CONTENTS

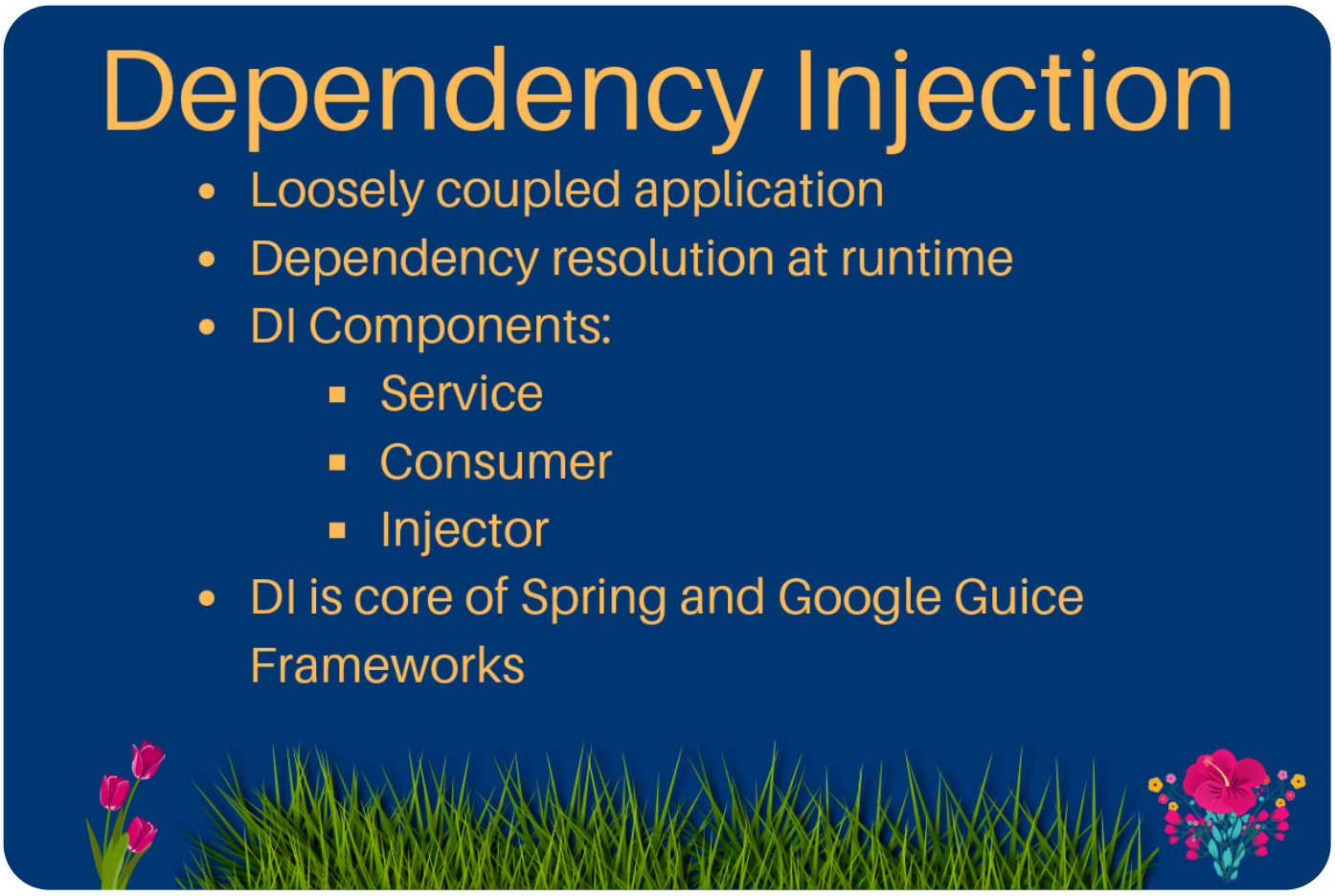
// TUTORIAL //

Java Dependency Injection - DI Design Pattern Example Tutorial

Published on August 3, 2022

Design Patterns Java

By Pankaj



While we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the “report an issue“ button at the bottom of the tutorial.

Java Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement **dependency injection in java** to move the dependency resolution from compile-time to runtime.

Java Dependency Injection

Java Dependency injection seems hard to grasp with theory, so I would take a simple example and then we will see how to use dependency injection pattern to achieve loose coupling and extendability in the application. Let's say we have an application where we consume `EmailService` to send emails. Normally we would implement this like below.

```
package com.journaldev.java.legacy;

public class EmailService {

	public void sendEmail(String message, String receiver){
		//logic to send email
		System.out.println("Email sent to "+receiver+ " with Message="+message);
	}
}
```

`EmailService` class holds the logic to send an email message to the recipient email address. Our application code will be like below.

```
package com.journaldev.java.legacy;

public class MyApplication {

	private EmailService email = new EmailService();

	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.email.sendEmail(msg, rec);
	}
}
```

Our client code that will use `MyApplication` class to send email messages will be like below.

```
package com.journaldev.java.legacy;

public class MyLegacyTest {

	public static void main(String[] args) {
		MyApplication app = new MyApplication();
		app.processMessages("Hi Pankaj", "pankaj@abc.com");
	}

}
```

At first look, there seems nothing wrong with the above implementation. But above code logic has certain limitations.

- `MyApplication` class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in the future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even harder.
- If we want to extend our application to provide an additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.
- Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

One can argue that we can remove the email service instance creation from `MyApplication` class by having a constructor that requires email service as an argument.

```
package com.journaldev.java.legacy;

public class MyApplication {

	private EmailService email = null;

	public MyApplication(EmailService svc){
		this.email=svc;
	}

	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.email.sendEmail(msg, rec);
	}
}
```

But in this case, we are asking client applications or test classes to initializing the email service that is not a good design decision. Now let's see how we can apply java dependency injection pattern to solve all the problems with the above implementation. Dependency Injection in java requires at least the following:

1. Service components should be designed with base class or interface. It's better to prefer interfaces or abstract classes that would define contract for the services.
2. Consumer classes should be written in terms of service interface.
3. Injector classes that will initialize the services and then the consumer classes.

Java Dependency Injection - Service Components

For our case, we can have `MessageService` that will declare the contract for service implementations.

```
package com.journaldev.java.dependencyinjection.service;

public interface MessageService {

	void sendMessage(String msg, String rec);
}
```

Now let's say we have Email and SMS services that implement the above interfaces.

```
package com.journaldev.java.dependencyinjection.service;

public class EmailServiceImpl implements MessageService {

	@Override
	public void sendMessage(String msg, String rec) {
		//logic to send email
		System.out.println("Email sent to "+rec+ " with Message="+msg);
	}

}
```

```
package com.journaldev.java.dependencyinjection.service;

public class SMSServiceImpl implements MessageService {

	@Override
	public void sendMessage(String msg, String rec) {
		//logic to send SMS
		System.out.println("SMS sent to "+rec+ " with Message="+msg);
	}

}
```

Our dependency injection java services are ready and now we can write our consumer class.

Java Dependency Injection - Service Consumer

We are not required to have base interfaces for consumer classes but I will have a `Consumer` interface declaring contract for consumer classes.

```
package com.journaldev.java.dependencyinjection.consumer;

public interface Consumer {

	void processMessages(String msg, String rec);
}
```

My consumer class implementation is like below.

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

	private MessageService service;

	public MyDIApplication(MessageService svc){
		this.service=svc;
	}

	@Override
	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.service.sendMessage(msg, rec);
	}

}
```

Notice that our application class is just using the service. It does not initialize the service that leads to better "*separation of concerns*". Also use of service interface allows us to easily test the application by mocking the MessageService and bind the services at runtime rather than compile time. Now we are ready to write **java dependency injector classes** that will initialize the service and also consumer classes.

Java Dependency Injection - Injectors Classes

Let's have an interface `MessageServiceInjector` with method declaration that returns the `Consumer` class.

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

public interface MessageServiceInjector {

	public Consumer getConsumer();
}
```

Now for every service, we will have to create injector classes like below.

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		return new MyDIApplication(new EmailServiceImpl());
	}

}
```

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.SMSServiceImpl;

public class SMSServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		return new MyDIApplication(new SMSServiceImpl());
	}

}
```

Now let’s see how our client applications will use the application with a simple program.

```
package com.journaldev.java.dependencyinjection.test;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.injector.EmailServiceInjector;
import com.journaldev.java.dependencyinjection.injector.MessageServiceInjector;
import com.journaldev.java.dependencyinjection.injector.SMSServiceInjector;

public class MyMessageDITest {

	public static void main(String[] args) {
		String msg = "Hi Pankaj";
		String email = "pankaj@abc.com";
		String phone = "4088888888";
		MessageServiceInjector injector = null;
		Consumer app = null;

		//Send email
		injector = new EmailServiceInjector();
		app = injector.getConsumer();
		app.processMessages(msg, email);

		//Send SMS
		injector = new SMSServiceInjector();
		app = injector.getConsumer();
		app.processMessages(msg, phone);
	}

}
```

As you can see that our application classes are responsible only for using the service. Service classes are created in injectors. Also if we have to further extend our application to allow facebook messaging, we will have to write Service classes and injector classes only. So dependency injection implementation solved the problem with hard-coded dependency and helped us in making our application flexible and easy to extend. Now let's see how easily we can test our application class by mocking the injector and service classes.

Java Dependency Injection - JUnit Test Case with Mock Injector and Service

```
package com.journaldev.java.dependencyinjection.test;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

```
import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.injector.MessageServiceInjector;
import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplicationJUnitTest {

	private MessageServiceInjector injector;
	@Before
	public void setUp(){
		//mock the injector with anonymous class
		injector = new MessageServiceInjector() {

			@Override
			public Consumer getConsumer() {
				//mock the message service
				return new MyDIApplication(new MessageService() {

					@Override
					public void sendMessage(String msg, String rec)
						System.out.println("Mock Message Service

					}
				});
			}
		};
	}

	@Test
	public void test() {
		Consumer consumer = injector.getConsumer();
		consumer.processMessages("Hi Pankaj", "pankaj@abc.com");
	}

	@After
	public void tear(){
		injector = null;
	}

}
```

As you can see that I am using anonymous classes to *mock the injector and service classes* and I can easily test my application methods. I am using JUnit 4 for the above test class, so make sure it's in your project build path if you are running above test class. We have used constructors to inject the dependencies in the application classes, another way is to use a setter method to **inject dependencies** in application classes. For setter method dependency injection, our application class will be implemented like below.

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

	private MessageService service;

	public MyDIApplication(){}

	//setter dependency injection
	public void setService(MessageService service) {
		this.service = service;
	}

	@Override
```

Injection in Java is a way to achieve **inversion of control** (**IoC**) in our application by moving objects binding from compile time to runtime. We can achieve IoC through Factory Pattern, Template Method Design Pattern, Strategy Pattern and Service Locator pattern too. **Spring Dependency Injection**, **Google Guice** and **Java EE CDI** frameworks facilitate the process of dependency injection through use of Java Reflection API and java annotations. All we need is to annotate the field, constructor or setter method and configure them in configuration xml files or classes.

Benefits of Java Dependency Injection

Some of the benefits of using Dependency Injection in Java are:

- Separation of Concerns
- Boilerplate Code reduction in application classes because all work to initialize dependencies is handled by the injector component
- Configurable components makes application easily extendable
- Unit testing is easy with mock objects

Disadvantages of Java Dependency Injection

Java Dependency injection has some disadvantages too:

- If overused, it can lead to maintenance issues because the effect of changes are known at runtime.
- Dependency injection in java hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

Download Dependency Injection Project

That's all for **dependency injection pattern in java**. It's good to know and use it when we are in control of the services.

About the authors

Pankaj Author

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

JournalDev • December 19, 2020

hi, why does it need the consumer class and not enough to use the injector in order to return the appropriate message service

- irving

JournalDev • September 6, 2020

Pankaj, hello, could you explain, plz, what do we test here? @Test public void test() { Consumer consumer = injector.getConsumer();

consumer.processMessages(“Hi Pankaj”, “pankaj@abc.com”); }

- Hannah

JournalDev • July 11, 2020

Is this code correct? @Component public class Bean { @Inject private Bean1 anotherBean1; @Inject private Bean2 anotherBean2; private String property; public Bean(Bean1 anotherBean1, Bean2 anotherBean2) { this.anotherBean1 = anotherBean1; this.anotherBean2 = anotherBean2; } public void method1() { // use all member variables } }

- JAMIE

Show replies

JournalDev • March 22, 2020

Thank you very much !!! Really good and clear concepts...

- Roshan

JournalDev • December 4, 2019

Amazing! Thank you! Your articles are outstanding! They are clear and precise to the point.

- Perkone

JournalDev • October 25, 2019

While the article outlines key elements of dependency injection, the code examples provided illustrate how one can implement the factory design pattern.

This is obvious because you have to explicitly call getConsumer() every time. Anyway, decent article but the examples will cause confusion.

- Andrei

JournalDev • September 22, 2019

So clear to me now, thanks!

- Sizhong Du

JournalDev • April 5, 2019

TL;DR: DI just put all those connection points into an Object.

- Nyng

Show replies

JournalDev • January 29, 2019

Great article and well explained! Would you consider to add a few diagrams to this article? Or, at least, one diagram before and after the inversion of dependencies?

- Alejandro Carlstein

JournalDev • January 3, 2019

A great article. A message for those who don't understand the advantage as I did. It can look confusing at first and why we need so much classes and interfaces. But when you are going to test them as individual components you will know the advantage of it. You no more need to modify classes when there is a new change If we use this pattern we can avoid rewriting test cases again.

- Saravanan

Load More Comments

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

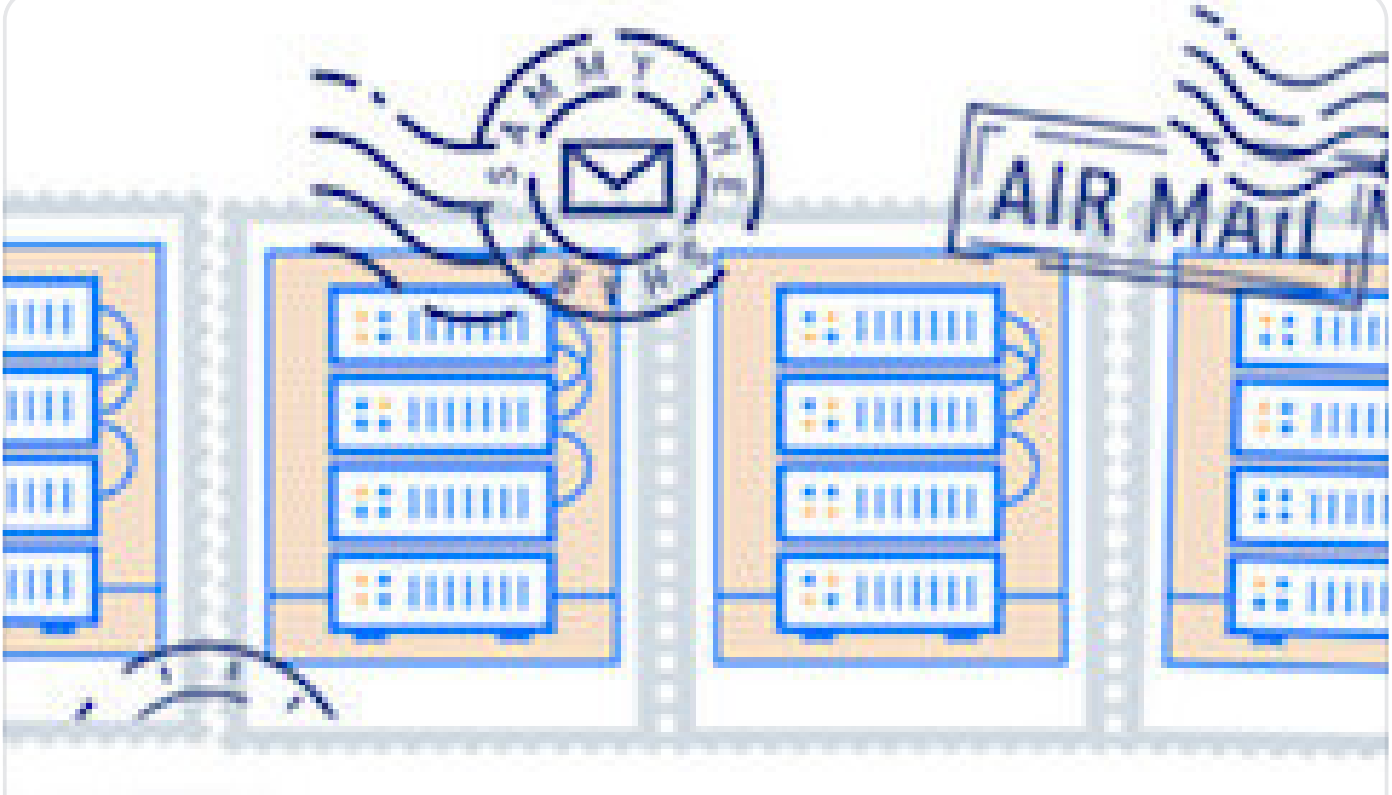
MySQL

Docker

Kubernetes

All tutorials →

Talk to an expert →



Get our biweekly newsletter

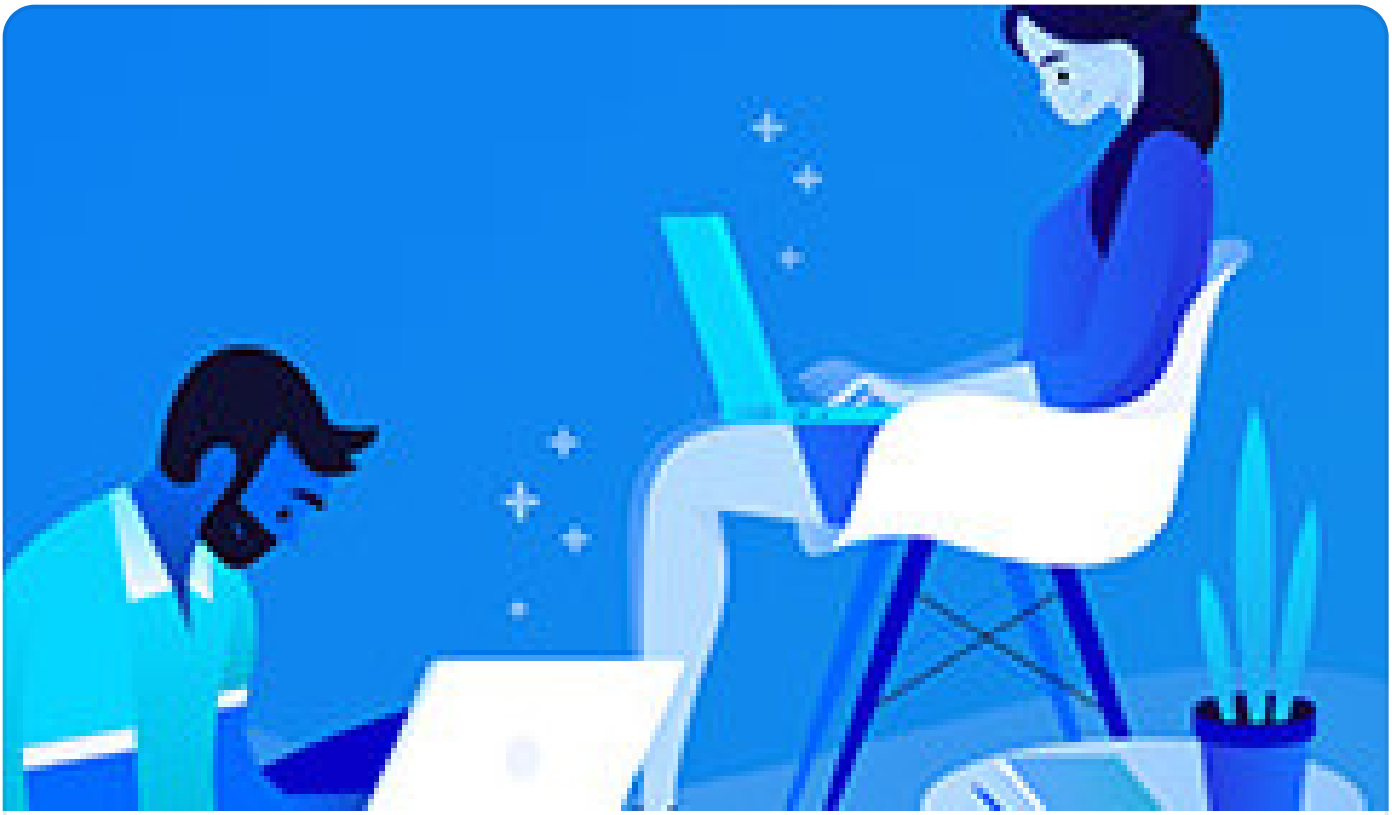
Sign up for Infrastructure as a Newsletter.

Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →



Become a contributor

Get paid to write technical tutorials and select a tech-focused charity to receive a matching donation.

Learn more →

Featured on Community

Kubernetes Course · Learn Python 3 · Machine Learning in Python

Getting started with Go · Intro to Kubernetes

DigitalOcean Products

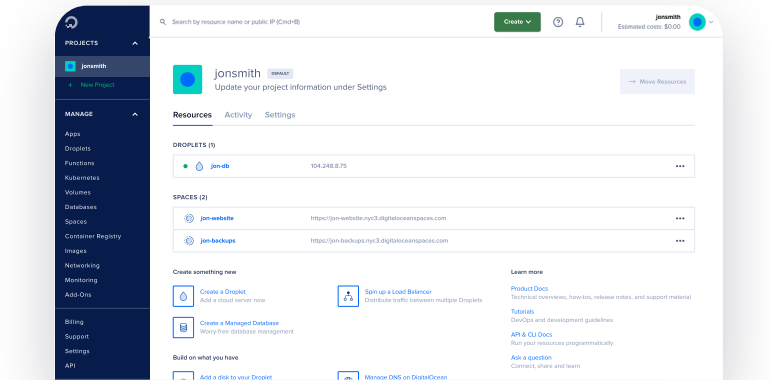
Cloudways · Virtual Machines · Managed Databases · Managed Kubernetes

Block Storage · Object Storage · Marketplace · VPC · Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow — whether you're running one virtual machine or ten thousand.

Learn more



Get started for free

Sign up and get $200 in credit for your first 60 days with DigitalOcean.

Get started

This promotional offer applies to new accounts only.

Company

Products

Community

Solutions

Contact