Journal of Engineering and Technology. – 2022. – Vol. 09, Issue 01 [Electronic resource]. – Access mode : https://www.researchgate.net/publication/357909376.
4. Collaborative Tourism Application / Nidhi Daulat, Mihir Chheda, Mishkat Shaikh and other // International Journal for Research in Applied Science & Engineering Technology. – 2023. – Vol. 11, Issue III [Electronic resource]. – Access mode : https://www.researchgate.net/publication/369668637.
5. *Krutika Desai* Developing a Social Platform using MERN Stack / *Krutika Desai, Dr. Jinan Fiaidhi*. – 2022 [Electronic resource]. – Access mode : https://www.researchgate.net/publication/366231687.
6. NestJS documentation [Electronic resource]. – Access mode : https://docs.nestjs.com.
7. *Мартін Р.* Чиста архітектура: Мистецтво розроблення програмного забезпечення / *Р.Мартін*. – Харків : Ранок, 2020. – 368 с.
8. *Richardson Chris* Microservices patterns / *Chris Richardson* // Manning. – 2018. – 520 с.
9. Defining Multi-Tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective / *J.Kabbedijk, C. Bezemer, S. Jansen, A. Zaidman* // Journal of Systems and Software 100. – October. – 2014 [Електронний ресурс]. – Режим доступу до ресурсу: https://www.researchgate.net/publication/267455810.

References:

1. Shevchenko, R. and Ianovskyi, I. (2022), *Reitynh mov prohramuvannia 2022*, [Online], available at: https://dou.ua/lenta/articles/language-rating-2022
2. *NodeJS documentation*, [Online], available at: https://nodejs.org/en/docs
3. Aishna Gupta, Anuska Rakshit, Mansi Raturi et al. (2022), «A Web-based Book Application using MongoDB & Nodejs», *International Research Journal of Engineering and Technology*, Vol. 09, Issue 01, [Online], available at: https://www.researchgate.net/publication/357909376
4. Nidhi Daulat, Mihir Chheda, Mishkat Shaikh et al. (2023), «Collaborative Tourism Application», *International Journal for Research in Applied Science & Engineering Technology*, Vol. 11, Issue III, [Online], available at: https://www.researchgate.net/publication/369668637
5. Krutika, Desai and Dr. Jinan Fiaidhi (2022), *Developing a Social Platform using MERN Stack*, [Online], available at: https://www.researchgate.net/publication/366231687
6. *NestJS documentation*, [Online], available at: https://docs.nestjs.com
7. Martin, R. (2020), *Chysta arkhitektura: Mystetstvo rozroblennia prohramnoho zabezpechennia*, Ranok, Kharkiv, 368 p.

8. Richardson, Chris (2018), «Microservices patterns», *Manning*, 520 p.
9. Kabbedijk, J., Bezemer, C., Jansen, S. and Zaidman, A. (2014), «Defining Multi-Tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective», *Journal of Systems and Software 100*, October, [Online], available at: https://www.researchgate.net/publication/267455810

Ronskyi Sviatoslav Igorovych – post-graduate student of the Zhytomyr Polytechnic State University.
https://orcid.org/0000-0002-5411-4113.
Research interests:
– software architecture;
– information systems and technologies.

Ронський С.І.

Практичні приклади використання AsyncLocalStorage в NodeJS корпоративних вебдодатках

Мови програмування JavaScript та TypeScript з кожним роком збільшують свою долю на ринку розробки додатків, у томі числі, в бекенд розробці. Середовище виконання NodeJS розширює свій API та надає все більше можливостей. У цій статті досліджується практичне використання AsyncLocalStorage в корпоративних вебдодатках на платформі Node.js. AsyncLocalStorage є потужним інструментом, який дозволяє зберігати та передавати контекстну інформацію між асинхронними операціями, що полегшує розробку складних додатків. Стаття розпочинається з короткого огляду поняття AsyncLocalStorage та розгляду потенційної архітектури для прикладу. Далі розглядається базове використання AsyncLocalStorage та досліджуємо практичні сценарії використання AsyncLocalStorage в корпоративному вебдодатку, а саме – журналювання та ідентифікацію тенанта. У кожному сценарії розглядається проблематика та надаються конкретні приклади коду, які демонструють, як ефективно використовувати AsyncLocalStorage. Нарешті, стаття закінчується висновками, в яких підкреслюється важливість використання AsyncLocalStorage для поліпшення продуктивності та підтримки великих корпоративних вебдодатків на Node.js. Ця стаття стане корисним ресурсом для розробників, які бажають розширити свої знання про використання AsyncLocalStorage та впроваджувати його в свої проєкти.

Ключові слова: корпоративні додатки; вебдодатки; журналювання; мультитенантність; JavaScript; NodeJS.

The article was sent to the editorial board on 28.04.2023.