**Microservices**

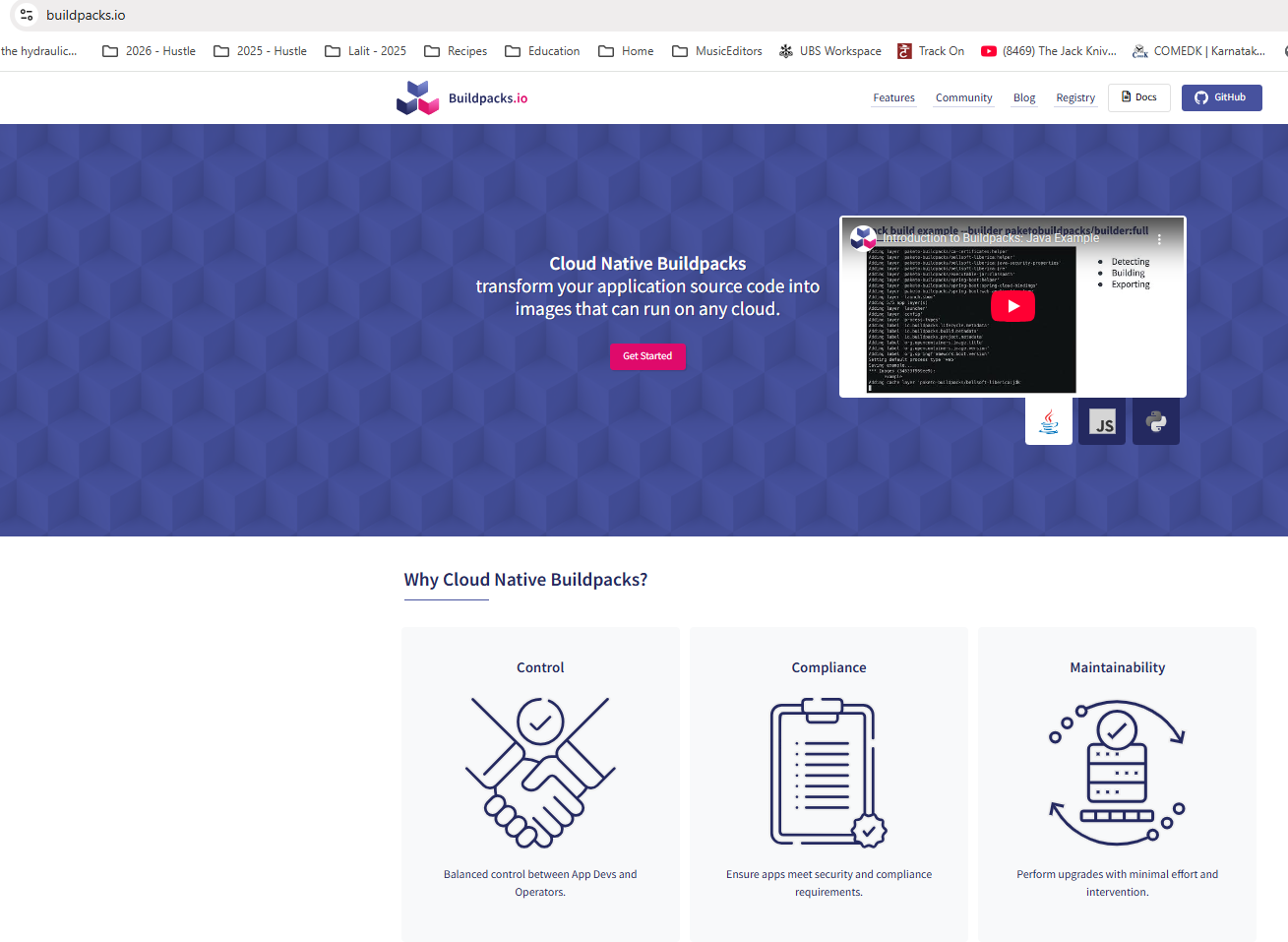
Contents

[BuildPacks 2](#_Toc212532847)

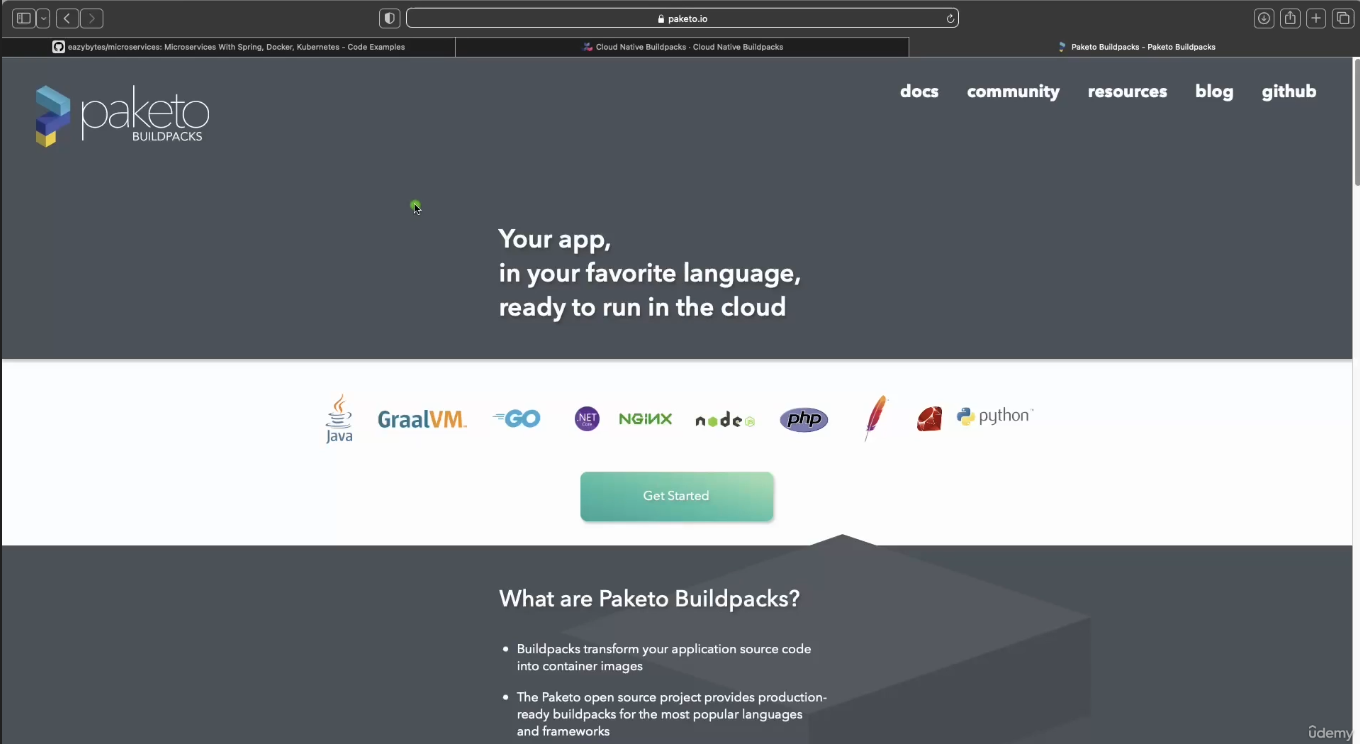
[Google Jib 10](#_Toc212532848)

### BuildPacks

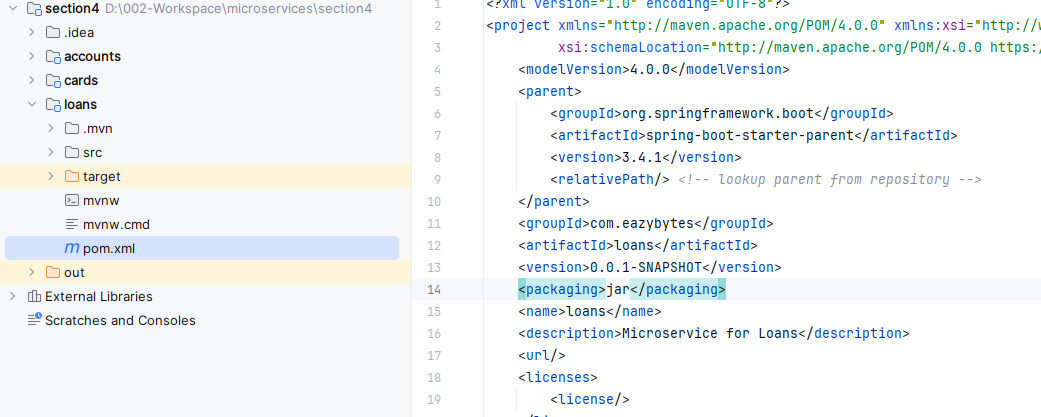
So what is a Buildpacks. Using Buildpacks, we can transform our application source code into a Docker image that can run on any cloud. There is no need of writing low level instructions with the help of Docker file. With a single maven command, we can generate a Docker image very easily. So this buildpacks is developed by Heroku. So initially they started this concept of Buildpacks. Later on both Pivotal and Heroku. They work together and they build a cloud native build packs and with these cloud native build packs, we can generate Docker images very easily because behind the scenes build a pack is going to scan all your source code, it is going to scan your dependencies and accordingly it is going to generate a Docker image.



So during the generation of this Docker image, this build packs is going to follow all the standards of Docker in terms of security, in terms of compressing, in terms of caching. So regardless, whatever standard you talk about, Buildpacks is going to take care of everything. And if you want to achieve the same result with the Docker file approach, you need to be a super, super expert at Docker and understanding all the standards of Docker. Since we are the developers and don't want to learn everything about Docker, it will be a wise decision to use Buildpacks instead of Docker file. Because already these build packs is built based upon years of experience at Heroku and Pivotal and they know all the standards that we need to follow while generating a Docker image. We can use this ready to solution instead of reinventing the wheel again and again. So Buildpacks is a framework or an ecosystem or a concept. Based upon this ecosystem, we have Pacchetto Buildpacks that we can use for Java based applications.



Apart from Java, it is also going to support many other languages like Go, Graalvm, Python, Ruby, PHP node. So if your microservice or if your web application is written in any of these languages, you can safely use buildpacks. And behind the scenes this buildpacks is going to use these Pacchetto build packs, which is an implementation of Buildpacks ecosystem. I hope you are clear with this quick introduction about Buildpacks. Now let's try to understand how to generate a Docker image with the help of this Buildpacks. For the same, first, let me close all these files of accounts microservice, post that I'm going to go to the loans microservice inside this pom.xml, first I need to make sure just after this version I'm mentioning the packaging as jar.



So once we mention this packaging as jar, you need to make sure you have a plugin related to the maven inside your pom.xml. Like you can see whenever we try to generate a spring boot web application from start.spring.io website. By default you will get this plugin which is like Spring boot maven plugin. With the help of this Spring boot Maven plugin.



We can generate a Docker image and behind the scenes this Spring Boot Maven plugin is going to leverage Buildpacks and Paquito to generate a production ready Docker image. But in order to generate a Docker image, first we need to provide what is the name that we want to consider for that Docker image that we are going to generate. That's why inside this configurations we need to invoke a tag which is image. So inside this image tag we need to invoke one more tag, which is name. Under this name tag we need to provide what is that Docker image name. The Docker image name that we want to follow is very similar to accounts microservice First, we need to make sure that we are mentioning the Docker username that we have created inside the Docker website. So my username is eazybytes, so please make sure you are mentioning your own Docker username, post that slash followed by what is the name of your application. The name of my application is loans. Again this name can be anything but you need to make sure with the help of this Docker image name, you should be able to identify your microservice. That's why I'm giving this name as loans, which is same as my microservice name. After this, we need to mention colon. And what is the tag name or what is the version name? The tag name that we want to follow here is s4, which indicates section 4, since this image belongs to the section 4, I'm mentioning this S4 here.

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<image>

<name>lalitrnagpal/loans:s4</name>

</image>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

</build>

</project>

Now, if you see, instead of hard coding these loans anyway we decided to use the same as microservice name. What we can do is we can try to read this application name with the help of this dollar and curly braces.

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<image>

<name>lalitrnagpal/${project.artifactId}:s4</name>

</image>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

</build>

</project>

So inside this I'm going to mention project.artifactId, this project.artifactId we have defined in the top. So the artifactId name is loans. So the same name comes here. This way I'm trying to avoid hardcoding my microservice name.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance> xsi:schemaLocation=

"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.4.1</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.eazybytes</groupId>

<artifactId>loans</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>loans</name>

<description>Microservice for Loans</description>

. . .

</project>

So with this we have mentioned that Docker image name that we want to consider whenever we are trying to generate a Docker image with the help of Buildpacks.

So let me save this changes post that I'll go to the terminal. So here, let me clean the console. Right now my terminal is at a location where my pom.xml of loans microservice is available here. I need to issue a maven command and the command is mvn spring-boot:build-image. So with this command we are telling to the maven to generate a docker image of my spring boot microservice by leveraging buildpacks. So behind the scenes it is going to leverage Buildpacks. Let me press enter here and if you are doing this very first time, it is going to take some five minutes because it has to download all the Buildpacks or package related libraries and images inside your local system. You can see right now it is downloading the base image from the paketo buildpacks using this base image only it is going to generate a Docker image. That's why very first time it is going to take some time. But once you have done this from next time onwards, it is going to be quick compared to the very first time. So let's wait for this to complete.

PS D:\002-Workspace\microservices\section4\loans> mvn spring-boot:build-image

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------< com.eazybytes:loans >-------------------------

[INFO] Building loans 0.0.1-SNAPSHOT

[INFO] from pom.xml

[INFO] --------------------------------[ jar ]---------------------------------

[INFO]

[INFO] >>> spring-boot:3.4.1:build-image (default-cli) > package @ loans >>>

[INFO]

[INFO] --- resources:3.3.1:resources (default-resources) @ loans ---

[INFO] Copying 1 resource from src\main\resources to target\classes

[INFO] Copying 1 resource from src\main\resources to target\classes

[INFO]

[INFO] --- compiler:3.13.0:compile (default-compile) @ loans ---

[INFO] Nothing to compile - all classes are up to date.

[INFO]

[INFO] --- resources:3.3.1:testResources (default-testResources) @ loans ---

[INFO] skip non existing resourceDirectory D:\002-Workspace\microservices\section4\loans\src\test\resources

[INFO]

[INFO] --- compiler:3.13.0:testCompile (default-testCompile) @ loans ---

[INFO] Nothing to compile - all classes are up to date.

[INFO]

[INFO] --- surefire:3.5.2:test (default-test) @ loans ---

[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JunitPlatformProvider

. . .

PS D:\002-Workspace\microservices\section4\loans> mvn spring-boot:build-image

. . .

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 0%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 5%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 9%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 17%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 23%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 29%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 32%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 34%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 35%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 36%

[INFO] > Pulling builder image 'docker.io/paketobuildpacks/builder-jammy-java-tiny:latest' 100%

[INFO] > Pulled builder image 'paketobuildpacks/builder-jammy-java-tiny@sha256:da6ccadf959933cb4c2c463e9e75527174cd966fcd211335a0133efc087d90ac'

. . .

Right now it is 36% post at 41% and please make sure while running this mvn command, your Docker server should be running behind the scenes. If you don't have your Docker server is running your Buildpacks can't do anything because at the end of the day, behind the scenes it is going to give instructions to your Docker server to generate a Docker image inside your local system. So you can see it is trying to understand my Java version, which version I'm using inside my pom.xml So inside my pom.xml I have mentioned the Java version as 17. If you can go to the top of this pom.xml, here under this properties we have mentioned the Java version as 17.

So based upon all the details and dependencies that we have mentioned inside the pom.xml, it is going to scan for all the dependencies. And by considering all these behind the scenes it is going to generate a Docker image. Finally, after waiting for few more minutes, my Docker image generation of loans microservice got successful with the help of Buildpacks. You can see a Docker image is built with this image name. Now I can try to clean this console and run the docker images command. So here I'm trying to run the same and this time you can see there is also a loans image with the tag S4 and the size is 311 MB.

[INFO] [creator] Saving docker.io/lalitrnagpal/loans:s4...

[INFO] [creator] \*\*\* Images (048cf5d6b293):

[INFO] [creator] docker.io/lalitrnagpal/loans:s4

[INFO] [creator] Adding cache layer 'paketo-buildpacks/syft:syft'

[INFO] [creator] Adding cache layer 'paketo-buildpacks/spring-boot:spring-cloud-bindings'

[INFO] [creator] Adding cache layer 'buildpacksio/lifecycle:cache.sbom'

[INFO]

[INFO] Successfully built image 'docker.io/lalitrnagpal/loans:s4'

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

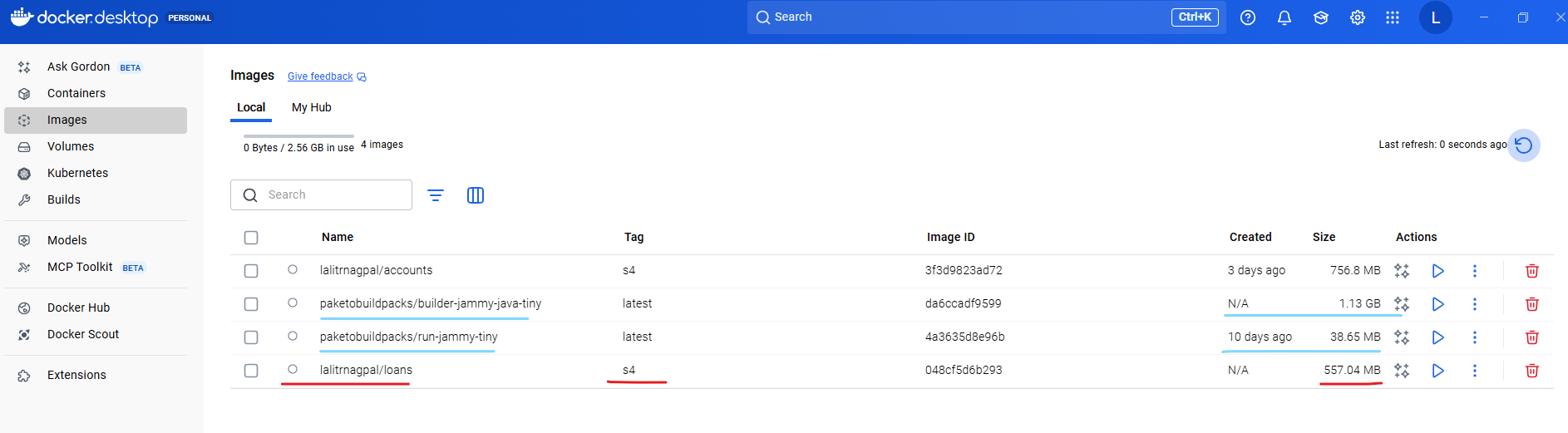
[INFO] ------------------------------------------------------------------------

[INFO] Total time: 02:03 min

[INFO] Finished at: 2025-10-27T07:26:15+05:30

[INFO] ------------------------------------------------------------------------

PS D:\002-Workspace\microservices\section4\loans>



PS D:\002-Workspace\microservices\section4\loans> docker image list

REPOSITORY TAG IMAGE ID CREATED SIZE

lalitrnagpal/accounts s4 3f3d9823ad72 2 days ago 757MB

paketobuildpacks/run-jammy-tiny latest 4a3635d8e96b 9 days ago 38.7MB

lalitrnagpal/loans s4 048cf5d6b293 45 years ago 557MB

paketobuildpacks/builder-jammy-java-tiny latest da6ccadf9599 45 years ago 1.13GB

PS D:\002-Workspace\microservices\section4\loans>

If you see for accounts microservice, we have used the Docker file and the size of the accounts Docker image is 757 MB because we don't know how to follow the best standards, how to cache multiple layers, how to compress our multiple components inside Docker image. We don't know about all those standards and the Docker concept, but with the help of Buildpacks, you can see there is a lot of improvement in terms of the size of our Docker image. It reduced from 757 MB to 557 MB. So this is one of the great advantage if you use the products like Buildpacks, you can also see there is a paketo Buildpacks Docker image downloaded. So this image is going to do all the work of generating a Docker image for your spring boot microservice. And this is of 1.13 GB. So this is a very heavy image and of course you don't have to take this into your deployment. You just need only the loans Docker image. Now I can try to run my loans Docker image as a container.

PS D:\002-Workspace\. . .\loans> docker run -d -p 8081:8080 lalitrnagpal/loans:s4

6462e849d4ca6df9dc712a80942a23ac0360ad064393040dfdc88314d89f4593

PS D:\002-Workspace\microservices\section4\loans> docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6462e849d4ca lalitrnagpal/loans:s4 "/cnb/process/web" 5 seconds ago Up 4 seconds 0.0.0.0:8081->8080/tcp, [::]:8081->8080/tcp blissful\_cartwright

PS D:\002-Workspace\microservices\section4\loans> docker stop 6462e849d4ca

6462e849d4ca

For the same, the command is very similar if you go to the previous commands that we have executed. So this is the one which we have executed docker run -d -p and the loans microservice is going to start at the port 8090. First, I need to make sure I'm mentioning the same and I will try to expose at the same port which is 8090 and the Docker image name of loans is eazybytes/loans and the tag name is S4 only. So we should be good with that. Now, if I try to execute this command, a Docker container will get created behind the scenes. You may see this warning, don't worry, you can simply ignore that. Now if you can go to the Docker desktop first, under the images, you should be able to see the new image that got generated, which is lalitrnagpa/loans:s4.

PS D:\...\section4\loans> docker run -d -p 8090:8090 lalitrnagpal/loans:s4

3c0666f5f0b1d833014497aa53300f019a03f02df1383fb72de0d7e063d0706d

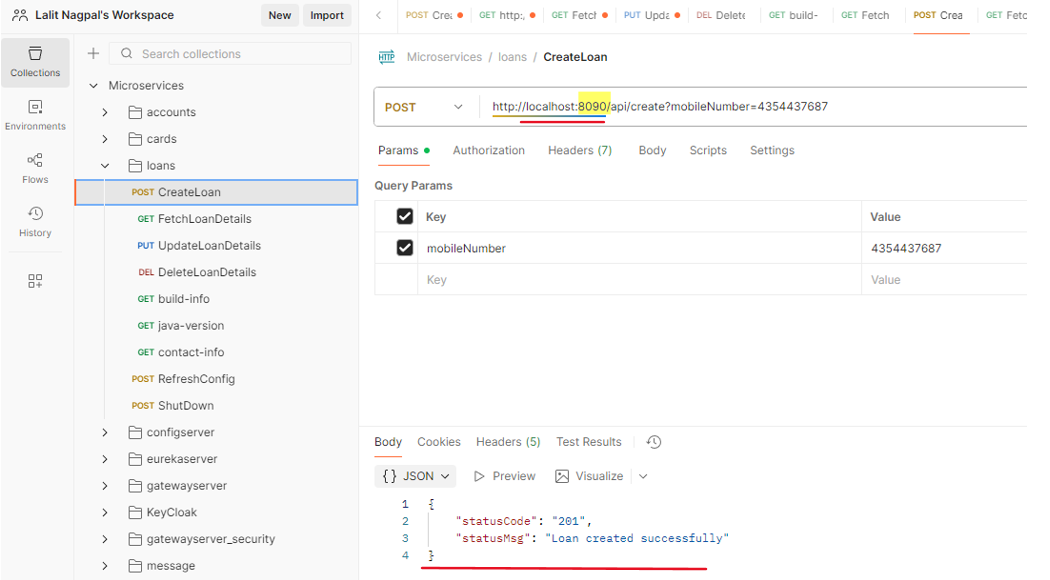
PS D:\002-Workspace\microservices\section4\loans> docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

3c0666f5f0b1 lalitrnagpal/loans:s4 "/cnb/process/web" 5 seconds ago Up 4 seconds 0.0.0.0:8090->8090/tcp, [::]:8090->8090/tcp silly\_edison

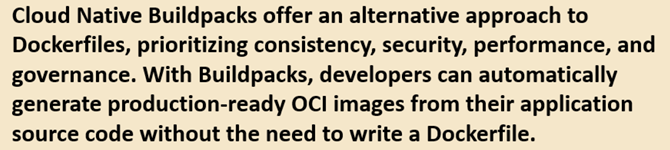
6462e849d4ca lalitrnagpal/loans:s4 "/cnb/process/web" About a minute ago Exited (143) 36 seconds ago blissful\_cartwright

And very similarly, if we can go to the containers, you can see there is a running container of loans. If you can click on this container name, you should be able to see all the logs and you can see your loans microservice started at the port 8090. I can also confirm the same by invoking a REST API against loans microservice with the help of postman. So here you can see, I'm trying to send a request for my create loan with the help of this mobile number. As soon as I click send, you should be able to get a response. So I'm getting a successful response here. So this confirms we have a proper running container of loans microservice that built with the help of Buildpacks concept.



So let's try to revise the steps very quickly that we followed inside this Buildpacks approach. The very first step that we need to follow while building a Docker image with the help of Buildpacks is, we need to add the image name details inside your pom.xml like you can see on the right hand side. Along with the image name, please make sure you also have the Spring boot Maven plugin configured inside your pom.xml. This comes by default inside your spring boot web application. If you don't have, please mention that by referring to the code present inside the GitHub repo. Post that we need to run a mvn command which is mvn spring boot build image. So when you try to run this command behind the scenes, your spring boot Maven plugin is going to utilize Buildpacks to generate a Docker image without the need of Docker file.

|  |  |
| --- | --- |
|  |  |



Once you generate a Docker image, you can run this Docker image as a Docker container by using the command which is Docker run along with the port mapping and your Docker image name. So these are the steps that we need to follow. So I have mentioned this inside this slide so that in future you can always refer them and you can always try to brush up the concepts that we have discussed. So on a high level, what is Buildpacks? Buildpacks offers alternative approach to Docker files. So without writing the Docker files this Buildpacks, it is going to help developers to automatically generate production ready Docker images from their application source code. And while generating these Docker images, these Buildpacks is going to follow all the production standards like security, caching, performance, compressing. So that's why this approach is better than Docker file approach. With this, I'm assuming your super clear about Buildpacks in the next lecture, let's try to explore the last approach that we have, which is Google Jib.

### Google Jib

Let's try to explore Jib from the Google to generate a Docker image for the cards microservice. So like you can see you need to go to this URL which is github.com slash Google container tool slash jib. The same URL I have mentioned inside the GitHub repository. Under the important links you can see this is the link of Google Jib website. Once you are onto this GitHub repository, you can scroll down and there are instructions on how to use this jib and containerize our Java applications. So please note that Jib is going to work only for Java applications, whereas Buildpacks is going to work for many other famous languages like Python, Ruby, Node.js. So there are many other languages that Buildpacks is going to support apart from Java, whereas Jib is strictly for Java applications. Since we are building Java based microservice, we can use this Jib without any issues. We can scroll down here. There are links to know more details about how to use this Jeep with the help of Maven and Gradle. If you are using Gradle, you can click on this Gradle link. Otherwise you can click on this Maven. Since we are using Maven, we need to click on this Maven link, which will redirect us to the GitHub repo page where this Maven plugin is available. If you can scroll down here, there are details on how to get started. The very first setup that you need to do is inside your Pom.xml. We need to make sure to copy this plugin detail which will help us to generate a Docker image. So let me copy these configurations. Now I'll go to my cards microservice. So here inside my cards microservice, first I'll try to delete the existing content of target folder. For that I'll go to my Pom.xml and I'll make sure to mention the packaging details after this versioning. So the packaging configurations that I want to use is Jar. So let me mention the same. Now we can go to the end of this palm dot XML. We already have a Spring Boot Maven plugin here, so we can introduce one more plugin. So let me copy the new plugin details. So the plugin details is it belongs to a group ID comm dot google dot cloud dot tools. And the artifact ID is Maven plugin. The version is 3.3.2. And under this configuration we need to mention a tag which is two. After this we can mention the image and what is the image name? The image name we can follow the very similar to what we have been following for the loans microservice. So let me copy the name from the loans microservice. So I'll take this value and I'll mention the same under the cards palm dot XML as well. So Easybeats is the Docker user name that I have created. And the project dot artifact id of this Pom.xml is cards. So there is no need to worry even though we copied from the loans. Since this variable is going to get derived from the existing pom.xml we should be good post that. The tag name that I want to mention here is S4. Once we added these changes inside Pom.xml, we can go to the GitHub repository here. If you can scroll down, there is a command that you can use to generate a Docker image. So the command is Maven compile jib colon docker build. So this will use your local Docker server setup to generate a Docker image. So let's try to execute this command inside my terminal. So for the same first I need to go to the cards microservice location. So I'm just going there was that I'm clearing my console. After this the command that I need to run is Maven compile jib followed by colon and docker build with B as capital. So if I try to execute this command, it will scan all the details inside my pom.xml and it is going to generate a docker image for my cards microservice. And this jib is going to be faster than build packs. You can see I'm able to get my Docker image within 11 seconds, whereas builder packs is going to take a lot of time. So now let me clear the console and try to run the Docker images. So if you can see here we got a new Docker image with the name cards. And this also is having a size of 322 MB, which is very similar to build a pack, but far better than the image size that we have for the accounts microservice. This confirms that Jib also is doing very good job in following all the standards and providing a production ready Docker image. Of course, this tool is from Google. We can confidently use this tool because they might be using this from many years and optimized it to a great extent. That's why we can always leverage these kind of open source tools like jib and build packs. Now here you may have question like for the images that got generated with the help of build packs and zip. There is a created value which is like 43 years ago or 53 years ago. Is this a bug inside this build? Packs and zip. No, this is not a bug. This is a feature provided by them. The reason on why they are using such old dates is they will have a starting date, which is somewhere in 1970 or 1960, which they are going to use whenever you try to generate a Docker image. And if you try to generate a Docker image again without making any changes, intentionally or unintentionally, this new Docker image is also going to have the same created it, and since it is going to have the same content inside it, the regeneration process of the same Docker image is going to be the same. But instead of this old created date, if they use a normal current created date, then the same is not going to work because they cannot really compare two images to identify if the same or not, because the created date is going to differ due to that reason, they are using older dates. You can also look about this these inside the web. Lot. Many people thought this is a bug inside the build packs and zip and why they are showing it as part three years or 53 years. But this is not a bug. This is a feature to optimize the generation of the Docker images again and again. So now my Docker image is available. As a next step we can run docker run command which is docker run, followed by hyphen d, followed by hyphen p. And the port mapping that I want to follow for the cards is 9000 9000. Since our cards microservice is going to start at 9000, I have mentioned this port mapping and the image name is easy byte slash cards and the tag name is S4. So now I'm trying to execute this command. You can see the new container started. I can go to the Docker desktop under the containers. Right now you can see there is a cards related container also running right now. I can also confirm by looking at the logs, you can see it is still starting and the logs are being updated behind the scenes. So now my cards application container is started. I can also validate if it is working properly or not. With the help of postman inside the postman, I'm trying to send a request to the create API available inside the card. So let me click on this send button. After clicking on the send button, you can see I'm getting a successful response, which means our cards microservice is also working fine. After converting as a Docker container, you can also try to run the Docker PS command which will show all the existing running containers. So these cards and loans containers are running and accounts microservice container is not running because I have stopped it long back. I hope you are also clear with the third approach, which is Google give and the steps that we need to follow before we try to close this lecture. Let me show you more details about Google Jib. There is a one more great advantage of Google Jib compared to Buildpacks. Whenever we are using Jib, we can generate a Docker image. Even if you don't have Docker installed inside your system. That's the beauty of jib. So if you don't want to install Docker inside your local system, but at the same time you want to convert your application to a Docker image, then you can use a command which is Maven compile jib build. So when you try to use this build command, it is going to generate a Docker image from your application. And the same will be pushed into the remote repository. Because if you're not running a Docker inside a local system, that means there is no place for your jib to generate the Docker image. That's why it will look for the remote repository details where it has to store. If you can see here there are many other remote repository details that you can configure. If you are using GCR to store your Docker image, you can mention this image name as gcr.io. And what is your GCP project name followed by what is your app name? And very similarly for ECR. Also we can configure image name by following these standards. Whereas for Docker Hub you can mention docker.io. What is your docker username and post that? What is your app name followed by tag name? Tag name is always optional. So whenever it is trying to push your Docker image into the remote repository, it may need some credentials like your Docker credentials or your Amazon credentials, or your GCP credentials. So to provide those details you can always look these documentation on how to provide. But in this course we are not going to use any of these approaches. Because first we want to store inside our local system. We want to test it. Once we satisfied then only we want to manually push into the remote repository. But this approach is going to be super, super helpful for the projects where the developers are not responsible to generate a Docker images, they are simply responsible to push the code into the GitHub repository inside the GitHub repository using CI, CD tools like Jenkins or GitHub actions or any other tool. You can write scripts to generate the Docker image from your application and inside your CI CD server you don't have to install this heavy software, which is Docker. In such scenarios, this is going to be super, super helpful. But like I said, inside this course we are going to use docker build command. Throughout the course. But I just wanted to highlight these advantages that we have with Jib. And this is the true differentiator for many projects where they are trying to prefer Jib compared to Buildpacks. Now let me show you the steps that we have followed to generate a container with the help of Google Jib. In the very first step, we need to configure the plugin details of Jib Maven plugin inside our Pom.xml. Like you can see on the right hand side and inside this plugin details. We should also configure what is the image name with the help of these two tag available under the configuration tag. Once we configure these details, we can run a command which is Maven compile jib docker build. And when we run this command, a Docker image will be generated inside your local system with the help of Docker server Were installed. And here we're not writing any Docker file. And the last step is once we have the Docker image available, you can run your Docker image as a container with the help of docker run command. And with that, your application is ready to accept requests in the form of Docker container. And a high level Google chip help us to generate production ready Docker images by following all the production standards like performance, security, caching, compressing and developers are not forced to write any Docker files, they can simply generate production ready images for their application source code without the need of writing any low level Docker file and if needed. Jib also is capable of generating a Docker image without even local docker setup. I hope you are clear with all the details that we discussed about Google Jib in the next lecture, let's try to compare three different approaches that we have, and try to select one of them for the rest of the course.