Async Local Storage

`AsyncLocalStorage` is a Node.js API (based on the `async_hooks` API) that provides an alternative way of propagating local state through the application without the need to explicitly pass it as a function parameter. It is similar to a thread-local storage in other languages.

The main idea of Async Local Storage is that we can *wrap* some function call with the `AsyncLocalStorage#run` call. All code that is invoked within the wrapped call gets access to the same `store`, which will be unique to each call chain.

In the context of NestJS, that means if we can find a place within the request's lifecycle where we can wrap the rest of the request's code, we will be able to access and modify state visible only to that request, which may serve as an alternative to REQUEST-scoped providers and some of their limitations.

Alternatively, we can use ALS to propagate context for only a part of the system (for example the *transaction* object) without passing it around explicitly across services, which can increase isolation and encapsulation.

Custom implementation

NestJS itself does not provide any built-in abstraction for `AsyncLocalStorage`, so let's walk through how we could implement it ourselves for the simplest HTTP case to get a better understanding of the whole concept:

> info **info** For a ready-made dedicated package, continue reading below.

1. First, create a new instance of the `AsyncLocalStorage` in some shared source file. Since we're using NestJS, let's also turn it into a module with a custom provider.

```
@@filename(als.module)
@Module({
  providers: [
    {
      provide: AsyncLocalStorage,
      useValue: new AsyncLocalStorage(),
    },
  ],
  exports: [AsyncLocalStorage],
})
export class AlsModule {}
```

> info **Hint** `AsyncLocalStorage` is imported from `async_hooks`.

2. We're only concerned with HTTP, so let's use a middleware to wrap the `next` function with `AsyncLocalStorage#run`. Since a middleware is the first thing that the request hits, this will make the `store` available in all enhancers and the rest of the system.

```
@@filename(app.module)
@Module({
  imports: [AlsModule]
  providers: [CatService],
  controllers: [CatController],
})
export class AppModule implements NestModule {
  constructor(
    // inject the AsyncLocalStorage in the module constructor,
    private readonly als: AsyncLocalStorage
  ) {}

  configure(consumer: MiddlewareConsumer) {
    // bind the middleware,
    consumer
      .apply((req, res, next) => {
        // populate the store with some default values
        // based on the request,
        const store = {
          userId: req.headers['x-user-id'],
        };
        // and pass the "next" function as callback
        // to the "als.run" method together with the store.
        this.als.run(store, () => next());
      })
      // and register it for all routes (in case of Fastify use '(.*)')
      .forRoutes('*');
  }
}
@@switch
@Module({
  imports: [AlsModule]
  providers: [CatService],
  controllers: [CatController],
})
@Dependencies(AsyncLocalStorage)
export class AppModule {
  constructor(als) {
    // inject the AsyncLocalStorage in the module constructor,
    this.als = als
  }

  configure(consumer) {
    // bind the middleware,
    consumer
      .apply((req, res, next) => {
        // populate the store with some default values
        // based on the request,
        const store = {
          userId: req.headers['x-user-id'],
        };
        // and pass the "next" function as callback
        // to the "als.run" method together with the store.
```

```
        this.als.run(store, () => next());
      })
      // and register it for all routes (in case of Fastify use '(.*)')
      .forRoutes('*');
  }
}
```

3. Now, anywhere within the lifecycle of a request, we can access the local store instance.

```
@@filename(cat.service)
@Injectable()
export class CatService {
  constructor(
    // We can inject the provided ALS instance.
    private readonly als: AsyncLocalStorage,
    private readonly catRepository: CatRepository,
  ) {}

  getCatForUser() {
    // The "getStore" method will always return the
    // store instance associated with the given request.
    const userId = this.als.getStore()["userId"] as number;
    return this.catRepository.getForUser(userId);
  }
}
@@switch
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
  constructor(als, catRepository) {
    // We can inject the provided ALS instance.
    this.als = als
    this.catRepository = catRepository
  }

  getCatForUser() {
    // The "getStore" method will always return the
    // store instance associated with the given request.
    const userId = this.als.getStore()["userId"] as number;
    return this.catRepository.getForUser(userId);
  }
}
```

4. That's it. Now we have a way to share request related state without needing to inject the whole `REQUEST` object.

> warning **warning** Please be aware that while the technique is useful for many use-cases, it inherently obfuscates the code flow (creating implicit context), so use it responsibly and especially avoid creating contextual "God objects".

NestJS CLS

The nestjs-cls package provides several DX improvements over using plain `AsyncLocalStorage` (`CLS` is an abbreviation of the term *continuation-local storage*). It abstracts the implementation into a `ClsModule` that offers various ways of initializing the `store` for different transports (not only HTTP), as well as a strong-typing support.

The store can then be accessed with an injectable `ClsService`, or entirely abstracted away from the business logic by using Proxy Providers.

> info **info** `nestjs-cls` is a third party package and is not managed by the NestJS core team. Please, report any issues found with the library in the appropriate repository.

Installation

Apart from a peer dependency on the `@nestjs` libs, it only uses the built-in Node.js API. Install it as any other package.

```
npm i nestjs-cls
```

Usage

A similar functionality as described above can be implemented using `nestjs-cls` as follows:

1. Import the `ClsModule` in the root module.

```
@@filename(app.module)
@Module({
  imports: [
    // Register the ClsModule,
    ClsModule.forRoot({
      middleware: {
        // automatically mount the
        // ClsMiddleware for all routes
        mount: true,
        // and use the setup method to
        // provide default store values.
        setup: (cls, req) => {
          cls.set('userId', req.headers['x-user-id']);
        },
      },
    }),
  ],
  providers: [CatService],
  controllers: [CatController],
})
export class AppModule {}
```

2. And then can use the `ClsService` to access the store values.

```
@@filename(cat.service)
@Injectable()
export class CatService {
  constructor(
    // We can inject the provided ClsService instance,
    private readonly cls: ClsService,
    private readonly catRepository: CatRepository,
  ) {}

  getCatForUser() {
    // and use the "get" method to retrieve any stored value.
    const userId = this.cls.get('userId');
    return this.catRepository.getForUser(userId);
  }
}
@@switch
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
  constructor(cls, catRepository) {
    // We can inject the provided ClsService instance,
    this.cls = cls
    this.catRepository = catRepository
  }

  getCatForUser() {
    // and use the "get" method to retrieve any stored value.
    const userId = this.cls.get('userId');
    return this.catRepository.getForUser(userId);
  }
}
```

3. To get strong typing of the store values managed by the `ClsService` (and also get auto-suggestions of the string keys), we can use an optional type parameter `ClsService<MyClsStore>` when injecting it.

```
export interface MyClsStore extends ClsStore {
  userId: number;
}
```

> info **hint** It it also possible to let the package automatically generate a Request ID and access it later with `cls.getId()`, or to get the whole Request object using `cls.get(CLS_REQ)`.

Testing

Since the `ClsService` is just another injectable provider, it can be entirely mocked out in unit tests.

However, in certain integration tests, we might still want to use the real `ClsService` implementation. In that case, we will need to wrap the context-aware piece of code with a call to `ClsService#run` or `ClsService#runWith`.

```
describe('CatService', () => {
  let service: CatService
  let cls: ClsService
  const mockCatRepository = createMock<CatRepository>()

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      // Set up most of the testing module as we normally would.
      providers: [
        CatService,
        {
          provide: CatRepository
          useValue: mockCatRepository
        }
      ],
      imports: [
        // Import the static version of ClsModule which only provides
        // the ClsService, but does not set up the store in any way.
        ClsModule
      ],
    }).compile()

    service = module.get(CatService)

    // Also retrieve the ClsService for later use.
    cls = module.get(ClsService)
  })

  describe('getCatForUser', () => {
    it('retrieves cat based on user id', async () => {
      const expectedUserId = 42
      mockCatRepository.getForUser.mockImplementationOnce(
        (id) => ({ userId: id })
      )

      // Wrap the test call in the `runWith` method
      // in which we can pass hand-crafted store values.
      const cat = await cls.runWith(
        { userId: expectedUserId },
        () => service.getCatForUser()
      )

      expect(cat.userId).toEqual(expectedUserId)
    })
  })
})
```

More information

Visit the NestJS CLS GitHub Page for the full API documentation and more code examples.