Design Patterns Explained – Adapter Pattern with Code Examples

THORBEN JANSSEN | MAY 25, 2018 |

DEVELOPER TIPS, TRICKS & RESOURCES (HTTPS://STACKIFY.COM/DEVELOPERS/) |

LEAVE A COMMENT (HTTPS://STACKIFY.COM/DESIGN-PATTERNS-EXPLAINED-ADAPTER-PATTERN-WITH-CODE-EXAMPLES/#RESPOND)

Topics/K

Java (https://s

Searc

Azure (https://s

AWS (https://s

Rel

Cloud (

tag=clou

Rec

Popul

Retrace

Pos

Cat

Searc

News

Get the l

Design patterns provide a reliable and easy way to follow proven design principles (https://stackify.com/solid-design-principles/) and to write well-structured and maintainable code. One of the popular and often used patterns in object-oriented software development is the adapter pattern. It follows Robert C. Martin's Dependency Inversion Principle (https://stackify.com/dependency-inversion-principle/) and enables you to reuse an existing class even so it doesn't implement an expected interface.

If you do some research on the adapter pattern, you will find two different versions of it:

1. The class adapter pattern that implements the adapter using inheritance (https://stackify.com/oop-concept-inheritance/).
2. The object adapter pattern that uses composition (https://stackify.com/oop-concepts-composition/) to reference an instance of the wrapped class within the adapter.

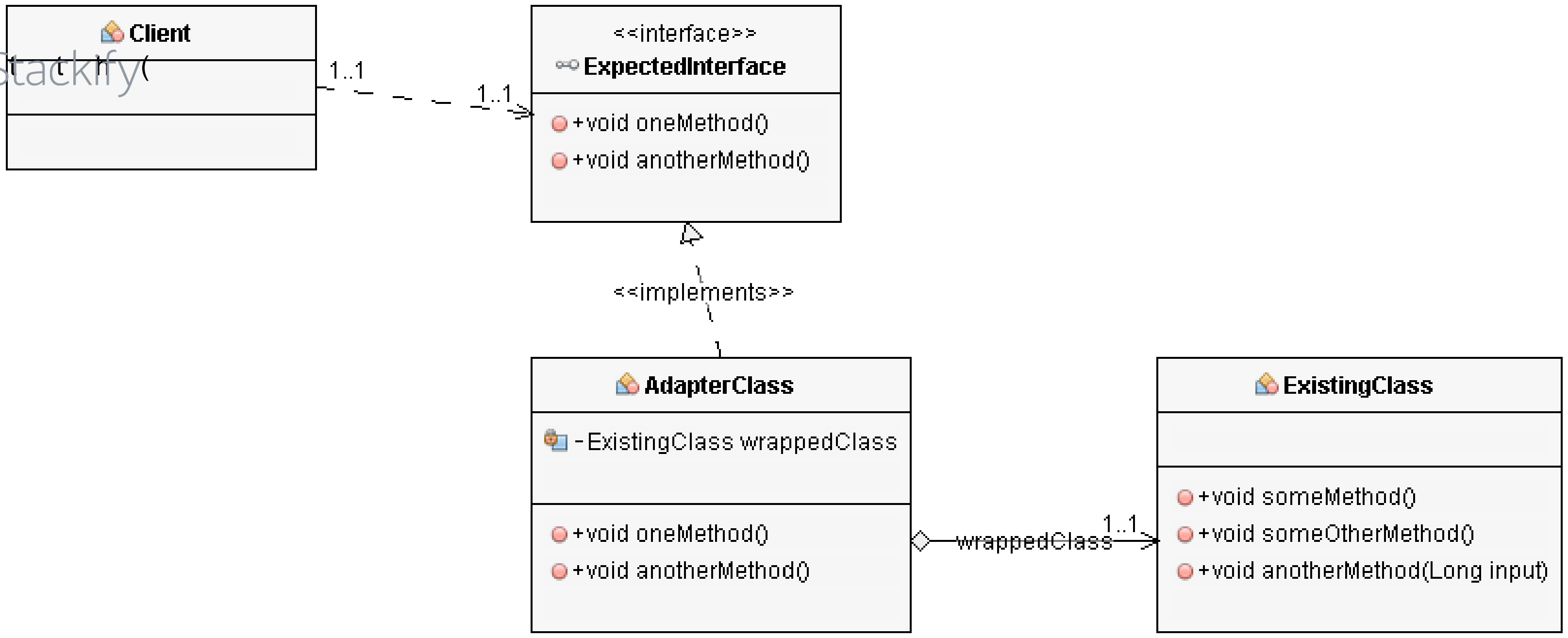
You are probably aware of all the discussions about inheritance vs. composition (https://www.thoughtworks.com/de/insights/blog/composition-vs-inheritance-how-choose). Composition provides more flexibility and avoids unexpected side-effect when you need to change existing code. Due to this, the object adapter pattern is by far the more popular approach and the one I will focus on in this article.

The Adapter Pattern

The general idea of an adapter in software development is identical to the one in the physical world. If you have been to different countries, you probably recognized that a lot of them are using differently shaped power sockets (https://www.power-plugs-sockets.com/). Quite often, they are shaped in a way that the plug of your electrical device doesn't fit. So, how do you connect the charger of your mobile phone or laptop to these power sockets?

The answer is simple. You get an adapter which you can put into the power socket and then you put your plug into the other end of the adapter. The adapter changes the form of your plug so that you can use it with the power socket. In that example and in most other situations, the adapter doesn't provide any additional functionality. It just enables you to connect your plug to the power socket.

The Adapter Pattern applies the same idea to object-oriented programming by introducing an additional adapter class between an interface (https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) and an existing class.



The adapter class implements the expected interface and keeps a reference to an object of the class you want to reuse. The methods defined by the interface call one or more methods on the referenced object and return a value of the expected type. By doing that, the adapter class fulfills the expected contract by implementing the interface and enables you to reuse existing, incompatible implementations.

Let's apply the pattern to an example.

Brewing coffee using the Adapter Pattern

I like to start my morning with a fresh cup of coffee. The only issue is that I need to get out of bed and prepare the coffee before I can drink it. It would be much better if it would be automatically prepared when my alarm rings. So, let's build a small app for it.

(https://info.stackify.com/cs/c/?cta_guid=eb8ba9dc-2898-4c16-a893-432ae7a086a9&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Fdesign-patterns-explained-adapter-pattern-with-code-examples%2F&redirect_url=APefjpEsdonADyMoG4MHq1GksFksoOEql6thJPNTsqvnyy_j0x7pzvYMe-0XdmwpSbqMwTGTx_OBJWkW-IR3nwiycrQb0YpsRcuA3U6m5lS3qZCJ3S7jep70wallITzRO02GUK59pDUVZgx36dwdlFUdB-BUTHuk6OoQFf6CFXIj7T7xWpAW_CPIdNRJLFBqem30F9g6GENnDlUpAcBho3UCeEW0aKFiQqc7eatCGh6HmA-tvcKrisYM-FsYmSL4MIsbhmEf2-09&click=680c2b9a-7080-4123-817d-8a1a9649b90f&hsutk=96dd063e62a1f430781aad63fce9a337&utm_referrer=https%3A%2F%2Fwww.google.com%2F&__hstc=23835621.96d

An app to brew a coffee

The *FilterCoffeeApp* does exactly that. It expects an implementation of the *FilterCoffeeMachine* interface as a constructor parameter and uses it in the *prepareCoffee* method to brew a cup of filter coffee.

```
public class FilterCoffeeApp {
    private Logger log = Logger.getLogger(
        FilterCoffeeApp.class.getSimpleName());

    private FilterCoffeeMachine coffeeMachine;

    public FilterCoffeeApp(FilterCoffeeMachine coffeeMachine) {
        this.coffeeMachine = coffeeMachine;
    }

    public Coffee prepareCoffee() {
        Coffee coffee = this.coffeeMachine.brewCoffee();
        log.info("Coffee is ready!");
        log.info(" -> " + coffee);
        return coffee;
    }
}
```

The *FilterCoffeeMachine* interface is relatively simple. It just defines the *brewCoffee* method which you can call to make a cup of coffee.

```
public interface FilterCoffeeMachine {
    Coffee brewCoffee();
}
```

That seems to be a good approach that enables you to use different coffee machines with your application. The only requirement is that all classes that represent a coffee machine implement the *FilterCoffeeMachine* interface.

A basic coffee machine

The *BasicCoffeeMachine* class implements that interface and can be used by the *FilterCoffeeApp*.

```
public class BasicCoffeeMachine implements FilterCoffeeMachine {
    private Configuration config;

    private Map<CoffeeSelection, GroundCoffee> groundCoffee; private BrewingUnit brewingUnit;

    public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee) {
        this.groundCoffee = coffee;
        this.brewingUnit = new BrewingUnit();
        this.config = new Configuration(30, 480);
    }

    @Override
    public Coffee brewCoffee() {
        // get the coffee
        GroundCoffee groundCoffee = this.groundCoffee.get(
            CoffeeSelection.FILTER_COFFEE);

        // brew a filter coffee
        return this.brewingUnit.brew(
            CoffeeSelection.FILTER_COFFEE, groundCoffee,
            this.config.getQuantityWater());
    }

    public void addGroundCoffee(
        CoffeeSelection sel, GroundCoffee newCoffee)
        throws CoffeeException {
        GroundCoffee existingCoffee = this.groundCoffee.get(sel);
        if (existingCoffee != null) {
            if (existingCoffee.getName().equals(newCoffee.getName())) {
                existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.getQuantity());
            } else {
                throw new CoffeeException(
                    "Only one kind of coffee supported for each CoffeeSelection.");
            }
        } else {
            this.groundCoffee.put(sel, newCoffee);
        }

    }
}
```

A premium coffee machine

But what happens if you get a new, more advanced coffee machine that doesn't implement the *FilterCoffeeMachine* interface?

Stackify

```
public class PremiumCoffeeMachine {

    private Map<CoffeeSelection, Configuration> configMap;
    private Map<CoffeeSelection, CoffeeBean> beans;
    private Grinder grinder;
    private BrewingUnit brewingUnit;

    public PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {
        this.beans = beans;
        this.grinder = new Grinder();
        this.brewingUnit = new BrewingUnit();
        this.configMap = new HashMap<>();
        this.configMap.put(
            CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480));
        this.configMap.put(
            CoffeeSelection.ESPRESSO, new Configuration(8, 28));
    }

    public Coffee brewCoffee(CoffeeSelection selection)
        throws CoffeeException {
        switch (selection) {
        case FILTER_COFFEE:
            return brewFilterCoffee();
        case ESPRESSO:
            return brewEspresso();
        default:
            throw new CoffeeException(
```

Topics/K
.NET Co
(https://
core/)
Java
(https://s
Searc
Azure
(https://s
Sea
AWS
(https://s
Rel
Cloud (
tag=clou
Rec
Popul
Retrace
Pos
Cat
Searc
News
Get the

```
                    "CoffeeSelection " + selection + " not supported");
            }
    }

    private Coffee brewEspresso() {
        Configuration config = configMap.get(
            CoffeeSelection.ESPRESSO);

        // grind the coffee beans
        GroundCoffee groundCoffee = this.grinder.grind(
            this.beans.get(CoffeeSelection.ESPRESSO),
            config.getQuantityCoffee());

        // brew an espresso
        return this.brewingUnit.brew(
            CoffeeSelection.ESPRESSO, groundCoffee,
            config.getQuantityWater());
    }

    private Coffee brewFilterCoffee() {
        Configuration config = configMap.get(
            CoffeeSelection.FILTER_COFFEE);

        // grind the coffee beans
        GroundCoffee groundCoffee = this.grinder.grind(
            this.beans.get(CoffeeSelection.FILTER_COFFEE),
            config.getQuantityCoffee());
        // brew a filter coffee
        return this.brewingUnit.brew(
            CoffeeSelection.FILTER_COFFEE, groundCoffee,
            config.getQuantityWater());
    }

    public void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans) throws CoffeeException
{
        CoffeeBean existingBeans = this.beans.get(sel);

        if (existingBeans != null) {
            if (existingBeans.getName().equals(newBeans.getName())) {
                existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());
            } else {
                throw new CoffeeException(
                    "Only one kind of coffee supported for each CoffeeSelection.");
            }
        } else {
            this.beans.put(sel, newBeans);
        }
    }
}
```

You can use the *Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException* method of the *PremiumCoffeeMachine* to prepare filter coffee or espresso. As you can see, the method has the same name as the one defined by the *FilterCoffeeMachine* interface, but the method signature is incompatible. It expects a parameter and declares an exception (https://stackify.com/specify-handle-exceptions-java/).

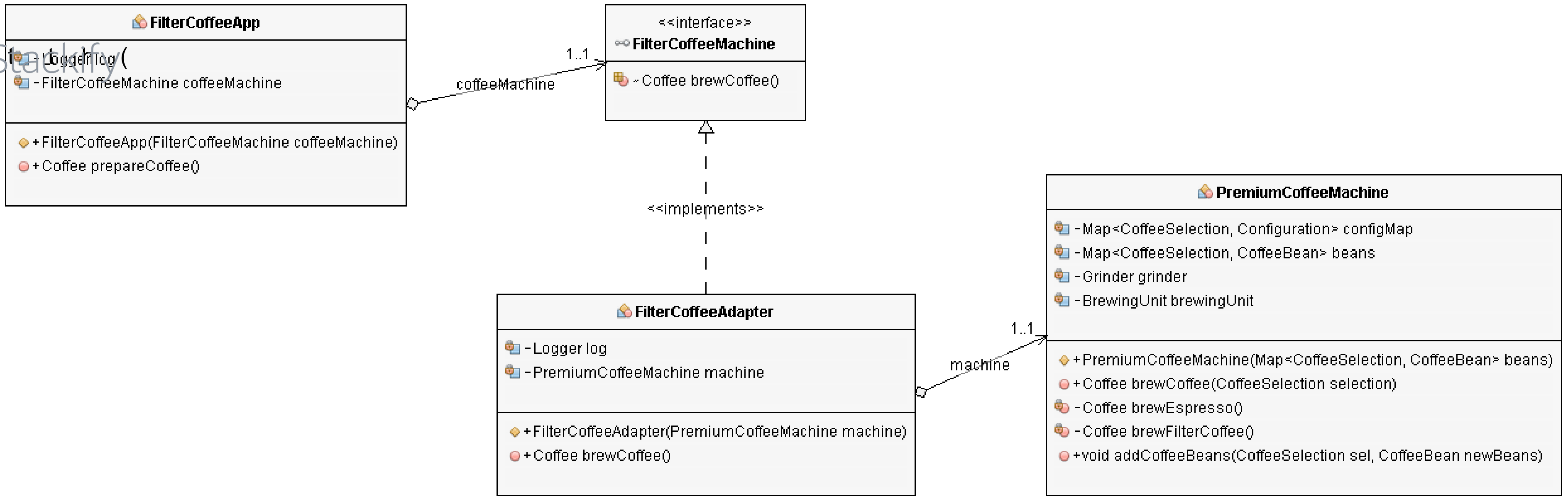
The *PremiumCoffeeMachine* represents a coffee machine, but it doesn't implement the *FilterCoffeeMachine* interface. So, you can't use it with the *FilterCoffeeApp*.

I will not change the class so that it implements the required interface. It's oftentimes not possible to change existing classes because they are implemented by a different team or the classes are used in other projects which don't contain the required interface. I also don't want to change the *FilterCoffeeMachine* interface. The *BasicCoffeeMachine* implements that interface and I would need to change that class whenever I change the interface. In these situations, it's better to apply the adapter pattern.

(https://info.stackify.com/cs/c/?cta_guid=eb8ba9dc-2898-4c16-a893-432ae7a086a9&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Fdesign-patterns-explained-adapter-pattern-with-code-examples%2F&redirect_url=APefjpEsdonADyMoG4MHq1GksFksoOEql6thJPNTsqvnyy_j0x7pzvYMe-0XdmwpSbqMwTGTx_OBJWkW-IR3nwiycrQb0YpsRcuA3U6m5lS3qZCJ3S7jep70wallITzRO02GUK59pDUVZgx36dwdlFUdB-BUTHuk6OoQFf6CFXIj7T7xWpAW_CPIdNRJLFBqem30F9g6GENnDlUpAcBho3UCeEW0aKFiQqc7eatCGh6HmA-tvcKrisYM-FsYmSL4MIsbhmEf2-09&click=680c2b9a-7080-4123-817d-8a1a9649b90f&hsutk=96dd063e62a1f430781aad63fce9a337&utm_referrer=https%3A%2F%2Fwww.google.com%2F&__hstc=23835621.96d

Implementing the adapter

By introducing an adapter class, that implements the *FilterCoffeeMachine* interface and wraps the *PremiumCoffeeMachine* class, you enable your *FilterCoffeeApp* to use the coffee machine.


FilterCoffeeApp
-Logger log
-FilterCoffeeMachine coffeeMachine
+FilterCoffeeApp(FilterCoffeeMachine coffeeMachine)
+Coffee prepareCoffee()
coffeeMachine
1..1
<<interface>>
FilterCoffeeMachine
~Coffee brewCoffee()
<<implements>>
FilterCoffeeAdapter
-Logger log
-PremiumCoffeeMachine machine
+FilterCoffeeAdapter(PremiumCoffeeMachine machine)
+Coffee brewCoffee()
machine
1..1
PremiumCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, CoffeeBean> beans
-Grinder grinder
-BrewingUnit brewingUnit
+PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
+Coffee brewCoffee(CoffeeSelection selection)
-Coffee brewEspresso()
-Coffee brewFilterCoffee()
+void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans)

For this example, the adapter class needs to fulfill two tasks:

1. It has to provide an implementation of the *FilterCoffeeMachine* interface.
2. The *brewCoffee* method needs to bridge the gap between the *brewCoffee* method defined by the interface and the *brewCoffee* method implemented by the *PremiumCoffeeMachine* class.

The interface and the existing class are not too different. That makes the implementation of the adapter class relatively simple.

```
public class FilterCoffeeAdapter implements FilterCoffeeMachine {
    private Logger log = Logger.getLogger(
        FilterCoffeeAdapter.class.getSimpleName());

    private PremiumCoffeeMachine machine;

    public FilterCoffeeAdapter(PremiumCoffeeMachine machine) {
        this.machine = machine;
    }

    @Override
    public Coffee brewCoffee() {
        try {
            return machine.brewCoffee(
                CoffeeSelection.FILTER_COFFEE);
        } catch (CoffeeException e) {
            log.severe(e.toString());
            return null;
        }
    }
}
```

As you can see in the code snippet, the *FilterCoffeeAdapter* class implements the *FilterCoffeeMachine* interface and expects a *PremiumCoffeeMachine* object as a constructor parameter. It keeps that object in a private field so that it can use it in the *brewCoffee* method.

The implementation of the *brewCoffee* method is the critical and for most adapter classes, the most difficult part. In this case, the *PremiumCoffeeMachine* class provides a method that you can call to perform the task. But it's more flexible and requires a *CoffeeSelection* enum value to define which kind of coffee it shall produce.

The method of the *PremiumCoffeeMachine* class throws a *CoffeeException* if it gets called with a *CoffeeSelection* value different than *FILTER_COFFEE* and *ESPRESSO*. The *brewCoffee* method of the *FilterCoffeeMachine* interface doesn't declare this exception and you need to handle it within the method implementation.

In this example, there is no perfect way to do that. You can either write a log message and return null, as I did in the code snippet, or you can throw a (https://stackify.com/specify-handle-exceptions-java/) *RuntimeException* (https://stackify.com/specify-handle-exceptions-java/).

In your application, you might have better ways to handle the exception. You might be able to perform a retry or to trigger a different business operation. This would make your application more robust and improve the implementation of your adapter.

Summary

The Adapter Pattern is an often-used pattern in object-oriented programming languages. Similar to adapters in the physical world, you implement a class that bridges the gap between an expected interface and an existing class. That enables you to reuse an existing class that doesn't implement a required interface and to use the functionality of multiple classes, that would otherwise be incompatible.

One advantage of the Adapter Pattern is that you don't need to change the existing class or interface. By introducing a new class, which acts as an adapter between the interface and the class, you avoid any changes to the existing code. That limits the scope of your changes to your software component and avoids any changes and side-effects in other components or applications.

The Adapter Pattern provides an implementation of the Dependency Inversion design principle. If you're not already familiar with it, I recommend reading about the different SOLID design principles. I wrote a series of articles explaining all five of them:

- The Single Responsibility Principle (https://stackify.com/solid-design-principles/)
- The Open/Closed Principle with Code Examples (https://stackify.com/solid-design-open-closed-principle/)
- The Liskov Substitution Principle with Code Examples (https://stackify.com/solid-design-liskov-substitution-principle/)
- Interface Segregation with Code Examples (https://stackify.com/interface-segregation-principle/)
- Dependency Inversion Principle with Code Examples (https://stackify.com/dependency-inversion-principle/)

About the Author | Latest Posts

About Thorben Janssen

Thorben is an independent trainer and author of the Amazon bestselling book *Hibernate Tips - More than 70 solutions to common Hibernate problems.*He writes about Java EE related topics on his blog Thoughts on Java (https://www.thoughts-on-java.org).

(https://plus.google.com/u/0/b/109323639830862412239/109323639830862412239)
(http://thjanssen123) (https://www.linkedin.com/in/thorbenjanssen/)
(https://plus.google.com/u/0/b/109323639830862412239/109323639830862412239?rel=author)

Leave a Reply

Your email address will not be published.

Comment

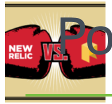
Name

Your Name

Email

Topics/K

.NET Co

core/)
Java
(https://s
Searc
Azure
(https://s
Sea
AWS
(https://s
Rel
Cloud (
tag=clou
Rec
Popul

Pop
Retrace
Pos

Cat
Searc
News

Get the l

Stackify

Your Email

Website

Your Website

Submit

Improve Your Code with Retrace APM

Stackify's APM tools are used by thousands of .NET, Java, and PHP developers all over the world.
Explore Retrace's product features to learn more.

(/retrace-application-performance-management/)

App Performance Monitoring (https://stackify.com/retrace-

(/retrace-code-profiling/)

Code Profiling (https://stackify.com/retrace-code-profiling/)

(/retrace-error-monitoring/)

Error Tracking (https://stackify.com/retrace-error-monitoring/)

(/retrace-log-management/)

Centralized Logging (https://stackify.com/retrace-log-management/)

(/retrace-app-metrics/)

App & Server Metrics (https://stackify.com/retrace-app-metrics/)

Topics/K

.NET Co

core/)

Java (https://s

Searc

Azure

https://s

Sea

AWS (https://s

Rel

Cloud (

tag=clou

Rec

Popul

Retrace

Pos

Cat

Searc

News

Get the

Stackify application-performance-management/)

Learn More (/retrace/)

Get In Touch	Product	Solutions	Resources	Company
Contact Us (https://stackify.com/contact-us/)	Retrace APM (https://stackify.com/retrace-application-performance-management/)	Log Management (https://stackify.com/retrace-log-management/)	Pricing (https://stackify.com/pricing	About Us (https://stackify.com/?page_id=16004)
Request a Demo (https://stackify.com/demo-request/)	Prefix (https://stackify.com/prefix/	Application Performance Management (https://stackify.com/retrace-application-performance-management/)	Case Studies (/stackify-case-studies/)	GDPR (https://stackify.com/about/g
8900 State Line Rd #100 Leawood, KS 66206	.NET Monitoring (/retrace-apm-dotnet/)	Application Metrics (https://stackify.com/retrace-app-metrics/)	Blog (https://stackify.com/blog/)	Careers (https://stackify.com/careers
816-888-5055 (tel:18168885055)	Java Monitoring (/retrace-apm-java/)	Azure Monitoring (https://stackify.com/azure-monitoring/)	Podcast (https://stackify.com/develo things/)	Security Information (https://stackify.com/stackify-security-information/)
[illegible]	PHP Monitoring (/retrace-apm-php/)	Error Tracking (https://stackify.com/retrace-error-monitoring/)	Documentation (https://docs.stackify.com/v	Terms & Conditions (https://stackify.com/terms-conditions/)
	Node.js Monitoring (/retrace-apm-nodejs/)		Free eBooks (https://stackify.com/stackify-developer-ebooks/)	Privacy Policy (https://stackify.com/privacy-policy/)
	Ruby Monitoring (/retrace-apm-ruby/)		Videos (https://www.youtube.com/s	

Topics/K

.NET Co

Java (https://s

Searc

Azure (https://s

AWS (https://s

Rel

Cloud (

tag=clou

Rec

Popul

Retrace

Pos

Cat

News

Get the l

: p Stackify (

[Retrace vs New Relic](https://stackify.com/new-relic-alternatives-for-developers/) (https://stackify.com/new-relic-alternatives-for-developers/)

[Retrace vs Application Insights](https://stackify.com/micros) (https://stackify.com/micros application-insights-alternative/)

[Explore Retrace Sandbox](https://sandbox.stackify.co) (https://sandbox.stackify.coı sandbox=true)

[Code Profiling](https://stackify.com/retrace) (https://stackify.com/retrace code-profiling/)

[What is APM?](https://stackify.com/retrace) (https://stackify.com/retrace application-performance-management/)

[Ideas Portal](https://ideas.stackify.com/?) (https://ideas.stackify.com/? _ga=2.150793076.3346703 1540781610.1523310878)

[APM ROI](https://stackify.com/6-types-apm-roi/) (https://stackify.com/6-types-apm-roi/)

software

Email

Searc

Sea

Rel

Rec

Pop

Pos
Cat

Newsl

Get the l